멀티 전략 운영 실전

제1장: 멀티 전략 운영의 필요성과 아키텍처 설계

1.1 왜 멀티 전략인가

단일 전략으로 자동매매 시스템을 운영하는 것은 하나의 주식에 전 재산을 투자하는 것과 같습니다. 시장 레짐이 변하면 한순간에 수익이 손실로 전환될 수 있습니다. 멀티 전략 운영은 다양한 시장 환경에서 안정적인 수익을 추구하는 핵심 방법론입니다.

멀티 전략의 핵심 이점은 다음과 같습니다:

- **분산 효과**: 전략 간 상관관계가 낮을수록 포트폴리오 변동성이 감소합니다
- **시장 적응력**: 추세장, 횡보장, 급등락장 각각에 특화된 전략이 번갈아 수익을 냅니다
- **리스크 관리**: 한 전략의 실패가 전체 시스템을 위협하지 않습니다
- **자본 효율성**: 유휴 자본을 다른 전략에 재배분할 수 있습니다

1.2 멀티 전략 아키텍처 패턴

멀티 전략 시스템의 아키텍처는 크게 세 가지 패턴으로 나뉩니다.

패턴 1: 독립 실행형 (Independent)

각 전략이 독립적인 프로세스로 실행되며, 서로 간섭하지 않습니다. 가장 단순하지만 자본 관리가 비효율적입니다.

```
# 독립 실행형 아키텍처 예시
import multiprocessing
from dataclasses import dataclass
from typing import Dict, Any

@dataclass
class StrategyConfig:
    name: str
    capital_ratio: float  # 전체 자본 대비 할당 비율
    params: Dict[str, Any]

class IndependentRunner:
```

```
    """각 전략을 독립 프로세스로 실행"""

    def __init__(self, total_capital: float):
        self.total_capital = total_capital
        self.processes: Dict[str, multiprocessing.Process] = {}

    def register_strategy(self, config: StrategyConfig, strategy_class):
        allocated = self.total_capital * config.capital_ratio
        process = multiprocessing.Process(
            target=self._run_strategy,
            args=(strategy_class, allocated, config.params),
            name=config.name
        )
        self.processes[config.name] = process

    def _run_strategy(self, strategy_class, capital, params):
        strategy = strategy_class(capital=capital, **params)
        strategy.run()

    def start_all(self):
        for name, proc in self.processes.items():
            proc.start()
            print(f"[시작] {name} 전략 프로세스 PID: {proc.pid}")

    def stop_all(self):
        for name, proc in self.processes.items():
            proc.terminate()
            proc.join(timeout=10)
            print(f"[중지] {name} 전략")
```

패턴 2: 중앙 집중형 (Centralized Orchestrator)

중앙 오케스트레이터가 모든 전략을 관리하며, 자본 배분과 리스크 관리를 통합적으로 수행합니다.

```
import asyncio
from abc import ABC, abstractmethod
from typing import List, Optional
from datetime import datetime
from enum import Enum

class StrategyState(Enum):
    IDLE = "idle"
    RUNNING = "running"
    PAUSED = "paused"
    ERROR = "error"

class BaseStrategy(ABC):
    """모든 전략의 기본 클래스"""

    def __init__(self, name: str):
```

```
        self.name = name
        self.state = StrategyState.IDLE
        self.allocated_capital = 0.0
        self.current_pnl = 0.0
        self.positions = {}
        self.last_signal_time: Optional[datetime] = None

    @abstractmethod
    async def generate_signal(self, market_data: dict) -> Optional[dict]:
        """시장 데이터를 받아 매매 신호 생성"""
        pass

    @abstractmethod
    async def on_fill(self, fill_data: dict):
        """주문 체결 시 콜백"""
        pass

    def get_exposure(self) -> float:
        """현재 포지션 노출도 계산"""
        return sum(
            abs(pos['quantity'] * pos['current_price'])
            for pos in self.positions.values()
        )

class CentralOrchestrator:
    """중앙 집중형 전략 오케스트레이터"""

    def __init__(self, total_capital: float):
        self.total_capital = total_capital
        self.strategies: List[BaseStrategy] = []
        self.risk_manager = RiskManager(total_capital)
        self.order_router = OrderRouter()
        self.running = False

    def add_strategy(self, strategy: BaseStrategy, capital_ratio: float):
        strategy.allocated_capital = self.total_capital * capital_ratio
        strategy.state = StrategyState.IDLE
        self.strategies.append(strategy)

    async def run(self):
        self.running = True
        while self.running:
            market_data = await self._fetch_market_data()

            tasks = []
            for strategy in self.strategies:
                if strategy.state == StrategyState.RUNNING:
                    tasks.append(
                        self._process_strategy(strategy, market_data)
                    )

            await asyncio.gather(*tasks)
            await asyncio.sleep(0.1)
```

```
    async def _process_strategy(self, strategy: BaseStrategy, market_data: dict):
        try:
            signal = await strategy.generate_signal(market_data)
            if signal is None:
                return

            # 리스크 매니저에서 주문 검증
            approved = self.risk_manager.validate_order(
                strategy=strategy,
                signal=signal,
                portfolio_state=self._get_portfolio_state()
            )

            if approved:
                await self.order_router.submit(signal)
                strategy.last_signal_time = datetime.now()

        except Exception as e:
            strategy.state = StrategyState.ERROR
            print(f"[오류] {strategy.name}: {e}")

    def _get_portfolio_state(self) -> dict:
        return {
            'total_capital': self.total_capital,
            'total_exposure': sum(s.get_exposure() for s in self.strategies),
            'total_pnl': sum(s.current_pnl for s in self.strategies),
            'strategy_states': {
                s.name: s.state.value for s in self.strategies
            }
        }

    async def _fetch_market_data(self) -> dict:
        # 실제 구현에서는 거래소 웹소켓 데이터 수신
        return {}
```

패턴 3: 이벤트 기반 (Event-Driven)

메시지 큐를 사용하여 전략 간 느슨한 결합을 유지하면서도 중앙 관리의 이점을 취합니다.

```
import asyncio
from collections import defaultdict
from typing import Callable, Any

class EventBus:
    """전략 간 이벤트 통신을 위한 이벤트 버스"""

    def __init__(self):
        self._subscribers = defaultdict(list)
        self._queue = asyncio.Queue()
```

```
    def subscribe(self, event_type: str, handler: Callable):
        self._subscribers[event_type].append(handler)

    async def publish(self, event_type: str, data: Any):
        await self._queue.put((event_type, data))

    async def process_events(self):
        while True:
            event_type, data = await self._queue.get()
            for handler in self._subscribers[event_type]:
                try:
                    await handler(data)
                except Exception as e:
                    print(f"이벤트 처리 오류 [{event_type}]: {e}")

class EventDrivenStrategy(BaseStrategy):
    """이벤트 기반 전략"""

    def __init__(self, name: str, event_bus: EventBus):
        super().__init__(name)
        self.event_bus = event_bus
        # 관심 이벤트 구독
        event_bus.subscribe('market_data', self.on_market_data)
        event_bus.subscribe('fill', self.on_fill)
        event_bus.subscribe('risk_alert', self.on_risk_alert)

    async def on_market_data(self, data: dict):
        signal = await self.generate_signal(data)
        if signal:
            await self.event_bus.publish('order_request', {
                'strategy': self.name,
                'signal': signal
            })

    async def on_risk_alert(self, data: dict):
        if data.get('action') == 'reduce_exposure':
            await self._reduce_positions(data['target_ratio'])

    async def _reduce_positions(self, target_ratio: float):
        for symbol, pos in self.positions.items():
            reduce_qty = pos['quantity'] * (1 - target_ratio)
            await self.event_bus.publish('order_request', {
                'strategy': self.name,
                'signal': {
                    'symbol': symbol,
                    'side': 'sell' if pos['quantity'] > 0 else 'buy',
                    'quantity': abs(reduce_qty),
                    'type': 'market'
                }
            })
```

1.3 아키텍처 선택 기준

기준	독립 실행형	중앙 집중형	이벤트 기반
구현 난이도	낮음	중간	높음
자본 효율성	낮음	높음	높음
전략 간 간섭	없음	가능	최소
확장성	좋음	보통	매우 좋음
장애 격리	완벽	취약	좋음
적합 전략 수	2~5개	5~20개	10개 이상

초보자는 독립 실행형으로 시작하여 점진적으로 중앙 집중형이나 이벤트 기반으로 마이그레이션하는 것을 권장합니다.

제2장: 전략 포트폴리오 구성과 상관관계 분석

2.1 전략 분류 체계

효과적인 멀티 전략 포트폴리오를 구성하려면 먼저 전략을 체계적으로 분류해야 합니다.

