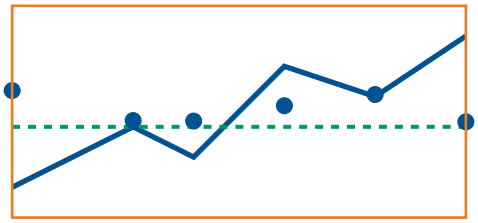
Algorithmic Stock Trading System

Comprehensive Development Plan



Market Analysis Engine

Document Type:	Technical Specification
Version:	1.0
Date:	November 30, 2025
Status:	Draft

This document synthesizes knowledge from foundational texts in value investing, technical analysis, trading psychology, quantitative methods, and machine learning for algorithmic trading.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Executive Summary

1.1 Document Purpose

This document presents a comprehensive development plan for building a professional-grade algorithmic stock trading system. The plan integrates theoretical foundations from established financial literature with modern software engineering practices to create a robust, scalable, and maintainable trading platform.

1.2 Knowledge Foundations

The system design draws from multiple domains of expertise, each contributing essential components to the overall architecture:

Table 1.1: Core Knowledge Domains and Source References

Domain	**Key Concepts**	**Primary Sources**
Fundamental Analysis	Value investing, company financials, long-term strategy	Graham
Technical Analysis	Chart patterns, indicators, price/volume analysis	Murphy, O'Neil, Pring
Trading Psychology	Discipline, risk tolerance, emotional control	Douglas, Schwager
Quantitative Methods	Backtesting, statistical modeling, systematic strategies	Chan (both books)
Machine Learning	Predictive models, feature engineering	Jansen, López de Prado
System Architecture	Direct market access, execution algorithms	Johnson, Aldridge

1.3 System Overview

The proposed algorithmic trading system consists of several interconnected components designed to work in concert:

1. **Data Layer**: Real-time and historical market data ingestion with comprehensive feature engineering

2. **Strategy Engine**: Multiple strategy implementations spanning fundamental, technical, and quantitative approaches
3. **Backtesting Framework**: Event-driven simulation with walk-forward optimization
4. **Risk Management**: Position sizing, portfolio constraints, and automated risk controls
5. **Machine Learning Pipeline**: Predictive modeling with proper cross-validation techniques
6. **Execution Engine**: Broker integration with smart order routing
7. **Monitoring Dashboard**: Real-time analytics and alerting

1.4 Success Criteria

The system must meet the following quantitative benchmarks to be considered production-ready:

- Sharpe Ratio > 1.0 on out-of-sample data
- Profit Factor > 1.5
- Maximum Drawdown $< 15\%$
- Daily Value at Risk (VaR) $< 2\%$
- Order Fill Rate $> 95\%$
- Execution Slippage < 10 basis points

Chapter 2

System Requirements Specification

2.1 Functional Requirements

2.1.1 Data Management

Table 2.1: Data Management Requirements

ID	Priority	Requirement Description
FR-D01	Critical	System shall ingest real-time market data with latency < 100ms
FR-D02	Critical	System shall store historical OHLCV data for all tracked symbols
FR-D03	High	System shall retrieve fundamental data (earnings, ratios, filings)
FR-D04	Medium	System shall process alternative data (news, sentiment, SEC filings)
FR-D05	High	System shall compute technical indicators in real-time
FR-D06	High	System shall handle data gaps and corporate actions appropriately

2.1.2 Strategy Management

Table 2.2: Strategy Management Requirements

ID	Priority	Requirement Description
FR-S01	Critical	System shall support multiple concurrent trading strategies
FR-S02	Critical	System shall generate buy/sell/hold signals with confidence scores
FR-S03	High	System shall support strategy parameter optimization
FR-S04	High	System shall enable/disable strategies without system restart
FR-S05	Medium	System shall support strategy combination and ensemble methods

2.1.3 Execution and Order Management

2.2 Non-Functional Requirements

Table 2.3: Execution Requirements

ID	Priority	Requirement Description
FR-E01	Critical	System shall submit orders via broker API
FR-E02	Critical	System shall track order status throughout lifecycle
FR-E03	Critical	System shall support market, limit, and stop orders
FR-E04	High	System shall implement execution algorithms (TWAP, VWAP)
FR-E05	High	System shall provide paper trading mode for validation
FR-E06	High	System shall handle partial fills and order modifications

2.2.1 Performance Requirements

Performance Targets

The system is designed for standard algorithmic trading, not high-frequency trading (HFT). Target latencies are measured in hundreds of milliseconds rather than microseconds.

Table 2.4: Performance Requirements

ID	Metric	Target Value
NFR-P01	Signal Generation Latency	< 500ms from data receipt to signal
NFR-P02	Order Submission Latency	< 200ms from signal to order sent
NFR-P03	Concurrent Symbols	≥ 1000 symbols monitored simultaneously
NFR-P04	Historical Data Processing	≥ 10 years of data processed in < 1 hour
NFR-P05	System Uptime	$\geq 99.9\%$ during market hours

2.2.2 Security Requirements

- **NFR-S01**: All API credentials shall be encrypted at rest using AES-256
- **NFR-S02**: Network communications shall use TLS 1.3 or higher
- **NFR-S03**: System shall implement role-based access control
- **NFR-S04**: All trading actions shall be logged with non-repudiation guarantees
- **NFR-S05**: System shall support multi-factor authentication for administrative access

2.2.3 Auditability Requirements

- **NFR-A01**: All signals generated shall be logged with timestamps and rationale
- **NFR-A02**: All orders shall be tracked from creation through settlement
- **NFR-A03**: Portfolio state shall be snapshotted at configurable intervals
- **NFR-A04**: System shall generate regulatory-compliant trade reports

Chapter 3

System Architecture

3.1 High-Level Architecture Overview

The system follows a modular, event-driven architecture that separates concerns across distinct components. This design enables independent scaling, testing, and deployment of each module.

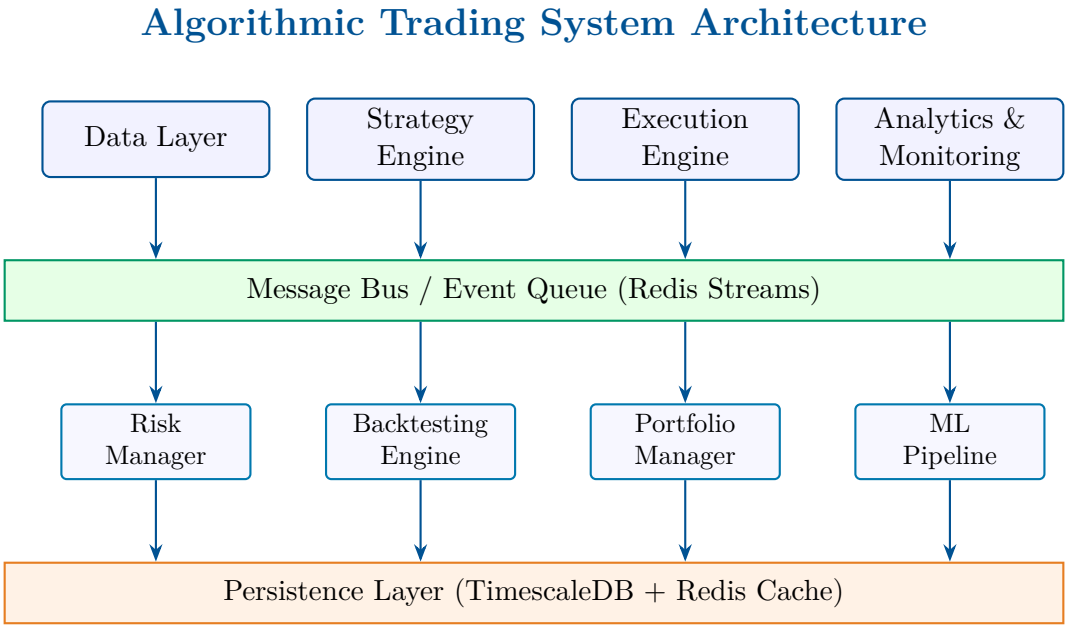


Figure 3.1: High-Level System Architecture Diagram

3.2 Component Specifications

3.2.1 Data Layer

The Data Layer is responsible for all data acquisition, normalization, and feature engineering operations.

Table 3.1: Data Layer Component Structure

Sub-Module	File	Responsibility
Connectors	`market_data_feed.py`	Real-time streaming price feeds
	`historical_data.py`	Historical OHLCV data retrieval
	`fundamental_data.py`	Financial statements, ratios, earnings
	`alternative_data.py`	News, sentiment, SEC filings
Processors	`data_cleaner.py`	Handle missing data, outliers, gaps
	`normalizer.py`	Standardize formats across sources
	`feature_engineer.py`	Technical indicators, derived features
Storage	`time_series_db.py`	TimescaleDB interface
	`cache_manager.py`	Redis caching for hot data

3.2.2 Strategy Engine

The Strategy Engine implements the trading logic, signal generation, and position sizing algorithms.

Table 3.2: Strategy Engine Component Structure

Category	File	Description
Base	`strategy_interface.py`	Abstract base class for strategies
	`signal.py`	Signal data structures
	`position.py`	Position sizing logic
Technical	`moving_average.py`	MA crossover strategies
	`momentum.py`	RSI, MACD-based strategies
	`mean_reversion.py`	Bollinger Bands, statistical arbitrage
	`canslim.py`	O'Neil's CAN SLIM implementation
	`pattern_recognition.py`	Chart pattern detection
Fundamental	`value_investing.py`	Graham-style value metrics
	`growth_investing.py`	PEG ratio, revenue growth
	`quality_screen.py`	ROE, debt ratios, margins
Quantitative	`statistical_arb.py`	Pairs trading, cointegration
	`factor_models.py`	Fama-French factor implementation
	`regime_detection.py`	Market regime classification
ML-Based	`lstm_predictor.py`	Price direction prediction
	`random_forest_signal.py`	Classification-based signals
	`reinforcement_agent.py`	RL-based trading agent
	`ensemble_strategy.py`	Multi-model combination

3.2.3 Execution Engine

The Execution Engine manages all aspects of order lifecycle from creation through settlement.

3.3 Data Flow Architecture

Table 3.3: Execution Engine Component Structure

Sub-Module	**File**	**Responsibility**
Order Management	`order.py`	Order objects (market, limit, stop)
	`order_router.py`	Route orders to appropriate broker
	`order_tracker.py`	Track order status through lifecycle
Brokers	`broker_interface.py`	Abstract broker API
	`alpaca_broker.py`	Alpaca Markets integration
	`interactive_brokers.py`	IBKR TWS integration
	`paper_trading.py`	Simulated execution for testing
Algorithms	`twap.py`	Time-weighted average price execution
	`vwap.py`	Volume-weighted average price execution
	`iceberg.py`	Iceberg order execution
Cost Models	`slippage_model.py`	Estimate execution costs
	`market_impact.py`	Model market impact of large orders

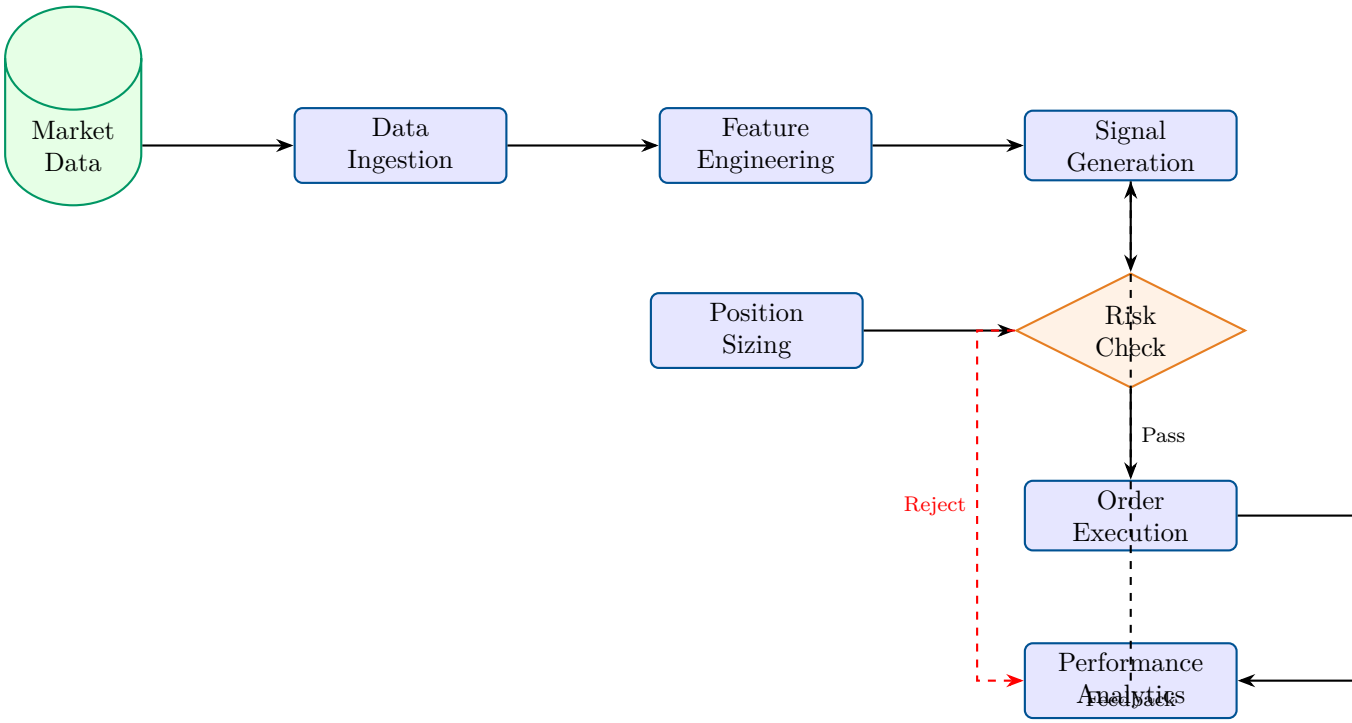


Figure 3.2: Data Flow Through the Trading System

3.4 Database Schema Design

The persistence layer uses TimescaleDB, a PostgreSQL extension optimized for time-series data.

Listing 3.1: Core Database Schema

```sql
-- Core price data (TimescaleDB hypertable)
CREATE TABLE ohlcv (
    symbol          VARCHAR(10) NOT NULL,
    timestamp       TIMESTAMPTZ NOT NULL,
    open            DECIMAL(12,4),
    high            DECIMAL(12,4),
    low             DECIMAL(12,4),
    close           DECIMAL(12,4),
    volume          BIGINT,
    adjusted_close  DECIMAL(12,4),
    PRIMARY KEY (symbol, timestamp)
);
SELECT create_hypertable('ohlcv', 'timestamp');

-- Trading signals generated by strategies
CREATE TABLE signals (
    id              SERIAL PRIMARY KEY,
    strategy_id     VARCHAR(50) NOT NULL,
    symbol          VARCHAR(10) NOT NULL,
    timestamp       TIMESTAMPTZ NOT NULL,
    signal_type     VARCHAR(10),  -- BUY, SELL, HOLD
    strength        DECIMAL(5,4), -- 0.0 to 1.0
    metadata        JSONB
);
CREATE INDEX idx_signals_symbol_ts ON signals(symbol, timestamp);

-- Order and execution tracking
CREATE TABLE orders (
    id              UUID PRIMARY KEY,
    symbol          VARCHAR(10) NOT NULL,
    side            VARCHAR(4),   -- BUY, SELL
    order_type      VARCHAR(10),  -- MARKET, LIMIT, STOP
    quantity        INTEGER,
    limit_price     DECIMAL(12,4),
    status          VARCHAR(20),
    created_at      TIMESTAMPTZ,
    filled_at       TIMESTAMPTZ,
    filled_price    DECIMAL(12,4),
    commission      DECIMAL(10,4)
);
CREATE INDEX idx_orders_status ON orders(status, created_at);

-- Portfolio state snapshots
CREATE TABLE portfolio_snapshots (
    id              SERIAL PRIMARY KEY,
    timestamp       TIMESTAMPTZ NOT NULL,
    total_value     DECIMAL(15,2),
    cash            DECIMAL(15,2),
    positions       JSONB,
    daily_pnl       DECIMAL(12,2),
    cumulative_pnl  DECIMAL(15,2)
```

```
52 );
53 SELECT create_hypertable('portfolio_snapshots', 'timestamp');
```

Chapter 4

Data Infrastructure

4.1 Data Sources Integration

Successful algorithmic trading requires diverse, high-quality data sources. The system integrates multiple providers to ensure comprehensive market coverage.

Table 4.1: Data Source Specifications

Data Type	Source Options	Update Freq.	Purpose
Real-time Prices	Alpaca, Polygon.io, IEX Cloud	Streaming	Live trading signals
Historical OHLCV	Yahoo Finance, Alpha Vantage	Daily	Backtesting
Fundamentals	SEC EDGAR, Financial Modeling Prep	Quarterly	Value screening
News/Sentiment	NewsAPI, Twitter API, Reddit	Real-time	Sentiment analysis
Economic Data	FRED, World Bank	Monthly	Macro regime detection
Options Data	CBOE, Polygon.io	Daily	Volatility analysis

4.2 Feature Engineering Pipeline

4.2.1 Technical Indicators Implementation

The feature engineering pipeline implements indicators from Murphy's *Technical Analysis of the Financial Markets* and Pring's *Technical Analysis Explained*.

Listing 4.1: Technical Indicators Implementation

```
import pandas as pd
import numpy as np
from typing import Tuple

class TechnicalIndicators:
    """
```

```
    Implements technical indicators from 'Technical Analysis
    of the Financial Markets' by John J. Murphy.
    """

    @staticmethod
    def sma(prices: pd.Series, period: int) → pd.Series:
        """Simple Moving Average"""
        return prices.rolling(window=period).mean()

    @staticmethod
    def ema(prices: pd.Series, period: int) → pd.Series:
        """Exponential Moving Average"""
        return prices.ewm(span=period, adjust=False).mean()

    @staticmethod
    def rsi(prices: pd.Series, period: int = 14) → pd.Series:
        """
        Relative Strength Index

        RSI = 100 - (100 / (1 + RS))
        where RS = Average Gain / Average Loss
        """
        delta = prices.diff()
        gain = delta.where(delta > 0,
           0).rolling(window=period).mean()
        loss = (-delta.where(delta < 0,
           0)).rolling(window=period).mean()
        rs = gain / loss
        return 100 - (100 / (1 + rs))

    @staticmethod
    def macd(prices: pd.Series,
             fast: int = 12,
             slow: int = 26,
             signal: int = 9) → Tuple[pd.Series, pd.Series,
                pd.Series]:
        """
        Moving Average Convergence Divergence

        Returns: (macd_line, signal_line, histogram)
        """
        ema_fast = prices.ewm(span=fast, adjust=False).mean()
        ema_slow = prices.ewm(span=slow, adjust=False).mean()
        macd_line = ema_fast - ema_slow
        signal_line = macd_line.ewm(span=signal,
           adjust=False).mean()
        histogram = macd_line - signal_line
        return macd_line, signal_line, histogram

    @staticmethod
    def bollinger_bands(prices: pd.Series,
                        period: int = 20,
                        std_dev: float = 2.0) → Tuple[pd.Series,
                           pd.Series, pd.Series]:
        """
        Bollinger Bands for mean reversion strategies

        Returns: (upper_band, middle_band, lower_band)
```

```
        """
        middle = prices.rolling(window=period).mean()
        std = prices.rolling(window=period).std()
        upper = middle + (std_dev * std)
        lower = middle - (std_dev * std)
        return upper, middle, lower

    @staticmethod
    def atr(high: pd.Series,
            low: pd.Series,
            close: pd.Series,
            period: int = 14) -> pd.Series:
        """
        Average True Range - volatility indicator

        Used for position sizing and stop-loss placement
        """
        tr1 = high - low
        tr2 = abs(high - close.shift(1))
        tr3 = abs(low - close.shift(1))
        true_range = pd.concat([tr1, tr2, tr3],
            axis=1).max(axis=1)
        return true_range.rolling(window=period).mean()

    @staticmethod
    def stochastic_oscillator(high: pd.Series,
                              low: pd.Series,
                              close: pd.Series,
                              k_period: int = 14,
                              d_period: int = 3) ->
                                 Tuple[pd.Series, pd.Series]:
        """
        Stochastic Oscillator

        %K = (Close - Lowest Low) / (Highest High - Lowest Low) *
            100
        %D = 3-period SMA of %K
        """
        lowest_low = low.rolling(window=k_period).min()
        highest_high = high.rolling(window=k_period).max()
        k = ((close - lowest_low) / (highest_high - lowest_low))
            * 100
        d = k.rolling(window=d_period).mean()
        return k, d
```

4.2.2 Advanced Feature Engineering

For machine learning models, additional feature engineering techniques from López de Prado's *Advances in Financial Machine Learning* are implemented:

Listing 4.2: Advanced Feature Engineering for ML

```
class AdvancedFeatures:
    """
    Advanced feature engineering techniques from
    'Advances in Financial Machine Learning' by Marcos Lopez de
        Prado
```

```
    """

    @staticmethod
    def fractional_diff(series: pd.Series,
                        d: float = 0.4,
                        threshold: float = 1e-5) -> pd.Series:
        """
        Fractional differentiation to maintain memory while
        achieving stationarity.

        This technique preserves long-term dependencies in the
            data
        while making it stationary for ML models.

        Parameters:
            series: Price series to differentiate
            d: Differentiation order (0 < d < 1)
            threshold: Weight threshold for truncation
        """
        weights = [1.0]
        k = 1
        while abs(weights[-1]) > threshold:
            weight = -weights[-1] * (d - k + 1) / k
            weights.append(weight)
            k += 1
        weights = np.array(weights[::-1])

        result = []
        for i in range(len(weights) - 1, len(series)):
            window = series.iloc[i - len(weights) + 1:i +
                1].values
            result.append(np.dot(weights, window))

        return pd.Series(
            result,
            index=series.index[len(weights) - 1:]
        )

    @staticmethod
    def compute_dollar_bars(trades: pd.DataFrame,
                            dollar_threshold: float) ->
                               pd.DataFrame:
        """
        Dollar bars: sample data when cumulative dollar volume
        reaches a threshold. This normalizes for varying trading
        activity across time.
        """
        trades['dollar_volume'] = trades['price'] *
            trades['volume']
        trades['cum_dollar'] = trades['dollar_volume'].cumsum()

        bars = []
        current_bar = {'open': None, 'high': -np.inf,
                       'low': np.inf, 'volume': 0}
        last_threshold = 0

        for _, trade in trades.iterrows():
            if current_bar['open'] is None:
```

```
                current_bar['open'] = trade['price']

            current_bar['high'] = max(current_bar['high'],
                trade['price'])
            current_bar['low'] = min(current_bar['low'],
                trade['price'])
            current_bar['volume'] += trade['volume']
            current_bar['close'] = trade['price']
            current_bar['timestamp'] = trade['timestamp']

            if trade['cum_dollar'] - last_threshold >=
                dollar_threshold:
                bars.append(current_bar.copy())
                last_threshold = trade['cum_dollar']
                current_bar = {'open': None, 'high': -np.inf,
                               'low': np.inf, 'volume': 0}

        return pd.DataFrame(bars)

    @staticmethod
    def get_entropy(series: pd.Series, bins: int = 10) -> float:
        """
        Shannon entropy for measuring information content
        in a price series.
        """
        counts, _ = np.histogram(series, bins=bins)
        probs = counts / len(series)
        probs = probs[probs > 0]  # Remove zeros
        return -np.sum(probs * np.log2(probs))
```

Chapter 5

Strategy Development

5.1 Strategy Framework Design

All trading strategies inherit from a common base class that enforces consistent interfaces and behavior.

Listing 5.1: Strategy Base Class

```python
from abc import ABC, abstractmethod
from dataclasses import dataclass, field
from enum import Enum
from typing import Dict, List, Optional
import pandas as pd

class SignalType(Enum):
    """Trading signal types"""
    BUY = "BUY"
    SELL = "SELL"
    HOLD = "HOLD"

@dataclass
class Signal:
    """
    Trading signal generated by a strategy

    Attributes:
        symbol: Stock ticker symbol
        signal_type: BUY, SELL, or HOLD
        strength: Confidence score from 0.0 to 1.0
        timestamp: When the signal was generated
        metadata: Additional context (reason, indicators used,
            etc.)
    """
    symbol: str
    signal_type: SignalType
    strength: float
    timestamp: pd.Timestamp
    metadata: Dict = field(default_factory=dict)

    def __post_init__(self):
        if not 0.0 <= self.strength <= 1.0:
```

```
33              raise ValueError("Signal strength must be between 0
                    and 1")
34
35  class BaseStrategy(ABC):
36      """
37      Abstract base class for all trading strategies.
38
39      This design is inspired by systematic approaches described in
40      Ernie Chan's 'Algorithmic Trading: Winning Strategies and
41      Their Rationale'.
42      """
43
44      def __init__(self,
45                   name: str,
46                   universe: List[str],
47                   params: Optional[Dict] = None):
48          self.name = name
49          self.universe = universe
50          self.params = params or {}
51          self.positions: Dict[str, int] = {}
52          self._is_active = True
53
54      @abstractmethod
55      def generate_signals(self,
56                           market_data: pd.DataFrame) ->
                                 List[Signal]:
57          """
58          Generate trading signals based on market data.
59
60          Must be implemented by all concrete strategies.
61
62          Args:
63              market_data: DataFrame with OHLCV data for all symbols
64
65          Returns:
66              List of Signal objects
67          """
68          pass
69
70      @abstractmethod
71      def calculate_position_size(self,
72                                  signal: Signal,
73                                  portfolio_value: float,
74                                  current_price: float) -> int:
75          """
76          Determine position size for a given signal.
77
78          Args:
79              signal: The trading signal
80              portfolio_value: Total portfolio value
81              current_price: Current price of the security
82
83          Returns:
84              Number of shares to trade
85          """
86          pass
87
88      def validate_signal(self, signal: Signal) -> bool:
```

```
        """
        Apply validation rules before acting on a signal.

        Can be overridden by subclasses for custom validation.
        """
        min_strength = self.params.get('min_signal_strength', 0.5)
        return signal.strength >= min_strength

    def activate(self):
        """Enable the strategy"""
        self._is_active = True

    def deactivate(self):
        """Disable the strategy without affecting positions"""
        self._is_active = False

    @property
    def is_active(self) -> bool:
        return self._is_active
```

5.2 CAN SLIM Strategy Implementation

The CAN SLIM strategy implements William O'Neil's proven 7-step process from *How to Make Money in Stocks*.