수익 원천별 분류:

- **추세 추종 (Trend Following)**: 가격 모멘텀을 포착하여 수익. 추세장에서 강함
- **평균 회귀 (Mean Reversion)**: 가격이 평균으로 돌아오는 성질을 이용. 횡보장에서 강함
- **차익거래 (Arbitrage)**: 시장 비효율성을 포착. 시장 중립적
- **마켓 메이킹 (Market Making)**: 호가 스프레드에서 수익. 유동성 제공
- **이벤트 드리븐 (Event Driven)**: 특정 이벤트 발생 시 포지션 진입

시간 프레임별 분류:

- **초단타 (Scalping)**: 수초~수분, 높은 거래 빈도
- **단기 (Intraday)**: 수분~수시간, 일중 포지션 청산
- **중기 (Swing)**: 수일~수주, 추세 포착

- **장기 (Position)**: 수주~수개월, 거시 트렌드

2.2 전략 상관관계 분석

멀티 전략의 핵심은 상관관계가 낮은 전략들을 조합하는 것입니다.

```
import numpy as np
import pandas as pd
from scipy import stats

class StrategyCorrelationAnalyzer:
    """전략 간 상관관계를 분석하는 클래스"""

    def __init__(self):
        self.returns_data = {}

    def add_strategy_returns(self, name: str, returns: pd.Series):
        """전략별 일별 수익률 데이터 추가"""
        self.returns_data[name] = returns

    def compute_correlation_matrix(self) -> pd.DataFrame:
        """전략 간 상관계수 행렬 계산"""
        df = pd.DataFrame(self.returns_data)
        return df.corr(method='pearson')

    def compute_rolling_correlation(
        self, strategy_a: str, strategy_b: str, window: int = 30
    ) -> pd.Series:
        """두 전략 간 롤링 상관계수 계산"""
        returns_a = self.returns_data[strategy_a]
        returns_b = self.returns_data[strategy_b]
        return returns_a.rolling(window).corr(returns_b)

    def find_optimal_combination(
        self, max_strategies: int = 5
    ) -> list:
        """상관관계 기반 최적 전략 조합 탐색"""
        df = pd.DataFrame(self.returns_data)
        corr_matrix = df.corr()
        strategy_names = list(self.returns_data.keys())

        # 탐욕적 알고리즘으로 낮은 상관관계 전략 선택
        selected = [strategy_names[0]]

        while len(selected) < max_strategies and len(selected) < len(strategy_names):
            best_candidate = None
            lowest_avg_corr = float('inf')

            for candidate in strategy_names:
                if candidate in selected:
                    continue
```

```
                # 이미 선택된 전략들과의 평균 상관계수 계산
                avg_corr = np.mean([
                    abs(corr_matrix.loc[candidate, s])
                    for s in selected
                ])

                if avg_corr < lowest_avg_corr:
                    lowest_avg_corr = avg_corr
                    best_candidate = candidate

            if best_candidate:
                selected.append(best_candidate)
                print(f"선택: {best_candidate} (평균 상관계수: {lowest_avg_corr:.4f})")

        return selected

    def regime_conditional_correlation(
        self, strategy_a: str, strategy_b: str,
        market_returns: pd.Series
    ) -> dict:
        """시장 레짐별 조건부 상관관계 분석"""
        returns_a = self.returns_data[strategy_a]
        returns_b = self.returns_data[strategy_b]

        # 시장 레짐 분류: 상승, 하락, 횡보
        bull = market_returns > market_returns.quantile(0.67)
        bear = market_returns < market_returns.quantile(0.33)
        sideways = ~bull & ~bear

        results = {}
        for regime, mask in [('상승장', bull), ('하락장', bear), ('횡보장', sideways)]:
            a_regime = returns_a[mask]
            b_regime = returns_b[mask]
            if len(a_regime) > 5:
                corr, p_value = stats.pearsonr(a_regime, b_regime)
                results[regime] = {
                    'correlation': round(corr, 4),
                    'p_value': round(p_value, 4),
                    'sample_count': len(a_regime)
                }

        return results
```

2.3 전략 분산 효과 측정

```
class DiversificationMetrics:
    """전략 분산 효과를 정량적으로 측정"""

    @staticmethod
    def diversification_ratio(weights: np.ndarray, cov_matrix: np.ndarray) -> float:
```

```
        """
    분산 비율(DR) 계산
        DR = (가중 평균 변동성) / (포트폴리오 변동성)
        DR > 1이면 분산 효과가 있음
        """
        individual_vols = np.sqrt(np.diag(cov_matrix))
        weighted_avg_vol = np.dot(weights, individual_vols)
        portfolio_vol = np.sqrt(weights @ cov_matrix @ weights)
        return weighted_avg_vol / portfolio_vol

    @staticmethod
    def marginal_contribution_to_risk(
        weights: np.ndarray, cov_matrix: np.ndarray
    ) -> np.ndarray:
        """각 전략의 한계 리스크 기여도 계산"""
        portfolio_vol = np.sqrt(weights @ cov_matrix @ weights)
        mcr = (cov_matrix @ weights) / portfolio_vol
        return mcr

    @staticmethod
    def strategy_contribution_report(
        strategy_names: list,
        weights: np.ndarray,
        cov_matrix: np.ndarray
    ) -> pd.DataFrame:
        """전략별 리스크 기여도 보고서 생성"""
        portfolio_var = weights @ cov_matrix @ weights
        portfolio_vol = np.sqrt(portfolio_var)

        mcr = (cov_matrix @ weights) / portfolio_vol
        component_risk = weights * mcr
        pct_contribution = component_risk / portfolio_vol * 100

        report = pd.DataFrame({
            '전략': strategy_names,
            '비중': weights,
            '한계리스크기여': mcr,
            '리스크기여': component_risk,
            '리스크기여율(%)': pct_contribution
        })

        return report
```

2.4 전략 포트폴리오 구성 예시

실전에서 효과적인 멀티 전략 포트폴리오 구성 예시입니다:

보수적 포트폴리오 (변동성 최소화): - 마켓 메이킹 전략: 40% 자본 배분 - 차익거래 전략: 30% 자본 배분 - 평균 회귀 전략: 20% 자본 배분 - 현금 보유: 10%

공격적 포트폴리오 (수익 극대화): - 추세 추종 전략: 35% 자본 배분 - 모멘텀 전략: 25% 자본 배분 - 이벤트 드리븐 전략: 20% 자본 배분 - 평균 회귀 전략: 15% 자본 배분 - 현금 보유: 5%

균형 포트폴리오 (추천): - 추세 추종 전략: 25% 자본 배분 - 평균 회귀 전략: 25% 자본 배분 - 차익거래 전략: 20% 자본 배분 - 마켓 메이킹 전략: 15% 자본 배분 - 이벤트 드리븐 전략: 10% 자본 배분 - 현금 보유: 5%

제3장: 동적 자본 배분 시스템

3.1 자본 배분의 원칙

정적 자본 배분은 시장 변화에 적응하지 못합니다. 동적 자본 배분 시스템은 각 전략의 실시간 성과와 시장 상황에 따라 자본을 재배분합니다.

```
from dataclasses import dataclass, field
from typing import Dict, List, Optional
from datetime import datetime, timedelta
import numpy as np

@dataclass
class StrategyPerformance:
    name: str
    daily_returns: List[float] = field(default_factory=list)
    current_drawdown: float = 0.0
    max_drawdown: float = 0.0
    sharpe_ratio: float = 0.0
    win_rate: float = 0.0
    current_allocation: float = 0.0
    min_allocation: float = 0.05  # 최소 5%
    max_allocation: float = 0.40  # 최대 40%

    @property
    def recent_sharpe(self, lookback: int = 30) -> float:
        if len(self.daily_returns) < lookback:
            return 0.0
        recent = self.daily_returns[-lookback:]
        mean_ret = np.mean(recent)
        std_ret = np.std(recent)
        if std_ret == 0:
            return 0.0
        return mean_ret / std_ret * np.sqrt(252)


class DynamicAllocator:
    """동적 자본 배분 엔진"""

    def __init__(
```

```
        self,
        total_capital: float,
        rebalance_interval: timedelta = timedelta(hours=24),
        max_single_rebalance: float = 0.10  # 1회 최대 10% 변경
    ):
        self.total_capital = total_capital
        self.rebalance_interval = rebalance_interval
        self.max_single_rebalance = max_single_rebalance
        self.strategies: Dict[str, StrategyPerformance] = {}
        self.last_rebalance: Optional[datetime] = None
        self.allocation_history: List[dict] = []

    def add_strategy(self, perf: StrategyPerformance):
        self.strategies[perf.name] = perf

    def should_rebalance(self) -> bool:
        if self.last_rebalance is None:
            return True
        return datetime.now() - self.last_rebalance >= self.rebalance_interval

    def compute_target_allocations(self) -> Dict[str, float]:
        """성과 기반 목표 배분 비율 계산"""
        scores = {}

        for name, perf in self.strategies.items():
            # 복합 점수 계산: 샤프비율 + 승률 - 드로다운 페널티
            sharpe_score = max(perf.recent_sharpe, 0)
            win_score = perf.win_rate
            dd_penalty = perf.current_drawdown * 2  # 드로다운에 2배 가중

            score = (sharpe_score * 0.5) + (win_score * 0.3) - (dd_penalty * 0.2)
            scores[name] = max(score, 0.01)  # 최소 점수 보장

        # 점수 기반 비율 계산
        total_score = sum(scores.values())
        raw_allocations = {
            name: score / total_score
            for name, score in scores.items()
        }

        # 최소/최대 제약 적용
        adjusted = self._apply_constraints(raw_allocations)
        return adjusted

    def _apply_constraints(self, allocations: Dict[str, float]) -> Dict[str, float]:
        """배분 비율에 최소/최대 제약 적용"""
        adjusted = {}
        excess = 0.0

        for name, alloc in allocations.items():
            perf = self.strategies[name]
            if alloc < perf.min_allocation:
                adjusted[name] = perf.min_allocation
```

```
                excess += alloc - perf.min_allocation
            elif alloc > perf.max_allocation:
                adjusted[name] = perf.max_allocation
                excess += alloc - perf.max_allocation
            else:
                adjusted[name] = alloc

        # 초과/부족분 재배분
        unconstrained = [
            n for n in adjusted
            if self.strategies[n].min_allocation < adjusted[n] < self.strategies[n].max_allo
        ]

        if unconstrained and abs(excess) > 0.001:
            share = excess / len(unconstrained)
            for name in unconstrained:
                adjusted[name] += share

        # 합이 1이 되도록 정규화
        total = sum(adjusted.values())
        return {name: alloc / total for name, alloc in adjusted.items()}

    def execute_rebalance(self) -> Dict[str, dict]:
        """리밸런싱 실행: 현재 배분 → 목표 배분으로 점진적 이동"""
        target = self.compute_target_allocations()
        changes = {}

        for name, target_alloc in target.items():
            current = self.strategies[name].current_allocation
            diff = target_alloc - current

            # 급격한 변경 방지
            if abs(diff) > self.max_single_rebalance:
                diff = self.max_single_rebalance * np.sign(diff)

            new_alloc = current + diff
            self.strategies[name].current_allocation = new_alloc

            changes[name] = {
                'previous': round(current, 4),
                'target': round(target_alloc, 4),
                'new': round(new_alloc, 4),
                'change': round(diff, 4),
                'capital': round(self.total_capital * new_alloc, 2)
            }

        self.last_rebalance = datetime.now()
        self.allocation_history.append({
            'timestamp': self.last_rebalance.isoformat(),
            'allocations': {n: s.current_allocation for n, s in self.strategies.items()}
        })

        return changes
```