CAN SLIM Criteria

- **C** – Current quarterly earnings per share (up 25%+)
- **A** – Annual earnings growth (25%+ over 5 years)
- **N** – New products, management, or price highs
- **S** – Supply and demand (shares outstanding + volume)
- **L** – Leader or laggard (relative strength ranking)
- **I** – Institutional sponsorship (10-60% ownership)
- **M** – Market direction (follow the general market)

Listing 5.2: CAN SLIM Strategy Implementation

```
class CANSLIMStrategy(BaseStrategy):
    """
    Implementation of William O'Neil's CAN SLIM system from
    'How to Make Money in Stocks'.

    This strategy identifies growth stocks with strong
       fundamentals
    and technical momentum at optimal buying points.
    """

    def __init__(self, universe: List[str]):
        super().__init__(
            name="CANSLIM",
            universe=universe,
            params={
                'min_eps_growth_qoq': 0.25,      # 25% quarterly
                'min_eps_growth_annual': 0.25,   # 25% annual
                'min_relative_strength': 80,     # RS rank >= 80
```

```
18                 'min_institutional': 0.10,        # 10%
                       institutional
19                 'max_institutional': 0.60,        # Not over-owned
20                 'volume_surge_ratio': 1.5,        # 50% above
                       average
21                 'price_to_52w_high': 0.95,        # Within 5% of
                       high
22             }
23         )
24         self.market_direction = None  # M factor
25
26     def _score_c_factor(self, fundamentals: Dict) -> float:
27         """Score current quarterly earnings"""
28         eps_growth = fundamentals.get('eps_growth_qoq', 0)
29         if eps_growth >= self.params['min_eps_growth_qoq']:
30             # Higher growth = higher score
31             return min(1.0, eps_growth / 0.50)  # Cap at 50%
                   growth
32         return 0.0
33
34     def _score_a_factor(self, fundamentals: Dict) -> float:
35         """Score annual earnings growth"""
36         annual_growth = fundamentals.get('eps_growth_annual_5yr',
               0)
37         if annual_growth >= self.params['min_eps_growth_annual']:
38             return min(1.0, annual_growth / 0.50)
39         return 0.0
40
41     def _score_n_factor(self,
42                         fundamentals: Dict,
43                         technicals: Dict) -> float:
44         """Score new highs and catalysts"""
45         score = 0.0
46
47         # Near 52-week high
48         price_vs_high = technicals.get('price_vs_52w_high', 0)
49         if price_vs_high >= self.params['price_to_52w_high']:
50             score += 0.5
51
52         # New product or catalyst (if available)
53         if fundamentals.get('has_recent_catalyst', False):
54             score += 0.5
55
56         return score
57
58     def _score_s_factor(self, technicals: Dict) -> float:
59         """Score supply and demand (volume analysis)"""
60         volume_ratio = technicals.get('volume_vs_avg', 1.0)
61         if volume_ratio >= self.params['volume_surge_ratio']:
62             return min(1.0, (volume_ratio - 1.0) / 1.0)
63         return 0.0
64
65     def _score_l_factor(self, technicals: Dict) -> float:
66         """Score leadership (relative strength)"""
67         rs_rank = technicals.get('relative_strength_rank', 0)
68         if rs_rank >= self.params['min_relative_strength']:
69             return (rs_rank - 80) / 20  # Scale 80-100 to 0-1
70         return 0.0
```

```
71
72      def _score_i_factor(self, fundamentals: Dict) → float:
73          """Score institutional sponsorship"""
74          inst_ownership =
               fundamentals.get('institutional_ownership', 0)
75          min_inst = self.params['min_institutional']
76          max_inst = self.params['max_institutional']
77
78          if min_inst <= inst_ownership <= max_inst:
79              # Optimal range gets full score
80              return 1.0
81          elif inst_ownership < min_inst:
82              return inst_ownership / min_inst
83          else:
84              # Over-owned is negative
85              return max(0, 1 - (inst_ownership - max_inst) / 0.20)
86
87      def _check_m_factor(self) → bool:
88          """Check market direction (requires separate analysis)"""
89          # In practice, this would check:
90          # - Major indices above 50-day MA
91          # - Number of new highs vs new lows
92          # - Distribution days count
93          return self.market_direction in ['uptrend',
               'confirmed_uptrend']
94
95      def score_stock(self,
96                      fundamentals: Dict,
97                      technicals: Dict) → float:
98          """
99          Calculate overall CAN SLIM score for a stock.
100
101         Returns score from 0.0 to 1.0
102         """
103         scores = {
104             'C': self._score_c_factor(fundamentals),
105             'A': self._score_a_factor(fundamentals),
106             'N': self._score_n_factor(fundamentals, technicals),
107             'S': self._score_s_factor(technicals),
108             'L': self._score_l_factor(technicals),
109             'I': self._score_i_factor(fundamentals),
110         }
111
112         # Weighted average (L is most important per O'Neil)
113         weights = {'C': 0.15, 'A': 0.15, 'N': 0.10,
114                    'S': 0.15, 'L': 0.25, 'I': 0.20}
115
116         total_score = sum(
117             scores[factor] * weights[factor]
118             for factor in scores
119         )
120
121         return total_score
122
123     def generate_signals(self,
124                          market_data: pd.DataFrame) →
                                List[Signal]:
125         """Generate CAN SLIM buy signals"""
```

```
        signals = []

        # Check market direction first (M factor)
        if not self._check_m_factor():
            return signals  # No buys in downtrend

        for symbol in self.universe:
            # Get fundamentals and technicals for symbol
            fundamentals = self._get_fundamentals(symbol)
            technicals = self._calculate_technicals(
                market_data[market_data['symbol'] == symbol]
            )

            score = self.score_stock(fundamentals, technicals)

            if score >= 0.7:  # Strong candidate
                signals.append(Signal(
                    symbol=symbol,
                    signal_type=SignalType.BUY,
                    strength=score,
                    timestamp=pd.Timestamp.now(),
                    metadata={
                        'strategy': 'CANSLIM',
                        'factors': self._get_factor_breakdown(
                            fundamentals, technicals
                        )
                    }
                ))

        return signals

    def calculate_position_size(self,
                                signal: Signal,
                                portfolio_value: float,
                                current_price: float) -> int:
        """
        Position sizing based on signal strength and O'Neil's
        pyramid buying approach.
        """
        # Base allocation: 5-10% of portfolio per position
        base_pct = 0.05 + (signal.strength - 0.7) * 0.15
        allocation = portfolio_value * base_pct
        shares = int(allocation / current_price)
        return shares
```

5.3 Mean Reversion Strategy

Mean reversion strategies capitalize on the tendency of prices to return to their historical averages.

Listing 5.3: Mean Reversion Strategy Implementation

```
class MeanReversionStrategy(BaseStrategy):
    """
    Bollinger Band mean reversion strategy based on concepts
    from 'Technical Analysis of the Financial Markets' by Murphy
```

```
    and 'Technical Analysis Explained' by Pring.

    This strategy buys when price is oversold (below lower band)
    and sells when overbought (above upper band).
    """

    def __init__(self, universe: List[str]):
        super().__init__(
            name="MeanReversion",
            universe=universe,
            params={
                'bb_period': 20,
                'bb_std': 2.0,
                'rsi_period': 14,
                'rsi_oversold': 30,
                'rsi_overbought': 70,
                'min_holding_period': 5,  # days
                'profit_target': 0.05,    # 5%
                'stop_loss': 0.03,        # 3%
            }
        )
        self.entry_prices: Dict[str, float] = {}
        self.entry_dates: Dict[str, pd.Timestamp] = {}

    def generate_signals(self,
                         market_data: pd.DataFrame) →
                            List[Signal]:
        """Generate mean reversion signals"""
        signals = []

        for symbol in self.universe:
            symbol_data = market_data[
                market_data['symbol'] == symbol
            ].copy()

            if len(symbol_data) < self.params['bb_period']:
                continue

            prices = symbol_data['close']

            # Calculate indicators
            upper, middle, lower =
                TechnicalIndicators.bollinger_bands(
                prices,
                self.params['bb_period'],
                self.params['bb_std']
            )
            rsi = TechnicalIndicators.rsi(
                prices,
                self.params['rsi_period']
            )

            current_price = prices.iloc[-1]
            current_rsi = rsi.iloc[-1]
            current_upper = upper.iloc[-1]
            current_lower = lower.iloc[-1]
            current_middle = middle.iloc[-1]

```

```
61              # Check for existing position
62              has_position = symbol in self.entry_prices
63
64              if has_position:
65                  # Check exit conditions
66                  signal = self._check_exit(
67                      symbol, current_price, current_rsi,
68                      current_upper, current_middle
69                  )
70                  if signal:
71                      signals.append(signal)
72              else:
73                  # Check entry conditions
74                  signal = self._check_entry(
75                      symbol, current_price, current_rsi,
76                      current_lower, current_upper
77                  )
78                  if signal:
79                      signals.append(signal)
80
81          return signals
82
83      def _check_entry(self,
84                       symbol: str,
85                       price: float,
86                       rsi: float,
87                       lower_band: float,
88                       upper_band: float) -> Optional[Signal]:
89          """Check for entry conditions"""
90
91          # Long entry: price below lower band + RSI oversold
92          if price < lower_band and rsi <
                self.params['rsi_oversold']:
93              # Strength based on how far below band
94              deviation = (lower_band - price) / price
95              strength = min(1.0, deviation / 0.05)  # Cap at 5%
                    deviation
96
97              return Signal(
98                  symbol=symbol,
99                  signal_type=SignalType.BUY,
100                 strength=strength,
101                 timestamp=pd.Timestamp.now(),
102                 metadata={
103                     'strategy': 'MeanReversion',
104                     'entry_type': 'oversold_bounce',
105                     'rsi': rsi,
106                     'bb_position': 'below_lower'
107                 }
108             )
109
110         return None
111
112     def _check_exit(self,
113                     symbol: str,
114                     price: float,
115                     rsi: float,
116                     upper_band: float,
```

```
                    middle_band: float) → Optional[Signal]:
        """Check for exit conditions"""

        entry_price = self.entry_prices.get(symbol, price)
        pnl_pct = (price - entry_price) / entry_price

        # Exit conditions:
        # 1. Price above upper band (overbought)
        # 2. RSI overbought
        # 3. Profit target reached
        # 4. Stop loss triggered

        should_exit = (
            (price > upper_band and rsi >
               self.params['rsi_overbought']) or
            pnl_pct >= self.params['profit_target'] or
            pnl_pct <= -self.params['stop_loss']
        )

        if should_exit:
            return Signal(
                symbol=symbol,
                signal_type=SignalType.SELL,
                strength=1.0,  # Full exit
                timestamp=pd.Timestamp.now(),
                metadata={
                    'strategy': 'MeanReversion',
                    'exit_type': self._determine_exit_type(
                        price, upper_band, rsi, pnl_pct
                    ),
                    'pnl_pct': pnl_pct
                }
            )

        return None

    def _determine_exit_type(self,
                             price: float,
                             upper: float,
                             rsi: float,
                             pnl: float) → str:
        """Determine the reason for exit"""
        if pnl >= self.params['profit_target']:
            return 'profit_target'
        elif pnl <= -self.params['stop_loss']:
            return 'stop_loss'
        elif price > upper:
            return 'overbought'
        else:
            return 'other'

    def calculate_position_size(self,
                                signal: Signal,
                                portfolio_value: float,
                                current_price: float) → int:
        """
        Position sizing with volatility adjustment.

```

```
174         Mean reversion positions are sized inversely to recent
175         volatility to maintain consistent risk.
176         """
177         base_allocation = 0.10 * signal.strength
178         allocation = portfolio_value * base_allocation
179         shares = int(allocation / current_price)
180         return shares
```

Chapter 6

Backtesting Framework

6.1 Backtester Design Principles

A robust backtesting framework is essential for strategy validation. The design follows principles from Ernie Chan's *Algorithmic Trading* and Kevin Davey's *Building Winning Algorithmic Trading Systems*.

Important

Common Backtesting Pitfalls to Avoid:

1. **Look-ahead bias**: Using information not available at decision time
2. **Survivorship bias**: Only testing on stocks that still exist
3. **Overfitting**: Optimizing parameters to historical noise
4. **Ignoring costs**: Not accounting for commissions and slippage
5. **Data snooping**: Testing too many variations on same data

6.2 Backtester Implementation

Listing 6.1: Backtester Core Implementation

```
from dataclasses import dataclass
from typing import List, Dict, Optional
import pandas as pd
import numpy as np

@dataclass
class BacktestConfig:
    """Configuration for backtest execution"""
    start_date: str
    end_date: str
    initial_capital: float
    commission_rate: float = 0.001    # 0.1% per trade
    slippage_model: str = "fixed"     # fixed, percentage, volume
    slippage_bps: float = 5.0         # basis points
    margin_requirement: float = 1.0   # 1.0 = no margin

@dataclass
class BacktestResults:
    """Container for backtest performance metrics"""
```

```
    total_return: float
    annualized_return: float
    sharpe_ratio: float
    sortino_ratio: float
    max_drawdown: float
    calmar_ratio: float
    win_rate: float
    profit_factor: float
    total_trades: int
    avg_trade_duration: float
    equity_curve: pd.Series
    drawdown_series: pd.Series
    trades: pd.DataFrame
    monthly_returns: pd.Series

class EventDrivenBacktester:
    """
    Event-driven backtesting engine.

    This architecture processes market events chronologically,
    simulating realistic order fills and portfolio updates.
    """

    def __init__(self, config: BacktestConfig):
        self.config = config
        self.portfolio =
            SimulatedPortfolio(config.initial_capital)
        self.trades: List[Dict] = []
        self.equity_history: List[Dict] = []
        self.slippage_model = self._create_slippage_model()

    def _create_slippage_model(self) -> 'SlippageModel':
        """Factory method for slippage models"""
        if self.config.slippage_model == "fixed":
            return FixedSlippage(self.config.slippage_bps)
        elif self.config.slippage_model == "percentage":
            return PercentageSlippage(self.config.slippage_bps)
        elif self.config.slippage_model == "volume":
            return VolumeBasedSlippage(self.config.slippage_bps)
        else:
            raise ValueError(f"Unknown slippage model")

    def run(self,
            strategy: BaseStrategy,
            market_data: pd.DataFrame) -> BacktestResults:
        """
        Execute backtest simulation.

        Args:
            strategy: Strategy instance to test
            market_data: Historical OHLCV data

        Returns:
            BacktestResults with performance metrics
        """
        # Sort data chronologically
        market_data = market_data.sort_values('timestamp')
        dates = market_data['timestamp'].dt.date.unique()
```

```
77
78          for date in dates:
79              # Get data up to current date (no look-ahead)
80              historical_data = market_data[
81                  market_data['timestamp'].dt.date <= date
82              ]
83              current_data = market_data[
84                  market_data['timestamp'].dt.date == date
85              ]
86
87              # Generate signals using only historical data
88              signals = strategy.generate_signals(historical_data)
89
90              # Execute valid signals
91              for signal in signals:
92                  if strategy.validate_signal(signal):
93                      self._execute_signal(signal, current_data)
94
95              # Update portfolio mark-to-market
96              self._update_portfolio(current_data)
97
98              # Record equity
99              self.equity_history.append({
100                 'date': date,
101                 'equity': self.portfolio.total_value,
102                 'cash': self.portfolio.cash,
103                 'positions_value': self.portfolio.positions_value
104             })
105
106         return self._calculate_results()
107
108     def _execute_signal(self,
109                         signal: Signal,
110                         current_data: pd.DataFrame):
111         """Execute a trading signal with realistic fills"""
112
113         symbol_data = current_data[
114             current_data['symbol'] == signal.symbol
115         ]
116
117         if symbol_data.empty:
118             return
119
120         # Use open price for fill (next bar execution)
121         base_price = symbol_data['open'].iloc[0]
122         volume = symbol_data['volume'].iloc[0]
123
124         # Apply slippage
125         fill_price = self.slippage_model.apply(
126             base_price,
127             signal.signal_type,
128             volume
129         )
130
131         # Calculate position size
132         position_size = self._calculate_shares(
133             signal, fill_price
134         )
```

```
135
136         if position_size == 0:
137             return
138
139         # Calculate commission
140         commission = abs(position_size * fill_price *
141                          self.config.commission_rate)
142
143         # Execute trade
144         if signal.signal_type == SignalType.BUY:
145             cost = position_size * fill_price + commission
146             if cost <= self.portfolio.cash:
147                 self.portfolio.buy(
148                     signal.symbol, position_size, fill_price
149                 )
150                 self.portfolio.cash -= commission
151                 self._record_trade(
152                     signal, position_size, fill_price, commission
153                 )
154
155         elif signal.signal_type == SignalType.SELL:
156             current_position =
                    self.portfolio.get_position(signal.symbol)
157             if current_position > 0:
158                 sell_qty = min(position_size, current_position)
159                 self.portfolio.sell(
160                     signal.symbol, sell_qty, fill_price
161                 )
162                 self.portfolio.cash -= commission
163                 self._record_trade(
164                     signal, -sell_qty, fill_price, commission
165                 )
166
167     def _calculate_results(self) -> BacktestResults:
168         """Calculate comprehensive performance metrics"""
169
170         equity_df = pd.DataFrame(self.equity_history)
171         equity = equity_df['equity']
172
173         # Daily returns
174         returns = equity.pct_change().dropna()
175
176         # Total and annualized return
177         total_return = (equity.iloc[-1] / equity.iloc[0]) - 1
178         n_days = len(returns)
179         annualized_return = (1 + total_return) ** (252 / n_days)
                - 1
180
181         # Risk metrics
182         volatility = returns.std() * np.sqrt(252)
183         sharpe_ratio = (annualized_return / volatility
184                         if volatility > 0 else 0)
185
186         # Sortino (downside deviation)
187         downside_returns = returns[returns < 0]
188         downside_std = downside_returns.std() * np.sqrt(252)
189         sortino_ratio = (annualized_return / downside_std
190                          if downside_std > 0 else 0)
```

```
191
192         # Drawdown analysis
193         cumulative = (1 + returns).cumprod()
194         running_max = cumulative.expanding().max()
195         drawdown = (cumulative - running_max) / running_max
196         max_drawdown = drawdown.min()
197
198         # Calmar ratio
199         calmar_ratio = (annualized_return / abs(max_drawdown)
200                         if max_drawdown != 0 else 0)
201
202         # Trade analysis
203         trades_df = pd.DataFrame(self.trades)
204         if len(trades_df) > 0:
205             trades_df['pnl'] =
                    self._calculate_trade_pnl(trades_df)
206             winning = trades_df[trades_df['pnl'] > 0]
207             losing = trades_df[trades_df['pnl'] < 0]
208
209             win_rate = len(winning) / len(trades_df)
210
211             gross_profit = winning['pnl'].sum() if len(winning) >
                    0 else 0
212             gross_loss = abs(losing['pnl'].sum()) if len(losing)
                    > 0 else 1
213             profit_factor = gross_profit / gross_loss
214
215             avg_duration = trades_df['duration'].mean()
216         else:
217             win_rate = 0
218             profit_factor = 0
219             avg_duration = 0
220             trades_df = pd.DataFrame()
221
222         # Monthly returns
223         equity_df['date'] = pd.to_datetime(equity_df['date'])
224         monthly =
                equity_df.set_index('date')['equity'].resample('M').last()
225         monthly_returns = monthly.pct_change().dropna()
226
227         return BacktestResults(
228             total_return=total_return,
229             annualized_return=annualized_return,
230             sharpe_ratio=sharpe_ratio,
231             sortino_ratio=sortino_ratio,
232             max_drawdown=max_drawdown,
233             calmar_ratio=calmar_ratio,
234             win_rate=win_rate,
235             profit_factor=profit_factor,
236             total_trades=len(self.trades),
237             avg_trade_duration=avg_duration,
238             equity_curve=equity,
239             drawdown_series=drawdown,
240             trades=trades_df,
241             monthly_returns=monthly_returns
242         )
```

6.3 Walk-Forward Optimization

Walk-forward analysis prevents overfitting by validating optimized parameters on unseen data.

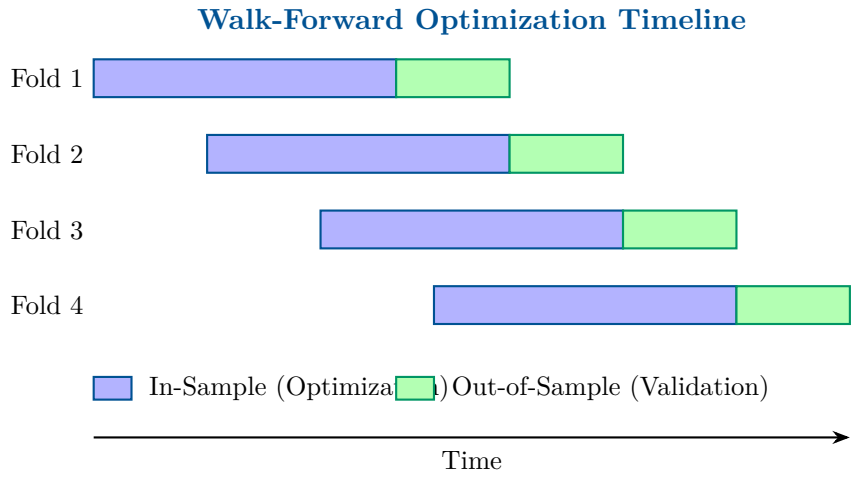


Figure 6.1: Walk-Forward Optimization Process

Listing 6.2: Walk-Forward Optimizer Implementation

```
class WalkForwardOptimizer:
    """
    Walk-forward optimization to prevent overfitting.

    This technique is recommended in Kevin Davey's
    'Building Winning Algorithmic Trading Systems'.
    """

    def __init__(self,
                 in_sample_pct: float = 0.70,
                 num_folds: int = 5,
                 embargo_periods: int = 5):
        """
        Args:
            in_sample_pct: Percentage of each fold for
                optimization
            num_folds: Number of walk-forward folds
            embargo_periods: Gap between in-sample and
                out-of-sample
        """
        self.in_sample_pct = in_sample_pct
        self.num_folds = num_folds
        self.embargo_periods = embargo_periods

    def optimize(self,
                 strategy_class: type,
                 param_grid: Dict,
                 market_data: pd.DataFrame,
                 objective: str = 'sharpe') -> Dict:
        """
        Perform walk-forward optimization.

        Args:
```

```
32              strategy_class: Strategy class to optimize
33              param_grid: Dictionary of parameter ranges
34              market_data: Historical market data
35              objective: Optimization metric ('sharpe', 'return',
                    'calmar')
36
37          Returns:
38              Dictionary with optimization results
39          """
40          results = []
41          fold_size = len(market_data) // self.num_folds
42
43          for fold in range(self.num_folds):
44              fold_start = fold * fold_size
45              fold_end = min(fold_start + fold_size,
                    len(market_data))
46
47              # Calculate split point
48              in_sample_end = fold_start + int(
49                  (fold_end - fold_start) * self.in_sample_pct
50              )
51
52              # Apply embargo gap
53              out_sample_start = in_sample_end +
                    self.embargo_periods
54
55              # Split data
56              in_sample = market_data.iloc[fold_start:in_sample_end]
57              out_sample =
                    market_data.iloc[out_sample_start:fold_end]
58
59              if len(out_sample) < 20:  # Minimum validation period
60                  continue
61
62              # Grid search on in-sample
63              best_params, is_results = self._grid_search(
64                  strategy_class, param_grid, in_sample, objective
65              )
66
67              # Validate on out-of-sample
68              oos_results = self._validate(
69                  strategy_class, best_params, out_sample
70              )
71
72              results.append({
73                  'fold': fold,
74                  'best_params': best_params,
75                  'in_sample_sharpe': is_results['sharpe_ratio'],
76                  'in_sample_return': is_results['total_return'],
77                  'out_sample_sharpe': oos_results.sharpe_ratio,
78                  'out_sample_return': oos_results.total_return,
79                  'degradation': (
80                      is_results['sharpe_ratio'] -
                            oos_results.sharpe_ratio
81                  ) / is_results['sharpe_ratio'] if
                        is_results['sharpe_ratio'] > 0 else 0
82              })
83
```

```
84          return self._aggregate_results(results)
85
86      def _grid_search(self,
87                       strategy_class: type,
88                       param_grid: Dict,
89                       data: pd.DataFrame,
90                       objective: str) -> tuple:
91          """Exhaustive grid search for optimal parameters"""
92
93          from itertools import product
94
95          # Generate all parameter combinations
96          keys = param_grid.keys()
97          values = param_grid.values()
98          combinations = list(product(*values))
99
100         best_score = -np.inf
101         best_params = None
102         best_results = None
103
104         for combo in combinations:
105             params = dict(zip(keys, combo))
106
107             # Create and test strategy
108             strategy = strategy_class(
109                 universe=data['symbol'].unique().tolist(),
110                 params=params
111             )
112
113             backtester = EventDrivenBacktester(BacktestConfig(
114                 start_date=str(data['timestamp'].min()),
115                 end_date=str(data['timestamp'].max()),
116                 initial_capital=100000
117             ))
118
119             results = backtester.run(strategy, data)
120
121             # Get objective score
122             score = getattr(results, f'{objective}_ratio'
123                             if objective != 'return'
124                             else 'total_return')
125
126             if score > best_score:
127                 best_score = score
128                 best_params = params
129                 best_results = {
130                     'sharpe_ratio': results.sharpe_ratio,
131                     'total_return': results.total_return
132                 }
133
134         return best_params, best_results
135
136     def _aggregate_results(self, results: List[Dict]) -> Dict:
137         """Aggregate results across all folds"""
138
139         df = pd.DataFrame(results)
140
141         return {
```

```
142             'folds': results,
143             'avg_oos_sharpe': df['out_sample_sharpe'].mean(),
144             'std_oos_sharpe': df['out_sample_sharpe'].std(),
145             'avg_degradation': df['degradation'].mean(),
146             'consistent_params':
                    self._find_consistent_params(results),
147             'robustness_score': self._calculate_robustness(df)
148         }
149
150     def _calculate_robustness(self, df: pd.DataFrame) → float:
151         """
152         Calculate strategy robustness score.
153
154         A robust strategy should have:
155         - Positive OOS Sharpe across all folds
156         - Low degradation from IS to OOS
157         - Consistent performance
158         """
159         positive_folds = (df['out_sample_sharpe'] > 0).mean()
160         avg_sharpe = df['out_sample_sharpe'].mean()
161         consistency = 1 - df['out_sample_sharpe'].std() /
                abs(avg_sharpe) \
162                       if avg_sharpe != 0 else 0
163         low_degradation = (df['degradation'] < 0.5).mean()
164
165         return (positive_folds + consistency + low_degradation) /
                3
```