3.2 드로다운 기반 자본 축소

전략의 드로다운이 일정 수준을 넘으면 자동으로 자본을 축소하여 손실을 제한합니다.

```
class DrawdownGuard:
    """드로다운 기반 자본 보호 시스템"""

    def __init__(self):
        self.thresholds = [
            (0.05, 0.80),  # 5% 드로다운 → 80%로 축소
            (0.10, 0.50),  # 10% 드로다운 → 50%로 축소
            (0.15, 0.25),  # 15% 드로다운 → 25%로 축소
            (0.20, 0.00),  # 20% 드로다운 → 전략 중단
        ]

    def get_scaling_factor(self, current_drawdown: float) -> float:
        """현재 드로다운에 따른 스케일링 팩터 반환"""
        scaling = 1.0
        for dd_threshold, scale in self.thresholds:
            if current_drawdown >= dd_threshold:
                scaling = scale

        return scaling

    def evaluate_strategy(self, perf: StrategyPerformance) -> dict:
        """전략 상태 평가 및 권고 사항 반환"""
        scaling = self.get_scaling_factor(perf.current_drawdown)
        adjusted_capital = perf.current_allocation * scaling

        status = 'normal'
        if scaling == 0:
            status = 'stopped'
        elif scaling < 0.5:
            status = 'critical'
        elif scaling < 1.0:
            status = 'warning'

        return {
            'strategy': perf.name,
            'drawdown': f"{perf.current_drawdown:.2%}",
            'scaling_factor': scaling,
            'adjusted_allocation': adjusted_capital,
            'status': status,
            'recommendation': self._get_recommendation(status, perf)
        }

    def _get_recommendation(self, status: str, perf: StrategyPerformance) -> str:
        recommendations = {
            'normal': '정상 운영 중',
            'warning': f'자본 축소 적용. 최근 성과 모니터링 필요',
            'critical': f'대폭 자본 축소. 전략 파라미터 재검토 권장',
            'stopped': f'전략 자동 중단. 수동 검토 후 재시작 필요'
```

```
        }
        return recommendations.get(status, '')
```

3.3 켈리 기준 기반 자본 배분

켈리 공식을 활용한 수학적 최적 배분 비율 계산:

```
class KellyAllocator:
    """켈리 기준 기반 최적 자본 배분"""

    def __init__(self, fraction: float = 0.5):
        """
        fraction: 풀 켈리 대비 사용 비율
    실전에서는 0.25~0.5 (쿼터~하프 켈리) 권장
        """
        self.fraction = fraction

    def kelly_criterion(
        self, win_rate: float, avg_win: float, avg_loss: float
    ) -> float:
        """
    켈리 공식: f = (bp - q) / b
        b = 평균 수익/평균 손실 비율
        p = 승률
        q = 패률 (1 - p)
        """
        if avg_loss == 0:
            return 0.0

        b = avg_win / avg_loss
        p = win_rate
        q = 1 - p

        kelly = (b * p - q) / b
        return max(kelly * self.fraction, 0)

    def multi_strategy_kelly(
        self, strategies: Dict[str, StrategyPerformance]
    ) -> Dict[str, float]:
        """여러 전략의 켈리 비율 동시 계산"""
        kelly_fractions = {}

        for name, perf in strategies.items():
            returns = np.array(perf.daily_returns)
            if len(returns) < 30:
                kelly_fractions[name] = perf.min_allocation
                continue

            wins = returns[returns > 0]
            losses = returns[returns < 0]
```

```
            if len(wins) == 0 or len(losses) == 0:
                kelly_fractions[name] = perf.min_allocation
                continue

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

            kelly_f = self.kelly_criterion(win_rate, avg_win, avg_loss)

            # 제약 조건 적용
            kelly_f = np.clip(kelly_f, perf.min_allocation, perf.max_allocation)
            kelly_fractions[name] = kelly_f

        # 합이 1을 넘지 않도록 정규화
        total = sum(kelly_fractions.values())
        if total > 1.0:
            kelly_fractions = {
                k: v / total for k, v in kelly_fractions.items()
            }

        return kelly_fractions
```

제4장: 통합 리스크 관리 시스템

4.1 포트폴리오 수준 리스크 관리

개별 전략의 리스크 관리만으로는 충분하지 않습니다. 멀티 전략 환경에서는 포트폴리오 전체의 리스크를 통합적으로 관리해야 합니다.

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

@dataclass
class RiskLimits:
    """포트폴리오 리스크 한도 설정"""
    max_total_exposure: float = 0.80       # 전체 자본 대비 최대 노출
    max_single_strategy_exposure: float = 0.30  # 단일 전략 최대 노출
    max_daily_loss: float = 0.03           # 일일 최대 손실 3%
    max_weekly_loss: float = 0.07          # 주간 최대 손실 7%
    max_correlation_exposure: float = 0.50  # 상관 전략 합산 최대 노출
    max_single_asset_exposure: float = 0.20 # 단일 자산 최대 노출
    emergency_stop_loss: float = 0.10       # 비상 정지 손실 10%

class PortfolioRiskManager:
```

```
    """포트폴리오 통합 리스크 관리"""

    def __init__(self, total_capital: float, limits: RiskLimits):
        self.total_capital = total_capital
        self.limits = limits
        self.daily_pnl_history: List[Tuple[datetime, float]] = []
        self.risk_events: List[dict] = []
        self.is_emergency_stopped = False

    def validate_order(
        self,
        strategy: BaseStrategy,
        signal: dict,
        portfolio_state: dict
    ) -> bool:
        """주문 전 리스크 검증"""
        if self.is_emergency_stopped:
            self._log_risk_event('BLOCKED', '비상 정지 활성화', strategy.name)
            return False

        checks = [
            self._check_total_exposure(signal, portfolio_state),
            self._check_strategy_exposure(strategy, signal),
            self._check_daily_loss_limit(),
            self._check_single_asset_concentration(signal, portfolio_state),
            self._check_correlated_exposure(strategy, signal, portfolio_state),
        ]

        for passed, reason in checks:
            if not passed:
                self._log_risk_event('REJECTED', reason, strategy.name)
                return False

        return True

    def _check_total_exposure(
        self, signal: dict, portfolio_state: dict
    ) -> Tuple[bool, str]:
        """전체 포트폴리오 노출도 검사"""
        current_exposure = portfolio_state['total_exposure']
        order_value = signal.get('quantity', 0) * signal.get('price', 0)
        projected = (current_exposure + order_value) / self.total_capital

        if projected > self.limits.max_total_exposure:
            return False, f"총 노출도 초과: {projected:.2%} > {self.limits.max_total_exposure:.
        return True, ""

    def _check_strategy_exposure(
        self, strategy: BaseStrategy, signal: dict
    ) -> Tuple[bool, str]:
        """단일 전략 노출도 검사"""
        current = strategy.get_exposure()
        order_value = signal.get('quantity', 0) * signal.get('price', 0)
```

```
        projected = (current + order_value) / self.total_capital

        if projected > self.limits.max_single_strategy_exposure:
            return False, f"{strategy.name} 노출도 초과: {projected:.2%}"
        return True, ""

    def _check_daily_loss_limit(self) -> Tuple[bool, str]:
        """일일 손실 한도 검사"""
        today = datetime.now().date()
        today_pnl = sum(
            pnl for ts, pnl in self.daily_pnl_history
            if ts.date() == today
        )
        daily_loss_pct = abs(min(today_pnl, 0)) / self.total_capital

        if daily_loss_pct >= self.limits.max_daily_loss:
            return False, f"일일 손실 한도 도달: {daily_loss_pct:.2%}"
        return True, ""

    def _check_single_asset_concentration(
        self, signal: dict, portfolio_state: dict
    ) -> Tuple[bool, str]:
        """단일 자산 집중도 검사"""
        symbol = signal.get('symbol', '')
        # 모든 전략에서 해당 자산의 총 노출 계산
        total_asset_exposure = 0
        for s_name, s_data in portfolio_state.get('strategy_positions', {}).items():
            if symbol in s_data:
                total_asset_exposure += abs(s_data[symbol].get('value', 0))

        order_value = signal.get('quantity', 0) * signal.get('price', 0)
        projected = (total_asset_exposure + order_value) / self.total_capital

        if projected > self.limits.max_single_asset_exposure:
            return False, f"{symbol} 자산 집중도 초과: {projected:.2%}"
        return True, ""

    def _check_correlated_exposure(
        self, strategy: BaseStrategy, signal: dict,
        portfolio_state: dict
    ) -> Tuple[bool, str]:
        """상관관계 높은 전략들의 합산 노출 검사"""
        # 상관관계 0.7 이상인 전략들을 그룹으로 묶어 합산 검사
        correlated_groups = portfolio_state.get('correlated_groups', {})

        for group_name, group_strategies in correlated_groups.items():
            if strategy.name in group_strategies:
                group_exposure = sum(
                    portfolio_state.get('strategy_exposures', {}).get(s, 0)
                    for s in group_strategies
                )
                if group_exposure / self.total_capital > self.limits.max_correlation_exposur
                    return False, f"상관 그룹 '{group_name}' 노출 초과"
```

```
        return True, ""

    def update_pnl(self, pnl: float):
        """PnL 업데이트 및 비상 정지 검사"""
        self.daily_pnl_history.append((datetime.now(), pnl))

        # 누적 손실 검사
        total_loss = sum(p for _, p in self.daily_pnl_history if p < 0)
        if abs(total_loss) / self.total_capital >= self.limits.emergency_stop_loss:
            self.is_emergency_stopped = True
            self._log_risk_event(
                'EMERGENCY_STOP',
                f"비상 정지 발동: 누적 손실 {abs(total_loss)/self.total_capital:.2%}",
                'SYSTEM'
            )

    def _log_risk_event(self, event_type: str, reason: str, strategy: str):
        self.risk_events.append({
            'timestamp': datetime.now().isoformat(),
            'type': event_type,
            'reason': reason,
            'strategy': strategy
        })
```

4.2 전략 간 충돌 감지 및 해결

여러 전략이 같은 자산에 대해 상반된 포지션을 취하는 충돌 상황을 감지하고 해결합니다.