Chapter 7

Risk Management

7.1 Risk Management Philosophy

Risk management is the cornerstone of successful algorithmic trading. As Mark Douglas emphasizes in *Trading in the Zone*, maintaining discipline and controlling risk is more important than finding the perfect entry.

Key Risk Principles

1. Never risk more than 2% of capital on a single trade
2. Use position sizing to normalize risk across different volatility levels
3. Implement automated stops to remove emotional decision-making
4. Monitor correlation to avoid concentrated sector bets
5. Have circuit breakers for extreme drawdowns

7.2 Risk Manager Implementation

Listing 7.1: Comprehensive Risk Manager

```
from typing import Tuple, Dict, Optional
import numpy as np

class RiskManager:
    """
    Comprehensive risk management system.

    Implements principles from 'Trading in the Zone' (Douglas)
    regarding discipline and emotional control through systematic
    rule enforcement.
    """

    def __init__(self, config: Dict):
        """
        Initialize risk manager with configuration.

        Args:
            config: Dictionary with risk parameters
        """
        # Position-level limits
```

```
21          self.max_position_pct = config.get('max_position_pct',
                0.10)
22          self.max_risk_per_trade =
                config.get('max_risk_per_trade', 0.02)
23
24          # Portfolio-level limits
25          self.max_portfolio_risk =
                config.get('max_portfolio_risk', 0.06)
26          self.max_sector_exposure =
                config.get('max_sector_exposure', 0.30)
27          self.max_correlation = config.get('max_correlation', 0.70)
28          self.max_positions = config.get('max_positions', 20)
29
30          # Stop-loss settings
31          self.default_stop_pct = config.get('default_stop_pct',
                0.05)
32          self.trailing_stop_pct = config.get('trailing_stop_pct',
                0.08)
33
34          # Circuit breakers
35          self.daily_loss_limit = config.get('daily_loss_limit',
                0.03)
36          self.weekly_loss_limit = config.get('weekly_loss_limit',
                0.06)
37          self.max_drawdown_halt = config.get('max_drawdown_halt',
                0.15)
38
39          # State tracking
40          self.daily_pnl = 0.0
41          self.weekly_pnl = 0.0
42          self.is_halted = False
43          self.halt_reason: Optional[str] = None
44
45      def calculate_position_size(self,
46                                  signal: 'Signal',
47                                  portfolio_value: float,
48                                  current_price: float,
49                                  volatility: float,
50                                  stop_price: Optional[float] = None
51                                  ) -> int:
52          """
53          Calculate appropriate position size using multiple
                methods.
54
55          Implements:
56          1. Fixed fractional (risk per trade)
57          2. Volatility-adjusted sizing
58          3. Kelly Criterion (optional)
59
60          Returns the most conservative of all methods.
61          """
62          sizes = []
63
64          # Method 1: Fixed fractional based on stop loss
65          if stop_price:
66              risk_per_share = abs(current_price - stop_price)
67              dollar_risk = portfolio_value *
                    self.max_risk_per_trade
```

```
68              fixed_fractional_size = int(dollar_risk /
                    risk_per_share)
69              sizes.append(fixed_fractional_size)
70
71          # Method 2: Maximum position size limit
72          max_allocation = portfolio_value * self.max_position_pct
73          max_position_size = int(max_allocation / current_price)
74          sizes.append(max_position_size)
75
76          # Method 3: Volatility-adjusted sizing
77          target_vol = 0.20  # Target 20% annualized volatility
78          vol_scalar = target_vol / volatility if volatility > 0
                else 1.0
79          vol_allocation = portfolio_value * 0.10 * vol_scalar *
                signal.strength
80          vol_adjusted_size = int(vol_allocation / current_price)
81          sizes.append(vol_adjusted_size)
82
83          # Return most conservative size
84          return max(0, min(sizes))
85
86      def check_trade_allowed(self,
87                              signal: 'Signal',
88                              portfolio: 'Portfolio',
89                              position_size: int,
90                              current_price: float,
91                              sector_map: Dict[str, str]
92                              ) -> Tuple[bool, str]:
93          """
94          Comprehensive pre-trade risk checks.
95
96          Returns:
97              Tuple of (allowed: bool, reason: str)
98          """
99          # Check circuit breakers first
100         if self.is_halted:
101             return False, f"Trading halted: {self.halt_reason}"
102
103         # Check daily loss limit
104         if self.daily_pnl <= -self.daily_loss_limit *
                portfolio.total_value:
105             return False, "Daily loss limit reached"
106
107         # Check weekly loss limit
108         if self.weekly_pnl <= -self.weekly_loss_limit *
                portfolio.total_value:
109             return False, "Weekly loss limit reached"
110
111         # Check maximum drawdown
112         if portfolio.current_drawdown <= -self.max_drawdown_halt:
113             self._halt_trading("Maximum drawdown exceeded")
114             return False, "Maximum drawdown exceeded - trading
                    halted"
115
116         trade_value = position_size * current_price
117
118         # Check position size limit
```

```
119         if trade_value > portfolio.total_value *
                self.max_position_pct:
120             return False, f"Exceeds max position size
                    ({self.max_position_pct:.0%})"
121
122         # Check number of positions
123         if (len(portfolio.positions) >= self.max_positions and
124             signal.symbol not in portfolio.positions):
125             return False, f"Maximum positions
                    ({self.max_positions}) reached"
126
127         # Check sector exposure
128         sector = sector_map.get(signal.symbol, 'Unknown')
129         current_sector_exposure = portfolio.get_sector_exposure(
130             sector, sector_map
131         )
132         new_exposure = current_sector_exposure + trade_value
133         if new_exposure > portfolio.total_value *
                self.max_sector_exposure:
134             return False, f"Exceeds {sector} sector limit
                    ({self.max_sector_exposure:.0%})"
135
136         # Check correlation with existing positions
137         if self._check_high_correlation(signal.symbol, portfolio):
138             return False, "High correlation with existing
                    positions"
139
140         # All checks passed
141         return True, "Trade approved"
142
143     def _check_high_correlation(self,
144                                 symbol: str,
145                                 portfolio: 'Portfolio') -> bool:
146         """
147         Check if new position is highly correlated with existing.
148
149         Prevents concentration risk from correlated positions.
150         """
151         if len(portfolio.positions) == 0:
152             return False
153
154         # In production, this would use historical return
                correlations
155         # Simplified version checks sector overlap
156         return False
157
158     def generate_stop_orders(self,
159                              symbol: str,
160                              entry_price: float,
161                              side: str,
162                              volatility: float
163                              ) -> Dict[str, float]:
164         """
165         Generate stop-loss and take-profit levels.
166
167         Uses ATR-based stops for volatility adjustment.
168         """
169         # ATR-based stop (2x ATR)
```

```
170         atr_stop_distance = 2 * volatility * entry_price
171
172         # Percentage-based stop
173         pct_stop_distance = entry_price * self.default_stop_pct
174
175         # Use wider of the two
176         stop_distance = max(atr_stop_distance, pct_stop_distance)
177
178         if side == "BUY":
179             stop_loss = entry_price - stop_distance
180             take_profit = entry_price + (stop_distance * 2)  #
                2:1 R/R
181         else:
182             stop_loss = entry_price + stop_distance
183             take_profit = entry_price - (stop_distance * 2)
184
185         return {
186             'stop_loss': round(stop_loss, 2),
187             'take_profit': round(take_profit, 2),
188             'trailing_stop_pct': self.trailing_stop_pct
189         }
190
191     def update_daily_pnl(self, pnl: float):
192         """Update daily P&L tracking"""
193         self.daily_pnl = pnl
194
195     def update_weekly_pnl(self, pnl: float):
196         """Update weekly P&L tracking"""
197         self.weekly_pnl = pnl
198
199     def reset_daily(self):
200         """Reset daily counters (call at market open)"""
201         self.daily_pnl = 0.0
202
203     def reset_weekly(self):
204         """Reset weekly counters (call on Monday)"""
205         self.weekly_pnl = 0.0
206
207     def _halt_trading(self, reason: str):
208         """Halt all trading activity"""
209         self.is_halted = True
210         self.halt_reason = reason
211
212     def resume_trading(self):
213         """Resume trading (manual intervention required)"""
214         self.is_halted = False
215         self.halt_reason = None
```

7.3 Position Sizing Methods

Listing 7.2: Position Sizing Implementations

```
1 class PositionSizer:
2     """
3     Multiple position sizing methods from quantitative
4     trading literature.
```

```
    """

    @staticmethod
    def fixed_fractional(portfolio_value: float,
                         risk_pct: float,
                         entry_price: float,
                         stop_price: float) → int:
        """
        Risk a fixed percentage of portfolio per trade.

        This is the most common professional approach.

        Args:
            portfolio_value: Total portfolio value
            risk_pct: Percentage to risk (e.g., 0.02 for 2%)
            entry_price: Planned entry price
            stop_price: Stop-loss price

        Returns:
            Number of shares to trade
        """
        risk_per_share = abs(entry_price - stop_price)
        if risk_per_share == 0:
            return 0

        dollar_risk = portfolio_value * risk_pct
        shares = int(dollar_risk / risk_per_share)
        return shares

    @staticmethod
    def kelly_criterion(win_rate: float,
                        avg_win: float,
                        avg_loss: float) → float:
        """
        Kelly Criterion for optimal position sizing.

        f* = (bp - q) / b

        where:
            b = avg_win / avg_loss (payoff ratio)
            p = win_rate
            q = 1 - p (loss rate)

        Args:
            win_rate: Historical win rate (0-1)
            avg_win: Average winning trade size
            avg_loss: Average losing trade size (positive)

        Returns:
            Optimal fraction of capital to risk (use half-Kelly)
        """
        if avg_loss == 0 or win_rate <= 0:
            return 0

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

```

```
63          kelly = (b * p - q) / b
64
65          # Use half-Kelly for safety (reduces volatility)
66          half_kelly = max(0, kelly * 0.5)
67
68          # Cap at 25% maximum
69          return min(half_kelly, 0.25)
70
71      @staticmethod
72      def volatility_parity(portfolio_value: float,
73                            target_portfolio_vol: float,
74                            asset_volatility: float,
75                            current_price: float,
76                            num_positions: int) -> int:
77          """
78          Equal volatility contribution from each position.
79
80          Each position contributes equally to portfolio volatility.
81
82          Args:
83              portfolio_value: Total portfolio value
84              target_portfolio_vol: Target annual volatility (e.g.,
                    0.10)
85              asset_volatility: Asset's annualized volatility
86              current_price: Current asset price
87              num_positions: Expected number of positions
88          """
89          if asset_volatility == 0 or num_positions == 0:
90              return 0
91
92          # Target volatility per position
93          vol_per_position = target_portfolio_vol /
                np.sqrt(num_positions)
94
95          # Dollar allocation to achieve target vol
96          allocation = (portfolio_value * vol_per_position) /
                asset_volatility
97
98          shares = int(allocation / current_price)
99          return shares
100
101     @staticmethod
102     def equal_weight(portfolio_value: float,
103                      current_price: float,
104                      num_positions: int) -> int:
105         """
106         Simple equal-weight allocation.
107
108         Each position gets equal dollar allocation.
109         """
110         if num_positions == 0:
111             return 0
112
113         allocation = portfolio_value / num_positions
114         shares = int(allocation / current_price)
115         return shares
```

Chapter 8

Machine Learning Pipeline

8.1 ML for Trading Overview

Machine learning offers powerful tools for pattern recognition and prediction, but requires careful implementation to avoid common pitfalls. This chapter draws from Stefan Jansen's *Machine Learning for Algorithmic Trading* and Marcos López de Prado's *Advances in Financial Machine Learning*.