```
class ConflictResolver:
    """전략 간 충돌 감지 및 해결"""

    def __init__(self):
        self.conflict_history = []

    def detect_conflicts(
        self, pending_orders: List[dict]
    ) -> List[dict]:
        """대기 중인 주문에서 충돌 감지"""
        conflicts = []
        symbol_orders = {}

        # 자산별로 주문 그룹화
        for order in pending_orders:
            symbol = order['symbol']
            if symbol not in symbol_orders:
                symbol_orders[symbol] = []
            symbol_orders[symbol].append(order)

        # 같은 자산에 대한 상반된 주문 탐지
```

```
        for symbol, orders in symbol_orders.items():
            buy_orders = [o for o in orders if o['side'] == 'buy']
            sell_orders = [o for o in orders if o['side'] == 'sell']

            if buy_orders and sell_orders:
                conflicts.append({
                    'symbol': symbol,
                    'buy_strategies': [o['strategy'] for o in buy_orders],
                    'sell_strategies': [o['strategy'] for o in sell_orders],
                    'buy_total': sum(o['quantity'] for o in buy_orders),
                    'sell_total': sum(o['quantity'] for o in sell_orders),
                })

        return conflicts

    def resolve_conflict(
        self, conflict: dict,
        strategy_priorities: Dict[str, int],
        strategy_performances: Dict[str, StrategyPerformance]
    ) -> List[dict]:
        """충돌 해결: 우선순위와 성과 기반"""

        # 방법 1: 네팅 (순포지션만 실행)
        net_quantity = conflict['buy_total'] - conflict['sell_total']

        if abs(net_quantity) < 0.001:
            # 완전 상쇄 → 주문 취소
            return []

        # 방법 2: 우선순위 기반 (성과가 좋은 전략 우선)
        all_strategies = conflict['buy_strategies'] + conflict['sell_strategies']
        best_strategy = max(
            all_strategies,
            key=lambda s: (
                strategy_priorities.get(s, 0),
                strategy_performances.get(s, StrategyPerformance(s)).sharpe_ratio
            )
        )

        # 우수 전략의 주문만 실행
        is_buyer = best_strategy in conflict['buy_strategies']
        return [{
            'symbol': conflict['symbol'],
            'side': 'buy' if is_buyer else 'sell',
            'quantity': abs(net_quantity),
            'strategy': best_strategy,
            'resolution': 'priority_based'
        }]
```

4.3 VaR 기반 포트폴리오 리스크 측정

```
class PortfolioVaR:
    """Value at Risk 기반 포트폴리오 리스크 측정"""

    @staticmethod
    def parametric_var(
        weights: np.ndarray,
        returns: np.ndarray,
        confidence: float = 0.95,
        holding_period: int = 1
    ) -> float:
        """파라메트릭 VaR 계산"""
        from scipy.stats import norm

        portfolio_returns = returns @ weights
        mu = np.mean(portfolio_returns)
        sigma = np.std(portfolio_returns)

        z_score = norm.ppf(1 - confidence)
        var = -(mu + z_score * sigma) * np.sqrt(holding_period)
        return var

    @staticmethod
    def historical_var(
        weights: np.ndarray,
        returns: np.ndarray,
        confidence: float = 0.95
    ) -> float:
        """역사적 시뮬레이션 VaR"""
        portfolio_returns = returns @ weights
        var = -np.percentile(portfolio_returns, (1 - confidence) * 100)
        return var

    @staticmethod
    def component_var(
        weights: np.ndarray,
        cov_matrix: np.ndarray,
        confidence: float = 0.95
    ) -> np.ndarray:
        """컴포넌트 VaR: 각 전략의 VaR 기여도"""
        from scipy.stats import norm

        portfolio_var = np.sqrt(weights @ cov_matrix @ weights)
        z_score = norm.ppf(confidence)

        marginal_var = (cov_matrix @ weights) / portfolio_var * z_score
        component_var = weights * marginal_var
        return component_var
```

제5장: 실시간 모니터링과 대시보드

5.1 모니터링 메트릭 수집

멀티 전략 시스템의 건전성을 실시간으로 파악하기 위한 핵심 메트릭입니다.

```
import time
import json
from typing import Dict, Any
from datetime import datetime

class MetricsCollector:
    """전략별 실시간 메트릭 수집기"""

    def __init__(self):
        self.metrics_store: Dict[str, list] = {}
        self.alerts: list = []

    def collect_strategy_metrics(
        self, strategy: BaseStrategy
    ) -> Dict[str, Any]:
        """전략별 핵심 메트릭 수집"""
        metrics = {
            'timestamp': datetime.now().isoformat(),
            'strategy': strategy.name,
            'state': strategy.state.value,

            # 수익성 메트릭
            'pnl': {
                'unrealized': self._calc_unrealized_pnl(strategy),
                'realized': strategy.current_pnl,
                'total': strategy.current_pnl + self._calc_unrealized_pnl(strategy),
            },

            # 포지션 메트릭
            'positions': {
                'count': len(strategy.positions),
                'long_exposure': self._calc_directional_exposure(strategy, 'long'),
                'short_exposure': self._calc_directional_exposure(strategy, 'short'),
                'net_exposure': strategy.get_exposure(),
            },

            # 성과 메트릭
            'performance': {
                'sharpe_ratio': self._calc_rolling_sharpe(strategy),
                'win_rate': self._calc_win_rate(strategy),
                'profit_factor': self._calc_profit_factor(strategy),
                'max_drawdown': self._calc_max_drawdown(strategy),
                'current_drawdown': self._calc_current_drawdown(strategy),
            },
```

```
            # 실행 메트릭
            'execution': {
                'orders_today': self._count_orders_today(strategy),
                'fill_rate': self._calc_fill_rate(strategy),
                'avg_slippage': self._calc_avg_slippage(strategy),
                'latency_ms': self._calc_avg_latency(strategy),
            },

            # 시스템 메트릭
            'system': {
                'memory_mb': self._get_memory_usage(strategy),
                'cpu_percent': self._get_cpu_usage(strategy),
                'last_heartbeat': strategy.last_signal_time.isoformat()
                    if strategy.last_signal_time else None,
            }
        }

        # 저장
        if strategy.name not in self.metrics_store:
            self.metrics_store[strategy.name] = []
        self.metrics_store[strategy.name].append(metrics)

        # 알림 조건 검사
        self._check_alert_conditions(metrics)

        return metrics

    def _check_alert_conditions(self, metrics: dict):
        """알림 조건 검사"""
        strategy = metrics['strategy']

        # 드로다운 알림
        dd = metrics['performance']['current_drawdown']
        if dd > 0.10:
            self.alerts.append({
                'level': 'critical',
                'strategy': strategy,
                'message': f'드로다운 경고: {dd:.2%}',
                'timestamp': metrics['timestamp']
            })

        # 체결률 알림
        fill_rate = metrics['execution']['fill_rate']
        if fill_rate < 0.80:
            self.alerts.append({
                'level': 'warning',
                'strategy': strategy,
                'message': f'체결률 저하: {fill_rate:.2%}',
                'timestamp': metrics['timestamp']
            })

        # 하트비트 알림 (5분 이상 무신호)
```

```
            last_hb = metrics['system']['last_heartbeat']
            if last_hb:
                last_time = datetime.fromisoformat(last_hb)
                if (datetime.now() - last_time).seconds > 300:
                    self.alerts.append({
                        'level': 'critical',
                        'strategy': strategy,
                        'message': '전략 무응답 (5분+)',
                        'timestamp': metrics['timestamp']
                    })

        def _calc_unrealized_pnl(self, strategy):
            return sum(
                pos.get('unrealized_pnl', 0)
                for pos in strategy.positions.values()
            )

        def _calc_directional_exposure(self, strategy, direction):
            total = 0
            for pos in strategy.positions.values():
                value = abs(pos['quantity'] * pos.get('current_price', 0))
                if direction == 'long' and pos['quantity'] > 0:
                    total += value
                elif direction == 'short' and pos['quantity'] < 0:
                    total += value
            return total

        def _calc_rolling_sharpe(self, strategy, window=30):
            returns = strategy.__dict__.get('_daily_returns', [])
            if len(returns) < window:
                return 0
            r = np.array(returns[-window:])
            if np.std(r) == 0:
                return 0
            return float(np.mean(r) / np.std(r) * np.sqrt(252))

        def _calc_win_rate(self, strategy):
            trades = strategy.__dict__.get('_closed_trades', [])
            if not trades:
                return 0
            wins = sum(1 for t in trades if t.get('pnl', 0) > 0)
            return wins / len(trades)

        def _calc_profit_factor(self, strategy):
            trades = strategy.__dict__.get('_closed_trades', [])
            gross_profit = sum(t['pnl'] for t in trades if t.get('pnl', 0) > 0)
            gross_loss = abs(sum(t['pnl'] for t in trades if t.get('pnl', 0) < 0))
            if gross_loss == 0:
                return float('inf') if gross_profit > 0 else 0
            return gross_profit / gross_loss

        def _calc_max_drawdown(self, strategy):
            returns = strategy.__dict__.get('_daily_returns', [])
```

```
        if not returns:
            return 0
        cumulative = np.cumsum(returns)
        running_max = np.maximum.accumulate(cumulative)
        drawdowns = running_max - cumulative
        return float(np.max(drawdowns)) if len(drawdowns) > 0 else 0

    def _calc_current_drawdown(self, strategy):
        returns = strategy.__dict__.get('_daily_returns', [])
        if not returns:
            return 0
        cumulative = np.cumsum(returns)
        peak = np.max(cumulative)
        return float(peak - cumulative[-1])

    def _count_orders_today(self, strategy):
        return strategy.__dict__.get('_orders_today', 0)

    def _calc_fill_rate(self, strategy):
        return strategy.__dict__.get('_fill_rate', 1.0)

    def _calc_avg_slippage(self, strategy):
        return strategy.__dict__.get('_avg_slippage', 0.0)

    def _calc_avg_latency(self, strategy):
        return strategy.__dict__.get('_avg_latency_ms', 0.0)

    def _get_memory_usage(self, strategy):
        return 0

    def _get_cpu_usage(self, strategy):
        return 0
```

5.2 Prometheus 메트릭 내보내기