Important

ML Pitfalls in Finance:

- Standard cross-validation causes data leakage
- Financial data has low signal-to-noise ratio
- Overfitting is extremely easy with many features
- Transaction costs can eliminate predicted alpha

8.2 Feature Engineering for ML

Listing 8.1: ML Feature Engineering Pipeline

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from typing import Tuple, List

class MLFeaturePipeline:
    """
    Machine learning feature engineering pipeline.

    Implements best practices from 'Machine Learning for
    Algorithmic Trading' by Stefan Jansen.
    """

    def __init__(self):
        self.scaler = StandardScaler()
        self.feature_columns: List[str] = []

    def create_features(self, df: pd.DataFrame) -> pd.DataFrame:
        """
```

```
        Comprehensive feature engineering for ML models.

        Categories:
        1. Return features
        2. Volatility features
        3. Technical indicators
        4. Volume features
        5. Relative features
        """
        features = df.copy()

        # ========== Return Features ==========
        for period in [1, 5, 10, 20, 60]:
            features[f'return_{period}d'] = (
                features['close'].pct_change(period)
            )
            features[f'log_return_{period}d'] = (
                np.log(features['close'] /
                    features['close'].shift(period))
            )

        # ========== Volatility Features ==========
        for period in [10, 20, 60]:
            features[f'volatility_{period}d'] = (
                features['return_1d'].rolling(period).std()
            )
            features[f'volatility_ratio_{period}d'] = (
                features[f'volatility_{period}d'] /
                features[f'volatility_{period}d'].shift(period)
            )

        # Parkinson volatility (using high-low)
        features['parkinson_vol'] = np.sqrt(
            (1 / (4 * np.log(2))) *
            (np.log(features['high'] / features['low']) ** 2)
        ).rolling(20).mean()

        # ========== Technical Indicators ==========
        # RSI
        features['rsi_14'] =
            self._calculate_rsi(features['close'], 14)
        features['rsi_divergence'] = (
            features['rsi_14'] - features['rsi_14'].shift(5)
        )

        # MACD
        ema12 = features['close'].ewm(span=12).mean()
        ema26 = features['close'].ewm(span=26).mean()
        features['macd'] = ema12 - ema26
        features['macd_signal'] =
            features['macd'].ewm(span=9).mean()
        features['macd_hist'] = features['macd'] -
            features['macd_signal']

        # Moving average features
        for period in [10, 20, 50, 200]:
            ma = features['close'].rolling(period).mean()
```

```
            features[f'ma_ratio_{period}'] = features['close'] / ma
            features[f'ma_slope_{period}'] = ma.pct_change(5)

        # Bollinger Band position
        bb_middle = features['close'].rolling(20).mean()
        bb_std = features['close'].rolling(20).std()
        features['bb_position'] = (
            (features['close'] - bb_middle) / (2 * bb_std)
        )

        # ========== Volume Features ==========
        features['volume_ma_ratio'] = (
            features['volume'] /
                features['volume'].rolling(20).mean()
        )
        features['dollar_volume'] = features['close'] *
            features['volume']
        features['dollar_volume_ma'] = (
            features['dollar_volume'].rolling(20).mean()
        )

        # On-balance volume trend
        obv = (np.sign(features['close'].diff()) *
            features['volume']).cumsum()
        features['obv_slope'] = obv.pct_change(10)

        # ========== Price Pattern Features ==========
        features['high_low_range'] = (
            (features['high'] - features['low']) /
                features['close']
        )
        features['close_position'] = (
            (features['close'] - features['low']) /
            (features['high'] - features['low'])
        )

        # Gap features
        features['overnight_gap'] = (
            features['open'] / features['close'].shift(1) - 1
        )

        # ========== Relative Features ==========
        # These would use market/sector data in production
        features['relative_strength'] = (
            features['return_20d'] -
                features['return_20d'].rolling(60).mean()
        )

        # Store feature columns (excluding target and identifiers)
        exclude = ['symbol', 'timestamp', 'open', 'high', 'low',
                   'close', 'volume', 'adjusted_close']
        self.feature_columns = [
            col for col in features.columns if col not in exclude
        ]

        return features.dropna()

```

```
125     def _calculate_rsi(self, prices: pd.Series, period: int) →
            pd.Series:
126         """Calculate RSI indicator"""
127         delta = prices.diff()
128         gain = delta.where(delta > 0, 0).rolling(period).mean()
129         loss = (-delta.where(delta < 0, 0)).rolling(period).mean()
130         rs = gain / loss
131         return 100 - (100 / (1 + rs))
132 
133     def create_labels(self,
134                       df: pd.DataFrame,
135                       method: str = 'triple_barrier',
136                       **kwargs) → pd.Series:
137         """
138         Create labels for supervised learning.
139 
140         Methods:
141         - 'binary': Simple up/down classification
142         - 'triple_barrier': Lopez de Prado's method
143         - 'fixed_horizon': Return over fixed period
144         """
145         if method == 'binary':
146             return self._binary_labels(df, kwargs.get('horizon',
                    5))
147         elif method == 'triple_barrier':
148             return self._triple_barrier_labels(
149                 df,
150                 kwargs.get('horizon', 10),
151                 kwargs.get('upper', 0.02),
152                 kwargs.get('lower', 0.02)
153             )
154         elif method == 'fixed_horizon':
155             return self._fixed_horizon_labels(
156                 df,
157                 kwargs.get('horizon', 5),
158                 kwargs.get('threshold', 0.01)
159             )
160         else:
161             raise ValueError(f"Unknown labeling method: {method}")
162 
163     def _triple_barrier_labels(self,
164                                df: pd.DataFrame,
165                                horizon: int,
166                                upper_barrier: float,
167                                lower_barrier: float) → pd.Series:
168         """
169         Triple-barrier labeling from Lopez de Prado.
170 
171         Labels based on which barrier is touched first:
172         - Upper barrier (take profit): +1
173         - Lower barrier (stop loss): -1
174         - Vertical barrier (time): sign of return
175         """
176         labels = pd.Series(index=df.index, dtype=float)
177         prices = df['close'].values
178 
179         for i in range(len(df) - horizon):
180             entry_price = prices[i]
```

```
            upper = entry_price * (1 + upper_barrier)
            lower = entry_price * (1 - lower_barrier)

            # Check each future bar
            for j in range(1, horizon + 1):
                if i + j >= len(prices):
                    break

                future_price = prices[i + j]

                if future_price >= upper:
                    labels.iloc[i] = 1
                    break
                elif future_price <= lower:
                    labels.iloc[i] = -1
                    break
                elif j == horizon:
                    # Vertical barrier hit
                    ret = (future_price - entry_price) /
                        entry_price
                    labels.iloc[i] = np.sign(ret)

        return labels

    def fit_transform(self, X: pd.DataFrame) -> np.ndarray:
        """Fit scaler and transform features"""
        return self.scaler.fit_transform(X[self.feature_columns])

    def transform(self, X: pd.DataFrame) -> np.ndarray:
        """Transform features using fitted scaler"""
        return self.scaler.transform(X[self.feature_columns])
```

8.3 Purged Cross-Validation

Standard cross-validation causes data leakage in time series. Purged K-Fold addresses this.

Listing 8.2: Purged K-Fold Cross-Validation

```
class PurgedKFold:
    """
    Purged K-Fold cross-validation for time series.

    From 'Advances in Financial Machine Learning' by
    Marcos Lopez de Prado.

    Key features:
    1. Purging: Remove training samples overlapping with test
    2. Embargo: Add buffer after test period
    """

    def __init__(self,
                 n_splits: int = 5,
                 embargo_pct: float = 0.01,
                 purge_pct: float = 0.01):
        """
        Args:
```

```
            n_splits: Number of folds
            embargo_pct: Fraction of data to embargo after test
            purge_pct: Fraction to purge before test
        """
        self.n_splits = n_splits
        self.embargo_pct = embargo_pct
        self.purge_pct = purge_pct

    def split(self,
              X: pd.DataFrame,
              y: pd.Series = None,
              groups = None):
        """
        Generate train/test indices with purging and embargo.

        Yields:
            Tuple of (train_indices, test_indices)
        """
        n_samples = len(X)
        indices = np.arange(n_samples)

        fold_size = n_samples // self.n_splits
        embargo_size = int(n_samples * self.embargo_pct)
        purge_size = int(n_samples * self.purge_pct)

        for fold in range(self.n_splits):
            # Test set boundaries
            test_start = fold * fold_size
            test_end = (fold + 1) * fold_size

            if fold == self.n_splits - 1:
                test_end = n_samples

            # Purge: remove samples just before test
            purge_start = max(0, test_start - purge_size)

            # Embargo: skip samples just after test
            embargo_end = min(n_samples, test_end + embargo_size)

            # Training indices (everything except purge + test +
               embargo)
            train_indices = np.concatenate([
                indices[:purge_start],
                indices[embargo_end:]
            ])

            # Test indices
            test_indices = indices[test_start:test_end]

            yield train_indices, test_indices

    def get_n_splits(self, X=None, y=None, groups=None) -> int:
        return self.n_splits


class CombinatorialPurgedCV:
    """
    Combinatorial Purged Cross-Validation (CPCV).
```

```
76
77      Generates all possible train/test combinations while
78      maintaining temporal order and applying purging.
79      """
80
81      def __init__(self,
82                   n_splits: int = 6,
83                   n_test_splits: int = 2,
84                   embargo_pct: float = 0.01):
85          self.n_splits = n_splits
86          self.n_test_splits = n_test_splits
87          self.embargo_pct = embargo_pct
88
89      def split(self, X: pd.DataFrame):
90          """Generate CPCV train/test splits"""
91          from itertools import combinations
92
93          n_samples = len(X)
94          indices = np.arange(n_samples)
95          fold_size = n_samples // self.n_splits
96          embargo_size = int(n_samples * self.embargo_pct)
97
98          # Create fold boundaries
99          folds = []
100         for i in range(self.n_splits):
101             start = i * fold_size
102             end = (i + 1) * fold_size if i < self.n_splits - 1
                    else n_samples
103             folds.append((start, end))
104
105         # Generate all test combinations
106         for test_fold_indices in
                combinations(range(self.n_splits), self.n_test_splits):
107             test_indices = []
108             train_indices = []
109
110             for fold_idx, (start, end) in enumerate(folds):
111                 if fold_idx in test_fold_indices:
112                     test_indices.extend(indices[start:end])
113                 else:
114                     # Apply embargo around test folds
115                     fold_train = indices[start:end]
116
117                     # Check proximity to test folds
118                     for test_idx in test_fold_indices:
119                         test_start, test_end = folds[test_idx]
120
121                         if fold_idx == test_idx - 1:
122                             # Just before test - apply embargo
123                             fold_train =
                                    fold_train[:-embargo_size]
124                         elif fold_idx == test_idx + 1:
125                             # Just after test - apply embargo
126                             fold_train = fold_train[embargo_size:]
127
128                     train_indices.extend(fold_train)
129
130             yield np.array(train_indices), np.array(test_indices)
```

8.4 Model Training and Evaluation

Listing 8.3: ML Model Training Pipeline

```
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
import warnings
warnings.filterwarnings('ignore')

class MLTradingModel:
    """
    Machine learning model for trading signal generation.
    """

    def __init__(self, model_type: str = 'random_forest'):
        self.model_type = model_type
        self.model = None
        self.feature_importance: pd.Series = None

    def _create_model(self):
        """Create the underlying ML model"""
        if self.model_type == 'random_forest':
            return RandomForestClassifier(
                n_estimators=100,
                max_depth=5,
                min_samples_leaf=50,
                min_samples_split=100,
                class_weight='balanced',
                n_jobs=-1,
                random_state=42
            )
        elif self.model_type == 'gradient_boost':
            return GradientBoostingClassifier(
                n_estimators=100,
                max_depth=3,
                learning_rate=0.1,
                min_samples_leaf=50,
                subsample=0.8,
                random_state=42
            )
        else:
            raise ValueError(f"Unknown model type:
                {self.model_type}")

    def train_with_cv(self,
                      X: pd.DataFrame,
                      y: pd.Series,
                      feature_columns: List[str]) -> Dict:
        """
        Train model with purged cross-validation.