```
from prometheus_client import Gauge, Counter, Histogram, start_http_server

class PrometheusExporter:
    """Prometheus 형식으로 메트릭 내보내기"""

    def __init__(self, port: int = 9090):
        # 게이지 메트릭
        self.pnl_gauge = Gauge(
            'strategy_pnl_total',
            'Total PnL by strategy',
            ['strategy']
        )
        self.exposure_gauge = Gauge(
            'strategy_exposure_ratio',
            'Exposure ratio by strategy',
            ['strategy', 'direction']
```

```
        )
        self.drawdown_gauge = Gauge(
            'strategy_drawdown',
            'Current drawdown by strategy',
            ['strategy']
        )
        self.allocation_gauge = Gauge(
            'strategy_allocation_ratio',
            'Capital allocation ratio',
            ['strategy']
        )

        # 카운터 메트릭
        self.orders_counter = Counter(
            'strategy_orders_total',
            'Total orders by strategy',
            ['strategy', 'side', 'status']
        )
        self.risk_events_counter = Counter(
            'risk_events_total',
            'Risk events count',
            ['event_type', 'strategy']
        )

        # 히스토그램 메트릭
        self.latency_histogram = Histogram(
            'order_latency_seconds',
            'Order execution latency',
            ['strategy'],
            buckets=[0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0]
        )

        start_http_server(port)

    def update_metrics(self, metrics: dict):
        """메트릭 업데이트"""
        strategy = metrics['strategy']

        self.pnl_gauge.labels(strategy=strategy).set(
            metrics['pnl']['total']
        )
        self.exposure_gauge.labels(strategy=strategy, direction='long').set(
            metrics['positions']['long_exposure']
        )
        self.exposure_gauge.labels(strategy=strategy, direction='short').set(
            metrics['positions']['short_exposure']
        )
        self.drawdown_gauge.labels(strategy=strategy).set(
            metrics['performance']['current_drawdown']
        )
```

5.3 알림 시스템 구축

```
import aiohttp
import asyncio

class AlertManager:
    """멀티 채널 알림 관리자"""

    def __init__(self):
        self.channels = {}
        self.alert_rules = []
        self.cooldown_map = {}  # 중복 알림 방지
        self.cooldown_seconds = 300  # 5분 쿨다운

    def add_discord_webhook(self, name: str, webhook_url: str):
        self.channels[name] = {
            'type': 'discord',
            'url': webhook_url
        }

    def add_telegram_bot(self, name: str, bot_token: str, chat_id: str):
        self.channels[name] = {
            'type': 'telegram',
            'bot_token': bot_token,
            'chat_id': chat_id
        }

    async def send_alert(self, alert: dict):
        """알림 발송 (쿨다운 적용)"""
        alert_key = f"{alert['strategy']}:{alert['message']}"
        now = time.time()

        if alert_key in self.cooldown_map:
            if now - self.cooldown_map[alert_key] < self.cooldown_seconds:
                return  # 쿨다운 중

        self.cooldown_map[alert_key] = now

        level = alert.get('level', 'info')
        emoji = {'info': 'ℹ', 'warning': '⚠', 'critical': '🚨'}.get(level, '📢')

        message = (
            f"{emoji} **{alert['strategy']}** - {level.upper()}\n"
            f"{alert['message']}\n"
            f"시간: {alert['timestamp']}"
        )

        for ch_name, channel in self.channels.items():
            try:
                if channel['type'] == 'discord':
                    await self._send_discord(channel['url'], message)
                elif channel['type'] == 'telegram':
```

```
                    await self._send_telegram(channel, message)
            except Exception as e:
                print(f"알림 발송 실패 [{ch_name}]: {e}")

    async def _send_discord(self, url: str, message: str):
        async with aiohttp.ClientSession() as session:
            await session.post(url, json={'content': message})

    async def _send_telegram(self, channel: dict, message: str):
        url = f"https://api.telegram.org/bot{channel['bot_token']}/sendMessage"
        async with aiohttp.ClientSession() as session:
            await session.post(url, json={
                'chat_id': channel['chat_id'],
                'text': message,
                'parse_mode': 'Markdown'
            })
```

제6장: 전략 생명주기 관리

6.1 전략 생명주기 상태 머신

전략은 개발부터 퇴역까지 명확한 생명주기를 가집니다.

```
from enum import Enum
from datetime import datetime, timedelta
from typing import Optional, Dict, List

class LifecycleStage(Enum):
    DEVELOPMENT = "development"      # 개발 중
    PAPER_TRADING = "paper_trading"  # 모의 거래
    INCUBATION = "incubation"        # 소액 실거래
    PRODUCTION = "production"         # 정식 운영
    DEGRADED = "degraded"            # 성능 저하
    SUSPENDED = "suspended"          # 일시 중단
    RETIRED = "retired"              # 퇴역

class StrategyLifecycleManager:
    """전략 생명주기 관리자"""

    # 허용된 상태 전이 정의
    ALLOWED_TRANSITIONS = {
        LifecycleStage.DEVELOPMENT: [LifecycleStage.PAPER_TRADING],
        LifecycleStage.PAPER_TRADING: [
            LifecycleStage.INCUBATION,
            LifecycleStage.DEVELOPMENT  # 재개발
        ],
        LifecycleStage.INCUBATION: [
            LifecycleStage.PRODUCTION,
```

```
            LifecycleStage.PAPER_TRADING  # 롤백
        ],
        LifecycleStage.PRODUCTION: [
            LifecycleStage.DEGRADED,
            LifecycleStage.SUSPENDED
        ],
        LifecycleStage.DEGRADED: [
            LifecycleStage.PRODUCTION,  # 회복
            LifecycleStage.SUSPENDED,
            LifecycleStage.RETIRED
        ],
        LifecycleStage.SUSPENDED: [
            LifecycleStage.PRODUCTION,  # 재개
            LifecycleStage.RETIRED
        ],
        LifecycleStage.RETIRED: [],  # 최종 상태
    }

    def __init__(self):
        self.strategies: Dict[str, dict] = {}

    def register_strategy(self, name: str, metadata: dict = None):
        """새 전략 등록"""
        self.strategies[name] = {
            'stage': LifecycleStage.DEVELOPMENT,
            'created_at': datetime.now(),
            'stage_history': [{
                'stage': LifecycleStage.DEVELOPMENT.value,
                'entered_at': datetime.now().isoformat(),
                'reason': '신규 등록'
            }],
            'metadata': metadata or {},
            'promotion_criteria': self._default_criteria()
        }

    def transition(self, name: str, target: LifecycleStage, reason: str) -> bool:
        """상태 전이 실행"""
        if name not in self.strategies:
            raise ValueError(f"미등록 전략: {name}")

        current = self.strategies[name]['stage']

        if target not in self.ALLOWED_TRANSITIONS.get(current, []):
            print(f"[거부] {name}: {current.value} → {target.value} 전이 불가")
            return False

        self.strategies[name]['stage'] = target
        self.strategies[name]['stage_history'].append({
            'stage': target.value,
            'entered_at': datetime.now().isoformat(),
            'reason': reason,
            'previous': current.value
        })
```

```
        print(f"[전이] {name}: {current.value} → {target.value} ({reason})")
        return True

    def _default_criteria(self) -> dict:
        return {
            'paper_to_incubation': {
                'min_days': 30,
                'min_sharpe': 1.5,
                'min_trades': 100,
                'max_drawdown': 0.10
            },
            'incubation_to_production': {
                'min_days': 14,
                'min_sharpe': 1.0,
                'min_profit_factor': 1.5,
                'max_drawdown': 0.08
            },
            'production_to_degraded': {
                'rolling_sharpe_below': 0.5,
                'consecutive_loss_days': 5,
                'drawdown_exceeds': 0.15
            }
        }

    def auto_evaluate(
        self, name: str, performance: StrategyPerformance
    ) -> Optional[LifecycleStage]:
        """성과 기반 자동 생명주기 평가"""
        strategy = self.strategies[name]
        current = strategy['stage']
        criteria = strategy['promotion_criteria']

        if current == LifecycleStage.PAPER_TRADING:
            c = criteria['paper_to_incubation']
            stage_days = self._days_in_stage(name)
            if (stage_days >= c['min_days'] and
                performance.sharpe_ratio >= c['min_sharpe'] and
                performance.max_drawdown <= c['max_drawdown']):
                self.transition(name, LifecycleStage.INCUBATION, '모의거래 기준 충족')
                return LifecycleStage.INCUBATION

        elif current == LifecycleStage.INCUBATION:
            c = criteria['incubation_to_production']
            stage_days = self._days_in_stage(name)
            if (stage_days >= c['min_days'] and
                performance.sharpe_ratio >= c['min_sharpe'] and
                performance.max_drawdown <= c['max_drawdown']):
                self.transition(name, LifecycleStage.PRODUCTION, '인큐베이션 기준 충족')
                return LifecycleStage.PRODUCTION

        elif current == LifecycleStage.PRODUCTION:
            c = criteria['production_to_degraded']
```

```
            if (performance.sharpe_ratio < c['rolling_sharpe_below'] or
                performance.current_drawdown > c['drawdown_exceeds']):
                self.transition(name, LifecycleStage.DEGRADED, '성능 저하 감지')
                return LifecycleStage.DEGRADED

        return None

    def _days_in_stage(self, name: str) -> int:
        history = self.strategies[name]['stage_history']
        last_entry = datetime.fromisoformat(history[-1]['entered_at'])
        return (datetime.now() - last_entry).days

    def get_status_report(self) -> List[dict]:
        """전체 전략 생명주기 현황 보고서"""
        report = []
        for name, data in self.strategies.items():
            report.append({
                'strategy': name,
                'stage': data['stage'].value,
                'days_in_stage': self._days_in_stage(name),
                'total_age_days': (datetime.now() - data['created_at']).days,
                'transitions': len(data['stage_history']),
            })
        return report
```

6.2 A/B 테스트 프레임워크

새 전략이나 파라미터 변경을 안전하게 검증하기 위한 A/B 테스트 시스템입니다.

```
import uuid
from typing import Tuple

class ABTestManager:
    """전략 A/B 테스트 관리"""

    def __init__(self):
        self.tests: Dict[str, dict] = {}

    def create_test(
        self,
        name: str,
        control_strategy: BaseStrategy,
        treatment_strategy: BaseStrategy,
        traffic_split: float = 0.5,  # 치료군에 배분할 비율
        duration_days: int = 14,
        min_trades: int = 50
    ) -> str:
        """A/B 테스트 생성"""
        test_id = str(uuid.uuid4())[:8]
```

```
        self.tests[test_id] = {
            'name': name,
            'control': {
                'strategy': control_strategy,
                'capital_ratio': 1 - traffic_split,
                'trades': [],
                'pnl': 0.0
            },
            'treatment': {
                'strategy': treatment_strategy,
                'capital_ratio': traffic_split,
                'trades': [],
                'pnl': 0.0
            },
            'started_at': datetime.now(),
            'duration_days': duration_days,
            'min_trades': min_trades,
            'status': 'running'
        }

        return test_id

    def evaluate_test(self, test_id: str) -> dict:
        """A/B 테스트 결과 평가"""
        test = self.tests[test_id]
        control = test['control']
        treatment = test['treatment']

        # 충분한 데이터 확인
        if (len(control['trades']) < test['min_trades'] or
            len(treatment['trades']) < test['min_trades']):
            return {'status': 'insufficient_data'}

        # 통계적 유의성 검정
        control_returns = [t['pnl'] for t in control['trades']]
        treatment_returns = [t['pnl'] for t in treatment['trades']]

        from scipy.stats import ttest_ind, mannwhitneyu

        t_stat, p_value = ttest_ind(control_returns, treatment_returns)

        # 실전 메트릭 비교
        result = {
            'status': 'completed',
            'control': self._calc_test_metrics(control),
            'treatment': self._calc_test_metrics(treatment),
            'statistical_test': {
                't_statistic': round(t_stat, 4),
                'p_value': round(p_value, 4),
                'significant': p_value < 0.05
            },
            'recommendation': self._make_recommendation(
                control_returns, treatment_returns, p_value
```

```
            )
        }

        return result

    def _calc_test_metrics(self, group: dict) -> dict:
        returns = [t['pnl'] for t in group['trades']]
        if not returns:
            return {}
        return {
            'total_pnl': sum(returns),
            'avg_pnl': np.mean(returns),
            'sharpe': np.mean(returns) / np.std(returns) * np.sqrt(252)
                if np.std(returns) > 0 else 0,
            'win_rate': sum(1 for r in returns if r > 0) / len(returns),
            'trade_count': len(returns),
            'max_drawdown': self._calc_drawdown(returns)
        }

    def _calc_drawdown(self, returns: list) -> float:
        cumulative = np.cumsum(returns)
        running_max = np.maximum.accumulate(cumulative)
        drawdown = running_max - cumulative
        return float(np.max(drawdown)) if len(drawdown) > 0 else 0

    def _make_recommendation(
        self, control: list, treatment: list, p_value: float
    ) -> str:
        if p_value >= 0.05:
            return "통계적으로 유의미한 차이 없음. 테스트 연장 또는 현행 유지 권장."

        treat_mean = np.mean(treatment)
        ctrl_mean = np.mean(control)

        if treat_mean > ctrl_mean:
            improvement = (treat_mean - ctrl_mean) / abs(ctrl_mean) * 100
            return f"치료군 우수 (+{improvement:.1f}%). 새 전략 채택 권장."
        else:
            return "대조군 우수. 새 전략 채택하지 않을 것을 권장."
```

6.3 전략 퇴역 프로세스

```
class RetirementManager:
    """전략 안전 퇴역 관리"""

    async def retire_strategy(
        self,
        strategy: BaseStrategy,
        orchestrator: CentralOrchestrator,
        reason: str
    ) -> dict:
```

```
        """전략 퇴역 실행 (안전한 포지션 정리 포함)"""

        retirement_log = {
            'strategy': strategy.name,
            'reason': reason,
            'started_at': datetime.now().isoformat(),
            'positions_closed': [],
            'final_pnl': 0
        }

        # 1단계: 새 주문 중단
        strategy.state = StrategyState.PAUSED
        print(f"[퇴역 1단계] {strategy.name} 새 주문 중단")

        # 2단계: 기존 포지션 점진적 청산
        for symbol, position in list(strategy.positions.items()):
            try:
                close_order = {
                    'symbol': symbol,
                    'side': 'sell' if position['quantity'] > 0 else 'buy',
                    'quantity': abs(position['quantity']),
                    'type': 'limit',  # 시장 충격 최소화
                    'price': position['current_price'],
                    'strategy': strategy.name
                }

                await orchestrator.order_router.submit(close_order)
                retirement_log['positions_closed'].append({
                    'symbol': symbol,
                    'quantity': position['quantity'],
                    'close_price': position['current_price']
                })

                # 주문 간 간격 (시장 충격 완화)
                await asyncio.sleep(1)

            except Exception as e:
                print(f"[퇴역 오류] {symbol} 포지션 청산 실패: {e}")

        # 3단계: 자본 회수 및 재배분
        recovered_capital = strategy.allocated_capital
        strategy.allocated_capital = 0

        # 4단계: 최종 기록
        retirement_log['completed_at'] = datetime.now().isoformat()
        retirement_log['final_pnl'] = strategy.current_pnl
        retirement_log['recovered_capital'] = recovered_capital

        strategy.state = StrategyState.IDLE

        print(f"[퇴역 완료] {strategy.name} - 회수 자본: {recovered_capital:,.0f}")
        return retirement_log
```

제7장: 실전 멀티 전략 시스템 구현

7.1 전체 시스템 통합

지금까지 다룬 모든 구성 요소를 통합하여 실전 멀티 전략 시스템을 구현합니다.

```
import asyncio
import signal
import logging
from pathlib import Path
import yaml

# 로깅 설정
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(name)s] %(levelname)s: %(message)s'
)
logger = logging.getLogger('MultiStrategySystem')

class MultiStrategySystem:
    """실전 멀티 전략 자동매매 시스템"""

    def __init__(self, config_path: str):
        self.config = self._load_config(config_path)
        self.total_capital = self.config['system']['total_capital']

        # 핵심 컴포넌트 초기화
        self.orchestrator = CentralOrchestrator(self.total_capital)
        self.allocator = DynamicAllocator(
            self.total_capital,
            rebalance_interval=timedelta(
                hours=self.config['system']['rebalance_hours']
            )
        )
        self.risk_manager = PortfolioRiskManager(
            self.total_capital,
            RiskLimits(**self.config.get('risk_limits', {}))
        )
        self.lifecycle = StrategyLifecycleManager()
        self.metrics = MetricsCollector()
        self.alerts = AlertManager()
        self.drawdown_guard = DrawdownGuard()

        self._shutdown_event = asyncio.Event()

    def _load_config(self, path: str) -> dict:
        with open(path, 'r') as f:
            return yaml.safe_load(f)

    async def initialize(self):
```

```
        """시스템 초기화"""
        logger.info("=== 멀티 전략 시스템 초기화 ===")

        # 전략 로딩
        for strategy_config in self.config['strategies']:
            strategy = self._create_strategy(strategy_config)
            capital_ratio = strategy_config['capital_ratio']

            self.orchestrator.add_strategy(strategy, capital_ratio)
            self.lifecycle.register_strategy(strategy.name)

            logger.info(
                f"전략 로딩: {strategy.name} "
                f"(자본 비율: {capital_ratio:.1%})"
            )

        # 알림 채널 설정
        if 'alerts' in self.config:
            for alert_config in self.config['alerts']:
                if alert_config['type'] == 'discord':
                    self.alerts.add_discord_webhook(
                        alert_config['name'],
                        alert_config['webhook_url']
                    )

        logger.info(f"총 {len(self.orchestrator.strategies)}개 전략 로딩 완료")

    def _create_strategy(self, config: dict) -> BaseStrategy:
        """설정에서 전략 인스턴스 생성"""
        strategy_map = {
            'momentum': MomentumStrategy,
            'mean_reversion': MeanReversionStrategy,
            'arbitrage': ArbitrageStrategy,
            'market_making': MarketMakingStrategy,
            'trend_following': TrendFollowingStrategy,
        }

        cls = strategy_map.get(config['type'])
        if cls is None:
            raise ValueError(f"미지원 전략 유형: {config['type']}")

        return cls(
            name=config['name'],
            **config.get('params', {})
        )

    async def run(self):
        """메인 실행 루프"""
        await self.initialize()

        # 시그널 핸들러 설정
        loop = asyncio.get_event_loop()
        for sig in (signal.SIGTERM, signal.SIGINT):
```

```
            loop.add_signal_handler(
                sig,
                lambda: asyncio.create_task(self.shutdown())
            )

        # 모든 전략 활성화
        for strategy in self.orchestrator.strategies:
            strategy.state = StrategyState.RUNNING

        logger.info("=== 시스템 시작 ===")

        # 병렬 태스크 실행
        await asyncio.gather(
            self._trading_loop(),
            self._monitoring_loop(),
            self._rebalance_loop(),
            self._lifecycle_loop(),
            self._wait_shutdown()
        )

    async def _trading_loop(self):
        """매매 실행 루프"""
        while not self._shutdown_event.is_set():
            try:
                market_data = await self._fetch_market_data()

                for strategy in self.orchestrator.strategies:
                    if strategy.state != StrategyState.RUNNING:
                        continue

                    signal_result = await strategy.generate_signal(market_data)
                    if signal_result is None:
                        continue

                    # 드로다운 가드 적용
                    perf = self._get_performance(strategy)
                    scaling = self.drawdown_guard.get_scaling_factor(
                        perf.current_drawdown
                    )

                    if scaling == 0:
                        strategy.state = StrategyState.PAUSED
                        logger.warning(f"{strategy.name} 드로다운 정지")
                        continue