        Returns:
            Dictionary with CV results and metrics
        """
        self.model = self._create_model()
        cv = PurgedKFold(n_splits=5, embargo_pct=0.01)
```

```
52
53          results = {
54              'accuracy': [],
55              'precision': [],
56              'recall': [],
57              'f1': [],
58              'feature_importance': []
59          }
60
61          X_features = X[feature_columns].values
62          y_values = y.values
63
64          for fold, (train_idx, test_idx) in enumerate(cv.split(X)):
65              X_train = X_features[train_idx]
66              y_train = y_values[train_idx]
67              X_test = X_features[test_idx]
68              y_test = y_values[test_idx]
69
70              # Remove NaN samples
71              train_mask = ~(np.isnan(X_train).any(axis=1) |
                    np.isnan(y_train))
72              test_mask = ~(np.isnan(X_test).any(axis=1) |
                    np.isnan(y_test))
73
74              X_train = X_train[train_mask]
75              y_train = y_train[train_mask]
76              X_test = X_test[test_mask]
77              y_test = y_test[test_mask]
78
79              if len(X_train) < 100 or len(X_test) < 20:
80                  continue
81
82              # Train
83              self.model.fit(X_train, y_train)
84
85              # Predict
86              y_pred = self.model.predict(X_test)
87
88              # Metrics
89              results['accuracy'].append(accuracy_score(y_test,
                    y_pred))
90              results['precision'].append(
91                  precision_score(y_test, y_pred,
                        average='weighted', zero_division=0)
92              )
93              results['recall'].append(
94                  recall_score(y_test, y_pred, average='weighted',
                        zero_division=0)
95              )
96              results['f1'].append(
97                  f1_score(y_test, y_pred, average='weighted',
                        zero_division=0)
98              )
99
100             # Feature importance
101             if hasattr(self.model, 'feature_importances_'):
102                 results['feature_importance'].append(
103                     self.model.feature_importances_
```

```
                )

        # Aggregate results
        summary = {
            'mean_accuracy': np.mean(results['accuracy']),
            'std_accuracy': np.std(results['accuracy']),
            'mean_precision': np.mean(results['precision']),
            'mean_recall': np.mean(results['recall']),
            'mean_f1': np.mean(results['f1']),
        }

        # Average feature importance
        if results['feature_importance']:
            avg_importance =
               np.mean(results['feature_importance'], axis=0)
            self.feature_importance = pd.Series(
                avg_importance,
                index=feature_columns
            ).sort_values(ascending=False)
            summary['top_features'] =
               self.feature_importance.head(10).to_dict()

        # Final training on all data
        full_mask = ~(np.isnan(X_features).any(axis=1) |
           np.isnan(y_values))
        self.model.fit(X_features[full_mask], y_values[full_mask])

        return summary

    def predict(self, X: pd.DataFrame, feature_columns:
       List[str]) -> np.ndarray:
        """Generate predictions"""
        return self.model.predict(X[feature_columns].values)

    def predict_proba(self, X: pd.DataFrame, feature_columns:
       List[str]) -> np.ndarray:
        """Generate prediction probabilities"""
        return self.model.predict_proba(X[feature_columns].values)
```

Chapter 9

Live Trading Execution

9.1 Broker Integration

The system supports multiple brokers through a common interface. This section demonstrates Alpaca Markets integration.

Listing 9.1: Broker Interface and Alpaca Implementation

```python
from abc import ABC, abstractmethod
from typing import Dict, List, Optional
import alpaca_trade_api as tradeapi

class BrokerInterface(ABC):
    """Abstract interface for broker connections"""

    @abstractmethod
    def connect(self) -> bool:
        """Establish connection to broker"""
        pass

    @abstractmethod
    def submit_order(self, order: 'Order') -> str:
        """Submit order, return order ID"""
        pass

    @abstractmethod
    def cancel_order(self, order_id: str) -> bool:
        """Cancel an open order"""
        pass

    @abstractmethod
    def get_order_status(self, order_id: str) -> Dict:
        """Get current order status"""
        pass

    @abstractmethod
    def get_position(self, symbol: str) -> Optional[Dict]:
        """Get current position for symbol"""
        pass

    @abstractmethod
    def get_account(self) -> Dict:
```

```
35          """Get account information"""
36          pass
37
38      @abstractmethod
39      def get_positions(self) → List[Dict]:
40          """Get all current positions"""
41          pass
42
43
44  class AlpacaBroker(BrokerInterface):
45      """
46      Alpaca Markets API integration.
47
48      Supports both paper and live trading.
49      """
50
51      def __init__(self,
52                   api_key: str,
53                   secret_key: str,
54                   paper: bool = True):
55          """
56          Initialize Alpaca connection.
57
58          Args:
59              api_key: Alpaca API key
60              secret_key: Alpaca secret key
61              paper: Use paper trading if True
62          """
63          base_url = ('https://paper-api.alpaca.markets'
64                      if paper
65                      else 'https://api.alpaca.markets')
66
67          self.api = tradeapi.REST(
68              api_key,
69              secret_key,
70              base_url,
71              api_version='v2'
72          )
73          self.paper = paper
74
75      def connect(self) → bool:
76          """Verify connection by fetching account"""
77          try:
78              account = self.api.get_account()
79              return account.status == 'ACTIVE'
80          except Exception as e:
81              print(f"Connection failed: {e}")
82              return False
83
84      def submit_order(self, order: 'Order') → str:
85          """Submit order to Alpaca"""
86          try:
87              alpaca_order = self.api.submit_order(
88                  symbol=order.symbol,
89                  qty=order.quantity,
90                  side=order.side.lower(),
91                  type=order.order_type.lower(),
92                  time_in_force='day',
```

```
93                  limit_price=order.limit_price if order.order_type
                        == 'LIMIT' else None,
94                  stop_price=order.stop_price if 'STOP' in
                        order.order_type else None,
95                  client_order_id=order.client_id
96              )
97              return alpaca_order.id
98          except Exception as e:
99              raise OrderExecutionError(f"Order submission failed:
                    {e}")
100
101     def cancel_order(self, order_id: str) -> bool:
102         """Cancel an open order"""
103         try:
104             self.api.cancel_order(order_id)
105             return True
106         except Exception as e:
107             print(f"Cancel failed: {e}")
108             return False
109
110     def get_order_status(self, order_id: str) -> Dict:
111         """Get order status"""
112         order = self.api.get_order(order_id)
113         return {
114             'id': order.id,
115             'status': order.status,
116             'filled_qty': int(order.filled_qty or 0),
117             'filled_avg_price': float(order.filled_avg_price or
                    0),
118             'created_at': order.created_at,
119             'updated_at': order.updated_at
120         }
121
122     def get_position(self, symbol: str) -> Optional[Dict]:
123         """Get position for a symbol"""
124         try:
125             pos = self.api.get_position(symbol)
126             return {
127                 'symbol': pos.symbol,
128                 'qty': int(pos.qty),
129                 'side': 'long' if int(pos.qty) > 0 else 'short',
130                 'avg_entry_price': float(pos.avg_entry_price),
131                 'market_value': float(pos.market_value),
132                 'cost_basis': float(pos.cost_basis),
133                 'unrealized_pl': float(pos.unrealized_pl),
134                 'unrealized_plpc': float(pos.unrealized_plpc),
135                 'current_price': float(pos.current_price)
136             }
137         except Exception:
138             return None
139
140     def get_positions(self) -> List[Dict]:
141         """Get all positions"""
142         positions = self.api.list_positions()
143         return [
144             {
145                 'symbol': pos.symbol,
146                 'qty': int(pos.qty),
```

```
                'market_value': float(pos.market_value),
                'unrealized_pl': float(pos.unrealized_pl)
            }
            for pos in positions
        ]

    def get_account(self) -> Dict:
        """Get account information"""
        account = self.api.get_account()
        return {
            'equity': float(account.equity),
            'cash': float(account.cash),
            'buying_power': float(account.buying_power),
            'portfolio_value': float(account.portfolio_value),
            'day_trade_count': int(account.daytrade_count),
            'pattern_day_trader': account.pattern_day_trader
        }
```

9.2 Trading Engine

Listing 9.2: Main Trading Engine

```
import asyncio
import logging
from datetime import datetime, time
from typing import List

class TradingEngine:
    """
    Main trading engine orchestrating all components.

    Runs the main event loop for live trading.
    """

    def __init__(self,
                 strategies: List[BaseStrategy],
                 broker: BrokerInterface,
                 risk_manager: RiskManager,
                 data_feed: 'DataFeed',
                 config: Dict):
        """
        Initialize trading engine.

        Args:
            strategies: List of active strategies
            broker: Broker connection
            risk_manager: Risk management instance
            data_feed: Market data feed
            config: Engine configuration
        """
        self.strategies = strategies
        self.broker = broker
        self.risk_manager = risk_manager
        self.data_feed = data_feed
        self.config = config

```

```
35          self.is_running = False
36          self.portfolio = Portfolio()
37          self.pending_orders: Dict[str, 'Order'] = {}
38
39          self.logger = logging.getLogger('TradingEngine')
40          self._setup_logging()
41
42      def _setup_logging(self):
43          """Configure logging"""
44          handler = logging.FileHandler('trading.log')
45          handler.setFormatter(logging.Formatter(
46              '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
47          ))
48          self.logger.addHandler(handler)
49          self.logger.setLevel(logging.INFO)
50
51      async def run(self):
52          """Main trading loop"""
53          self.is_running = True
54          self.logger.info("Trading engine started")
55
56          # Connect to broker
57          if not self.broker.connect():
58              self.logger.error("Failed to connect to broker")
59              return
60
61          # Initialize portfolio from broker
62          await self._sync_portfolio()
63
64          while self.is_running:
65              try:
66                  # Check market hours
67                  if not self._is_market_open():
68                      await asyncio.sleep(60)
69                      continue
70
71                  # Get latest market data
72                  market_data = await self.data_feed.get_latest()
73
74                  # Generate signals from all strategies
75                  all_signals = []
76                  for strategy in self.strategies:
77                      if strategy.is_active:
78                          signals =
                               strategy.generate_signals(market_data)
79                          all_signals.extend(signals)
80
81                  # Prioritize and filter signals
82                  signals = self._prioritize_signals(all_signals)
83
84                  # Process each signal
85                  for signal in signals:
86                      await self._process_signal(signal,
                           market_data)
87
88                  # Manage existing positions
89                  await self._manage_positions(market_data)
90
```

```
91                  # Update risk manager
92                  self._update_risk_tracking()
93
94                  # Log status
95                  if datetime.now().minute % 15 == 0:
96                      self._log_status()
97
98                  await
                        asyncio.sleep(self.config.get('tick_interval',
                        1))
99
100             except Exception as e:
101                 self.logger.error(f"Error in trading loop: {e}")
102                 await asyncio.sleep(5)
103
104     async def _process_signal(self,
105                               signal: Signal,
106                               market_data: pd.DataFrame):
107         """Process a single trading signal"""
108
109         # Get current price
110         symbol_data = market_data[market_data['symbol'] ==
                signal.symbol]
111         if symbol_data.empty:
112             return
113
114         current_price = symbol_data['close'].iloc[-1]
115
116         # Calculate volatility for position sizing
117         volatility = self._calculate_volatility(signal.symbol,
                market_data)
118
119         # Get account value
120         account = self.broker.get_account()
121         portfolio_value = account['portfolio_value']
122
123         # Calculate position size
124         position_size = self.risk_manager.calculate_position_size(
125             signal=signal,
126             portfolio_value=portfolio_value,
127             current_price=current_price,
128             volatility=volatility
129         )
130
131         if position_size == 0:
132             return
133
134         # Risk check
135         allowed, reason = self.risk_manager.check_trade_allowed(
136             signal=signal,
137             portfolio=self.portfolio,
138             position_size=position_size,
139             current_price=current_price,
140             sector_map=self.config.get('sector_map', {})
141         )
142
143         if not allowed:
```

```
144             self.logger.warning(f"Trade rejected for
                    {signal.symbol}: {reason}")
145             return
146
147         # Create order
148         order = Order(
149             symbol=signal.symbol,
150             side='BUY' if signal.signal_type == SignalType.BUY
                    else 'SELL',
151             quantity=position_size,
152             order_type='LIMIT',
153             limit_price=self._calculate_limit_price(current_price,
                    signal)
154         )
155
156         # Submit order
157         try:
158             order_id = self.broker.submit_order(order)
159             self.pending_orders[order_id] = order
160             self.logger.info(
161                 f"Order submitted: {order_id} - {signal.symbol} "
162                 f"{order.side} {position_size} @
                        {order.limit_price}"
163             )
164         except Exception as e:
165             self.logger.error(f"Order submission failed: {e}")
166
167     def _is_market_open(self) -> bool:
168         """Check if market is currently open"""
169         now = datetime.now()
170
171         # Skip weekends
172         if now.weekday() >= 5:
173             return False
174
175         # Check market hours (9:30 AM - 4:00 PM ET)
176         market_open = time(9, 30)
177         market_close = time(16, 0)
178
179         return market_open <= now.time() <= market_close
180
181     def _prioritize_signals(self, signals: List[Signal]) ->
            List[Signal]:
182         """
183         Prioritize and deduplicate signals.
184
185         Higher strength signals get priority. Only one signal
186         per symbol is allowed.
187         """
188         # Sort by strength descending
189         sorted_signals = sorted(signals, key=lambda s:
                s.strength, reverse=True)
190
191         # Deduplicate by symbol
192         seen_symbols = set()
193         unique_signals = []
194
195         for signal in sorted_signals:
```

```
196                 if signal.symbol not in seen_symbols:
197                     unique_signals.append(signal)
198                     seen_symbols.add(signal.symbol)
199 
200         return unique_signals
201 
202     def stop(self):
203         """Stop the trading engine"""
204         self.is_running = False
205         self.logger.info("Trading engine stopped")
```

Chapter 10

Monitoring and Analytics

10.1 Performance Metrics

Table 10.1: Key Performance Indicators (KPIs)

Metric	Formula	Interpretation
Sharpe Ratio	$\frac{R_p - R_f}{\sigma_p}$	Risk-adjusted return; > 1 is good, > 2 is excellent
Sortino Ratio	$\frac{R_p - R_f}{\sigma_d}$	Like Sharpe but only penalizes downside volatility
Max Drawdown	$\max\left(\frac{P_{peak} - P_{trough}}{P_{peak}}\right)$	Largest peak-to-trough decline
Calmar Ratio	$\frac{R_{annual}}{\lvert MaxDD \rvert}$	Return per unit of drawdown risk
Win Rate	$\frac{N_{wins}}{N_{total}}$	Percentage of profitable trades
Profit Factor	$\frac{\sum wins}{\sum losses}$	Ratio of gross profit to gross loss; > 1.5 is good

10.2 Alerting System

Listing 10.1: Alert Manager Implementation

```python
from dataclasses import dataclass
from typing import Callable, List
from enum import Enum

class AlertSeverity(Enum):
    INFO = "info"
    WARNING = "warning"
    CRITICAL = "critical"

@dataclass
class AlertRule:
    """Definition of an alert rule"""
    name: str
    condition: Callable[['Portfolio', pd.DataFrame], bool]
    severity: AlertSeverity
    message_template: str
    cooldown_minutes: int = 30

```

```python
class AlertManager:
    """
    Real-time alerting system for trading operations.
    """

    def __init__(self, notification_channels: List[str]):
        """
        Args:
            notification_channels: List of channels ('email',
                'slack', 'sms')
        """
        self.channels = notification_channels
        self.rules: List[AlertRule] = []
        self.last_triggered: Dict[str, datetime] = {}
        self._setup_default_rules()

    def _setup_default_rules(self):
        """Configure default alert rules"""

        # Drawdown alert
        self.add_rule(AlertRule(
            name="drawdown_warning",
            condition=lambda p, d: p.current_drawdown < -0.05,
            severity=AlertSeverity.WARNING,
            message_template="Portfolio drawdown at
                {drawdown:.2%}",
            cooldown_minutes=60
        ))

        # Critical drawdown
        self.add_rule(AlertRule(
            name="drawdown_critical",
            condition=lambda p, d: p.current_drawdown < -0.10,
            severity=AlertSeverity.CRITICAL,
            message_template="CRITICAL: Drawdown at
                {drawdown:.2%}. Consider halting.",
            cooldown_minutes=30
        ))

        # Daily loss limit
        self.add_rule(AlertRule(
            name="daily_loss",
            condition=lambda p, d: p.daily_pnl < -p.total_value *
                0.02,
            severity=AlertSeverity.WARNING,
            message_template="Daily loss exceeds 2%:
                {daily_pnl:.2%}",
            cooldown_minutes=60
        ))

        # High volatility
        self.add_rule(AlertRule(
            name="volatility_spike",
            condition=self._check_volatility_spike,
            severity=AlertSeverity.INFO,
            message_template="Market volatility spike detected",
            cooldown_minutes=120
        ))
```

```
72
73      def _check_volatility_spike(self,
74                                  portfolio: 'Portfolio',
75                                  market_data: pd.DataFrame) -> bool:
76          """Check for unusual market volatility"""
77          if 'VIX' in market_data['symbol'].values:
78              vix = market_data[market_data['symbol'] ==
                    'VIX']['close'].iloc[-1]
79              return vix > 25
80          return False
81
82      def add_rule(self, rule: AlertRule):
83          """Add a new alert rule"""
84          self.rules.append(rule)
85
86      def check_alerts(self,
87                       portfolio: 'Portfolio',
88                       market_data: pd.DataFrame):
89          """Check all alert conditions"""
90
91          for rule in self.rules:
92              # Check cooldown
93              if rule.name in self.last_triggered:
94                  elapsed = datetime.now() -
                        self.last_triggered[rule.name]
95                  if elapsed.total_seconds() <
                        rule.cooldown_minutes * 60:
96                      continue
97
98              # Evaluate condition
99              if rule.condition(portfolio, market_data):
100                 self._trigger_alert(rule, portfolio)
101
102     def _trigger_alert(self, rule: AlertRule, portfolio:
            'Portfolio'):
103         """Trigger an alert"""
104         message = rule.message_template.format(
105             drawdown=portfolio.current_drawdown,
106             daily_pnl=portfolio.daily_pnl / portfolio.total_value,
107             equity=portfolio.total_value
108         )
109
110         full_message = f"[{rule.severity.value.upper()}]
                {rule.name}: {message}"
111
112         for channel in self.channels:
113             self._send_notification(channel, full_message,
                    rule.severity)
114
115         self.last_triggered[rule.name] = datetime.now()
116
117     def _send_notification(self,
118                            channel: str,
119                            message: str,
120                            severity: AlertSeverity):
121         """Send notification through specified channel"""
122         if channel == 'slack':
123             self._send_slack(message, severity)
```

```
124             elif channel == 'email':
125                 self._send_email(message, severity)
126             elif channel == 'sms':
127                 self._send_sms(message, severity)
128 
129     def _send_slack(self, message: str, severity: AlertSeverity):
130         """Send Slack notification"""
131         # Implementation would use Slack webhook
132         pass
133 
134     def _send_email(self, message: str, severity: AlertSeverity):
135         """Send email notification"""
136         # Implementation would use SMTP or email service
137         pass
138 
139     def _send_sms(self, message: str, severity: AlertSeverity):
140         """Send SMS notification"""
141         # Implementation would use Twilio or similar
142         pass
```

Chapter 11

Deployment Architecture

11.1 Containerized Deployment

The system is designed for containerized deployment using Docker and Kubernetes.

Listing 11.1: Docker Compose Configuration

```yaml
version: '3.8'

services:
  trading-engine:
    build:
      context: .
      dockerfile: Dockerfile.trading
    environment:
      - ALPACA_API_KEY=${ALPACA_API_KEY}
      - ALPACA_SECRET_KEY=${ALPACA_SECRET_KEY}
      - DATABASE_URL=postgresql://postgres:password@timescaledb:5432/trading
      - REDIS_URL=redis://redis:6379
    depends_on:
      - timescaledb
      - redis
    restart: unless-stopped
    volumes:
      - ./logs:/app/logs

  backtester:
    build:
      context: .
      dockerfile: Dockerfile.backtester
    environment:
      - DATABASE_URL=postgresql://postgres:password@timescaledb:5432/trading
    volumes:
      - ./strategies:/app/strategies
      - ./data:/app/data

  data-collector:
    build:
      context: .
      dockerfile: Dockerfile.collector
```

```
    environment:
      - POLYGON_API_KEY=${POLYGON_API_KEY}
      -
          DATABASE_URL=postgresql://postgres:password@timescaledb:5432/trading
    depends_on:
      - timescaledb
    restart: unless-stopped

  dashboard:
    build:
      context: ./dashboard
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      - API_URL=http://trading-engine:8000

  timescaledb:
    image: timescale/timescaledb:latest-pg14
    environment:
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=trading
    volumes:
      - timescale-data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  redis:
    image: redis:7-alpine
    volumes:
      - redis-data:/data
    ports:
      - "6379:6379"

  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana
    ports:
      - "3001:3000"
    volumes:
      - grafana-data:/var/lib/grafana

volumes:
  timescale-data:
  redis-data:
  grafana-data:
```

11.2 CI/CD Pipeline

Listing 11.2: GitHub Actions CI/CD Pipeline

```yaml
name: Trading System CI/CD

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest pytest-cov

      - name: Run unit tests
        run: pytest tests/unit --cov=src --cov-report=xml

      - name: Upload coverage
        uses: codecov/codecov-action@v3

  backtest-validation:
    runs-on: ubuntu-latest
    needs: test
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run strategy validation
        run: python scripts/validate_strategies.py --min-sharpe
           0.5

      - name: Archive results
        uses: actions/upload-artifact@v3
        with:
          name: backtest-results
          path: results/

  deploy-staging:
    runs-on: ubuntu-latest
    needs: [test, backtest-validation]
```

```
    if: github.ref == 'refs/heads/develop'
    steps:
      - uses: actions/checkout@v4

      - name: Deploy to staging
        run: |
          kubectl config set-cluster staging
          kubectl apply -f k8s/staging/

  deploy-production:
    runs-on: ubuntu-latest
    needs: [test, backtest-validation]
    if: github.ref == 'refs/heads/main'
    environment: production
    steps:
      - uses: actions/checkout@v4

      - name: Deploy to production
        run: |
          kubectl config set-cluster production
          kubectl apply -f k8s/production/
```

Chapter 12

Development Roadmap

12.1 Implementation Timeline

Table 12.1: Development Phases and Milestones

Phase	Duration	Deliverables
Foundation	Weeks 1-2	Project setup, data models, database schema
Data Layer	Weeks 3-4	Data connectors, feature engineering pipeline
Backtesting	Weeks 5-6	Backtester engine, performance metrics
Strategies	Weeks 7-10	Implement 3-4 base strategies (CAN SLIM, Mean Reversion, Momentum)
Risk Management	Weeks 11-12	Position sizing, risk checks, stop-losses
Paper Trading	Weeks 13-14	Broker integration, simulated execution
ML Pipeline	Weeks 15-18	Feature engineering, model training, integration
Dashboard	Weeks 19-20	Monitoring UI, alerts, reporting
Live Trading	Weeks 21-24	Production deployment, gradual capital deployment

12.2 Success Criteria

Table 12.2: Success Criteria by Phase

Criterion	Target
Backtesting Validation	Sharpe > 1.0 OOS, Profit Factor > 1.5
Risk Discipline	Max Drawdown $< 15\%$, Daily VaR $< 2\%$
Execution Quality	Slippage < 10 bps, Fill Rate $> 95\%$
System Reliability	Uptime $> 99.9\%$ during market hours
Robustness	Positive returns in $> 70\%$ of walk-forward folds

12.3 Risk Considerations

Important

Key Risks to Monitor:

1. **Overfitting**: Strategies may perform well in backtest but fail live
2. **Regime Change**: Market conditions may invalidate strategy assumptions
3. **Execution Risk**: Live execution differs from backtest assumptions
4. **Technology Risk**: System failures during critical market moments
5. **Regulatory Risk**: Changes in trading rules or requirements

Appendix A

Reference Materials

A.1 Recommended Reading

A.1.1 Fundamental Analysis

- Graham, Benjamin. *The Intelligent Investor*
- Kratter, Matthew R. *A Beginner's Guide to the Stock Market*

A.1.2 Technical Analysis

- Murphy, John J. *Technical Analysis of the Financial Markets*
- O'Neil, William J. *How to Make Money in Stocks*
- Pring, Martin. *Technical Analysis Explained*

A.1.3 Trading Psychology

- Douglas, Mark. *Trading in the Zone*
- Schwager, Jack D. *Market Wizards*

A.1.4 Quantitative and Algorithmic Trading

- Chan, Ernie. *Algorithmic Trading: Winning Strategies and Their Rationale*
- Chan, Ernie. *Quantitative Trading*
- Davey, Kevin J. *Building Winning Algorithmic Trading Systems*
- Johnson, Barry. *Algorithmic Trading and DMA*

A.1.5 Machine Learning for Finance

- Jansen, Stefan. *Machine Learning for Algorithmic Trading*
- López de Prado, Marcos. *Advances in Financial Machine Learning*
- Hilpisch, Yves. *Python for Finance*

A.1.6 High-Frequency and Advanced Topics

- Aldridge, Irene. *High-Frequency Trading: A Practical Guide*
- Kissell, Robert. *The Science of Algorithmic Trading and Portfolio Management*

Appendix B

Glossary

Alpha
Excess return above benchmark performance

ATR
Average True Range; volatility indicator

Backtest
Historical simulation of trading strategy

Basis Point (bp)
1/100th of 1% (0.01%)

Calmar Ratio
Annualized return divided by maximum drawdown

Drawdown
Peak-to-trough decline in portfolio value

Kelly Criterion
Optimal position sizing formula

MACD
Moving Average Convergence Divergence indicator

OHLCV
Open, High, Low, Close, Volume data

RSI Relative Strength Index; momentum oscillator

Sharpe Ratio
Risk-adjusted return measure

Slippage
Difference between expected and actual execution price

Sortino Ratio
Like Sharpe but only penalizes downside volatility

TWAP
Time-Weighted Average Price execution algorithm

VaR
Value at Risk; maximum expected loss

VWAP
: Volume-Weighted Average Price execution algorithm