                    # 스케일링 적용
                    signal_result['quantity'] *= scaling

                    # 리스크 검증
                    portfolio_state = self.orchestrator._get_portfolio_state()
                    if self.risk_manager.validate_order(
                        strategy, signal_result, portfolio_state
                    ):
```

```
                        await self.orchestrator.order_router.submit(signal_result)

            except Exception as e:
                logger.error(f"매매 루프 오류: {e}", exc_info=True)

            await asyncio.sleep(0.1)

    async def _monitoring_loop(self):
        """모니터링 루프 (10초 간격)"""
        while not self._shutdown_event.is_set():
            try:
                for strategy in self.orchestrator.strategies:
                    metrics = self.metrics.collect_strategy_metrics(strategy)

                    # 알림 발송
                    for alert in self.metrics.alerts:
                        await self.alerts.send_alert(alert)
                    self.metrics.alerts.clear()

            except Exception as e:
                logger.error(f"모니터링 오류: {e}")

            await asyncio.sleep(10)

    async def _rebalance_loop(self):
        """자본 재배분 루프"""
        while not self._shutdown_event.is_set():
            try:
                if self.allocator.should_rebalance():
                    # 성과 데이터 업데이트
                    for strategy in self.orchestrator.strategies:
                        perf = self._get_performance(strategy)
                        self.allocator.strategies[strategy.name] = perf

                    changes = self.allocator.execute_rebalance()

                    for name, change in changes.items():
                        if abs(change['change']) > 0.01:
                            logger.info(
                                f"[리밸런싱] {name}: "
                                f"{change['previous']:.1%} → {change['new']:.1%} "
                                f"(자본: {change['capital']:,.0f}원)"
                            )

            except Exception as e:
                logger.error(f"리밸런싱 오류: {e}")

            await asyncio.sleep(60)

    async def _lifecycle_loop(self):
        """생명주기 관리 루프 (1시간 간격)"""
        while not self._shutdown_event.is_set():
            try:
```

```
                for strategy in self.orchestrator.strategies:
                    perf = self._get_performance(strategy)
                    new_stage = self.lifecycle.auto_evaluate(
                        strategy.name, perf
                    )
                    if new_stage:
                        await self.alerts.send_alert({
                            'level': 'info',
                            'strategy': strategy.name,
                            'message': f'생명주기 전이: → {new_stage.value}',
                            'timestamp': datetime.now().isoformat()
                        })

            except Exception as e:
                logger.error(f"생명주기 평가 오류: {e}")

            await asyncio.sleep(3600)

    def _get_performance(self, strategy: BaseStrategy) -> StrategyPerformance:
        """전략 성과 객체 생성"""
        perf = StrategyPerformance(name=strategy.name)
        perf.current_allocation = strategy.allocated_capital / self.total_capital
        perf.current_pnl = strategy.current_pnl
        return perf

    async def _fetch_market_data(self) -> dict:
        return {}

    async def _wait_shutdown(self):
        await self._shutdown_event.wait()

    async def shutdown(self):
        """안전한 시스템 종료"""
        logger.info("=== 시스템 종료 시작 ===")

        # 모든 전략 중단
        for strategy in self.orchestrator.strategies:
            strategy.state = StrategyState.PAUSED

        # 잔여 주문 취소
        logger.info("미체결 주문 취소 중...")

        # 최종 메트릭 저장
        final_report = {
            'shutdown_time': datetime.now().isoformat(),
            'strategies': []
        }

        for strategy in self.orchestrator.strategies:
            metrics = self.metrics.collect_strategy_metrics(strategy)
            final_report['strategies'].append(metrics)

        # 보고서 저장
```

```
        report_path = Path('reports') / f"shutdown_{datetime.now():%Y%m%d_%H%M%S}.json"
        report_path.parent.mkdir(exist_ok=True)
        with open(report_path, 'w') as f:
            json.dump(final_report, f, indent=2, ensure_ascii=False)

        logger.info(f"종료 보고서 저장: {report_path}")
        self._shutdown_event.set()
```

7.2 설정 파일 구조

```yaml
# config/multi_strategy.yaml

system:
  total_capital: 100000000  # 1억원
  rebalance_hours: 24
  base_currency: KRW

strategies:
  - name: btc_momentum
    type: momentum
    capital_ratio: 0.25
    params:
      symbols: [BTC/KRW]
      lookback_period: 20
      entry_threshold: 0.02
      stop_loss: 0.03
      take_profit: 0.06

  - name: eth_mean_reversion
    type: mean_reversion
    capital_ratio: 0.20
    params:
      symbols: [ETH/KRW]
      lookback_period: 50
      z_score_entry: 2.0
      z_score_exit: 0.5

  - name: cross_exchange_arb
    type: arbitrage
    capital_ratio: 0.20
    params:
      exchanges: [upbit, binance]
      symbols: [BTC/KRW, ETH/KRW]
      min_spread: 0.003

  - name: btc_market_maker
    type: market_making
    capital_ratio: 0.20
    params:
      symbol: BTC/KRW
      spread: 0.002
```

```
      order_levels: 5
      order_size_decay: 0.8

  - name: news_event
    type: trend_following
    capital_ratio: 0.15
    params:
      symbols: [BTC/KRW, ETH/KRW, XRP/KRW]
      trend_period: 50
      atr_multiplier: 2.0

risk_limits:
  max_total_exposure: 0.80
  max_single_strategy_exposure: 0.30
  max_daily_loss: 0.03
  max_weekly_loss: 0.07
  emergency_stop_loss: 0.10

alerts:
  - name: discord_trading
    type: discord
    webhook_url: "https://discord.com/api/webhooks/..."
  - name: telegram_alerts
    type: telegram
    bot_token: "your-bot-token"
    chat_id: "your-chat-id"
```

7.3 시스템 운영 체크리스트

멀티 전략 시스템을 안정적으로 운영하기 위한 체크리스트입니다.

일일 점검 항목: - 전 전략 PnL 확인 및 이상치 검사 - 드로다운 현황 확인 - 체결률 및 슬리피지 모니터링 - 미체결 주문 존재 여부 확인 - 시스템 리소스(CPU, 메모리, 디스크) 확인 - 로그 파일 오류 검사

주간 점검 항목: - 전략 간 상관관계 변화 확인 - 자본 배분 비율 적정성 검토 - 리스크 한도 위반 이력 검토 - 전략 생명주기 상태 평가 - 시장 레짐 변화 분석

월간 점검 항목: - 전략 성과 종합 보고서 작성 - 신규 전략 후보 탐색 및 백테스트 - 기존 전략 파라미터 최적화 검토 - 인프라 비용 대비 수익성 평가 - 리스크 모델 검증 (VaR 백테스트)

제8장: 고급 기법과 최적화

8.1 레짐 기반 동적 전략 선택

시장 레짐에 따라 활성화할 전략을 동적으로 전환하는 고급 기법입니다.

```
from sklearn.mixture import GaussianMixture
import numpy as np

class RegimeDetector:
    """시장 레짐 감지기"""

    def __init__(self, n_regimes: int = 3):
        self.n_regimes = n_regimes
        self.model = GaussianMixture(
            n_components=n_regimes,
            covariance_type='full',
            random_state=42
        )
        self.regime_labels = {
            0: 'low_volatility',   # 저변동성 (횡보)
            1: 'trending',         # 추세장
            2: 'high_volatility'   # 고변동성 (급등락)
        }
        self.regime_strategy_map = {
            'low_volatility': ['mean_reversion', 'market_making', 'arbitrage'],
            'trending': ['momentum', 'trend_following'],
            'high_volatility': ['arbitrage', 'market_making']
        }

    def fit(self, features: np.ndarray):
        """과거 데이터로 레짐 모델 학습"""
        self.model.fit(features)
        # 레짐 라벨 재매핑 (변동성 기준)
        means = self.model.means_
        vol_order = np.argsort(means[:, 1])  # 변동성 열 기준 정렬
        self.label_map = {vol_order[i]: i for i in range(self.n_regimes)}

    def predict_regime(self, features: np.ndarray) -> dict:
        """현재 시장 레짐 예측"""
        probabilities = self.model.predict_proba(features.reshape(1, -1))[0]
        predicted = self.model.predict(features.reshape(1, -1))[0]
        mapped_regime = self.label_map.get(predicted, predicted)

        return {
            'regime': self.regime_labels.get(mapped_regime, f'regime_{mapped_regime}'),
            'probabilities': {
                self.regime_labels[self.label_map[i]]: round(float(p), 4)
                for i, p in enumerate(probabilities)
            },
```

```
            'confidence': round(float(max(probabilities)), 4),
            'recommended_strategies': self.regime_strategy_map.get(
                self.regime_labels.get(mapped_regime, ''),
                []
            )
        }

    def compute_features(self, prices: np.ndarray, window: int = 20) -> np.ndarray:
        """시장 레짐 감지용 특성 계산"""
        returns = np.diff(np.log(prices))

        features = np.array([
            np.mean(returns[-window:]),                    # 평균 수익률
            np.std(returns[-window:]),                     # 변동성
            np.mean(returns[-window:]) / (np.std(returns[-window:]) + 1e-8),  # 샤프
            (prices[-1] - prices[-window]) / prices[-window],  # 모멘텀
            np.max(prices[-window:]) / np.min(prices[-window:]) - 1,  # 범위
        ])

        return features


class RegimeAdaptiveAllocator:
    """레짐 적응형 자본 배분기"""

    def __init__(
        self,
        regime_detector: RegimeDetector,
        base_allocator: DynamicAllocator
    ):
        self.regime_detector = regime_detector
        self.base_allocator = base_allocator
        self.regime_weights = {
            'low_volatility': {
                'mean_reversion': 1.5,
                'market_making': 1.3,
                'arbitrage': 1.2,
                'momentum': 0.5,
                'trend_following': 0.3
            },
            'trending': {
                'momentum': 1.5,
                'trend_following': 1.5,
                'mean_reversion': 0.3,
                'market_making': 0.7,
                'arbitrage': 1.0
            },
            'high_volatility': {
                'arbitrage': 1.3,
                'market_making': 0.5,
                'momentum': 0.8,
                'mean_reversion': 0.5,
                'trend_following': 0.7
```

```
            }
        }

    def compute_regime_adjusted_allocations(
        self, market_features: np.ndarray
    ) -> Dict[str, float]:
        """레짐 감지 결과를 반영한 배분 비율 계산"""
        regime_info = self.regime_detector.predict_regime(market_features)
        regime = regime_info['regime']
        confidence = regime_info['confidence']

        base_allocations = self.base_allocator.compute_target_allocations()
        weights = self.regime_weights.get(regime, {})

        adjusted = {}
        for strategy_name, base_alloc in base_allocations.items():
            strategy_type = self._get_strategy_type(strategy_name)
            regime_weight = weights.get(strategy_type, 1.0)

            # 신뢰도에 따라 조정 강도 결정
            adjusted_weight = 1.0 + (regime_weight - 1.0) * confidence
            adjusted[strategy_name] = base_alloc * adjusted_weight

        # 정규화
        total = sum(adjusted.values())
        return {k: v / total for k, v in adjusted.items()}

    def _get_strategy_type(self, name: str) -> str:
        type_map = {
            'btc_momentum': 'momentum',
            'eth_mean_reversion': 'mean_reversion',
            'cross_exchange_arb': 'arbitrage',
            'btc_market_maker': 'market_making',
            'news_event': 'trend_following',
        }
        return type_map.get(name, 'unknown')
```

8.2 전략 앙상블 기법

여러 전략의 신호를 결합하여 더 안정적인 매매 결정을 내리는 앙상블 기법입니다.

```
class StrategyEnsemble:
    """전략 앙상블: 여러 전략의 신호를 결합"""

    def __init__(self, voting_threshold: float = 0.6):
        self.strategies: List[BaseStrategy] = []
        self.strategy_weights: Dict[str, float] = {}
        self.voting_threshold = voting_threshold

    def add_strategy(self, strategy: BaseStrategy, weight: float = 1.0):
```

```
        self.strategies.append(strategy)
        self.strategy_weights[strategy.name] = weight

    async def generate_ensemble_signal(
        self, market_data: dict
    ) -> Optional[dict]:
        """앙상블 신호 생성"""
        signals = {}
        total_weight = 0

        for strategy in self.strategies:
            if strategy.state != StrategyState.RUNNING:
                continue

            signal = await strategy.generate_signal(market_data)
            if signal:
                weight = self.strategy_weights[strategy.name]
                signals[strategy.name] = {
                    'signal': signal,
                    'weight': weight
                }
                total_weight += weight

        if not signals:
            return None

        # 다수결 투표
        buy_weight = sum(
            s['weight'] for s in signals.values()
            if s['signal']['side'] == 'buy'
        )
        sell_weight = sum(
            s['weight'] for s in signals.values()
            if s['signal']['side'] == 'sell'
        )

        buy_ratio = buy_weight / total_weight
        sell_ratio = sell_weight / total_weight

        if buy_ratio >= self.voting_threshold:
            # 매수 합의: 가중 평균 수량 계산
            avg_qty = np.average(
                [s['signal']['quantity'] for s in signals.values()
                 if s['signal']['side'] == 'buy'],
                weights=[s['weight'] for s in signals.values()
                        if s['signal']['side'] == 'buy']
            )
            return {
                'side': 'buy',
                'quantity': avg_qty,
                'confidence': buy_ratio,
                'contributing_strategies': [
                    name for name, s in signals.items()
```

```
                    if s['signal']['side'] == 'buy'
                ]
            }

        elif sell_ratio >= self.voting_threshold:
            avg_qty = np.average(
                [s['signal']['quantity'] for s in signals.values()
                 if s['signal']['side'] == 'sell'],
                weights=[s['weight'] for s in signals.values()
                        if s['signal']['side'] == 'sell']
            )
            return {
                'side': 'sell',
                'quantity': avg_qty,
                'confidence': sell_ratio,
                'contributing_strategies': [
                    name for name, s in signals.items()
                    if s['signal']['side'] == 'sell'
                ]
            }

        return None  # 합의 부족
```

8.3 성능 최적화 팁

멀티 전략 시스템의 성능을 극대화하기 위한 실전 팁입니다.

1. 데이터 공유 최적화: 여러 전략이 같은 시장 데이터를 사용할 때, 중복 API 호출을 방지합니다.

```
class SharedDataFeed:
    """전략 간 시장 데이터 공유"""

    def __init__(self):
        self._cache = {}
        self._subscribers = defaultdict(list)
        self._lock = asyncio.Lock()

    async def subscribe(self, symbol: str, callback):
        self._subscribers[symbol].append(callback)

    async def update(self, symbol: str, data: dict):
        async with self._lock:
            self._cache[symbol] = data

        # 구독자에게 데이터 전파
        for callback in self._subscribers[symbol]:
            asyncio.create_task(callback(data))
```

```
    def get_latest(self, symbol: str) -> Optional[dict]:
        return self._cache.get(symbol)
```

2. 주문 배치 처리: 여러 전략의 주문을 모아서 한 번에 처리하여 API 호출 횟수를 줄입니다.

```
class OrderBatcher:
    """주문 배치 처리기"""

    def __init__(self, batch_interval_ms: int = 100):
        self.batch_interval = batch_interval_ms / 1000
        self.pending_orders = []
        self._lock = asyncio.Lock()

    async def add_order(self, order: dict):
        async with self._lock:
            self.pending_orders.append(order)

    async def process_batch(self):
        """배치 처리 루프"""
        while True:
            await asyncio.sleep(self.batch_interval)

            async with self._lock:
                if not self.pending_orders:
                    continue

                batch = self.pending_orders.copy()
                self.pending_orders.clear()

            # 같은 자산 주문 네팅
            netted = self._net_orders(batch)

            for order in netted:
                try:
                    await self._execute_order(order)
                except Exception as e:
                    logger.error(f"주문 실행 실패: {e}")

    def _net_orders(self, orders: List[dict]) -> List[dict]:
        """같은 자산에 대한 주문을 네팅"""
        symbol_orders = defaultdict(list)
        for order in orders:
            symbol_orders[order['symbol']].append(order)

        netted = []
        for symbol, sym_orders in symbol_orders.items():
            net_qty = sum(
                o['quantity'] if o['side'] == 'buy' else -o['quantity']
                for o in sym_orders
            )
```

```
            if abs(net_qty) > 0.001:
                netted.append({
                    'symbol': symbol,
                    'side': 'buy' if net_qty > 0 else 'sell',
                    'quantity': abs(net_qty),
                    'type': 'market'
                })

        return netted

    async def _execute_order(self, order: dict):
        # 실제 거래소 API 호출
        pass
```

3. 메모리 효율화: 시계열 데이터를 링 버퍼로 관리하여 메모리 사용량을 제어합니다.

```
class RingBuffer:
    """고정 크기 링 버퍼"""

    def __init__(self, capacity: int):
        self.capacity = capacity
        self.buffer = np.zeros(capacity)
        self.index = 0
        self.full = False

    def append(self, value: float):
        self.buffer[self.index] = value
        self.index = (self.index + 1) % self.capacity
        if self.index == 0:
            self.full = True

    def get_data(self) -> np.ndarray:
        if self.full:
            return np.concatenate([
                self.buffer[self.index:],
                self.buffer[:self.index]
            ])
        return self.buffer[:self.index]

    def __len__(self):
        return self.capacity if self.full else self.index
```

8.4 운영 자동화

시스템 운영을 자동화하여 인적 개입을 최소화합니다.

```
class OperationsAutomation:
    """운영 자동화"""
```

```
    def __init__(self, system: MultiStrategySystem):
        self.system = system

    async def daily_routine(self):
        """일일 자동 루틴"""
        # 1. 일일 성과 보고서 생성
        report = self._generate_daily_report()

        # 2. 전략 건전성 자동 점검
        health_issues = self._health_check()

        # 3. 로그 로테이션
        self._rotate_logs()

        # 4. 보고서 발송
        if health_issues:
            await self.system.alerts.send_alert({
                'level': 'warning',
                'strategy': 'SYSTEM',
                'message': f"일일 점검: {len(health_issues)}건 이슈 발견",
                'timestamp': datetime.now().isoformat()
            })

        return report

    def _generate_daily_report(self) -> dict:
        """일일 성과 보고서"""
        report = {
            'date': datetime.now().strftime('%Y-%m-%d'),
            'total_pnl': 0,
            'strategies': []
        }

        for strategy in self.system.orchestrator.strategies:
            perf = self.system._get_performance(strategy)
            strategy_report = {
                'name': strategy.name,
                'state': strategy.state.value,
                'pnl': strategy.current_pnl,
                'positions': len(strategy.positions),
                'allocation': perf.current_allocation
            }
            report['strategies'].append(strategy_report)
            report['total_pnl'] += strategy.current_pnl

        return report

    def _health_check(self) -> List[str]:
        """시스템 건전성 점검"""
        issues = []

        for strategy in self.system.orchestrator.strategies:
            if strategy.state == StrategyState.ERROR:
```

```
                issues.append(f"{strategy.name}: 에러 상태")

            if (strategy.last_signal_time and
                (datetime.now() - strategy.last_signal_time).hours > 24):
                issues.append(f"{strategy.name}: 24시간 이상 무신호")

        return issues

    def _rotate_logs(self):
        """오래된 로그 정리"""
        log_dir = Path('logs')
        if not log_dir.exists():
            return

        cutoff = datetime.now() - timedelta(days=30)
        for log_file in log_dir.glob('*.log'):
            if datetime.fromtimestamp(log_file.stat().st_mtime) < cutoff:
                log_file.unlink()
```

이 가이드에서 다룬 모든 기법을 단계적으로 적용하면, 개인 투자자도 기관급 멀티 전략 자동매매 시스템을 구축하고 안정적으로 운영할 수 있습니다. 핵심은 단순한 구조에서 시작하여 점진적으로 복잡성을 추가하는 것입니다. 독립 실행형 아키텍처로 2~3개 전략을 운영하며 경험을 쌓은 후, 중앙 집중형이나 이벤트 기반 아키텍처로 확장해 나가시기 바랍니다.