

# Claude Code Master Query - Trading System

## AIALgoTradeHits.com Complete Trading & Execution Guide

**Version:** 5.0

**Last Updated:** January 2026

**Location:** [Trading/masterquery.md](#)

---

## TABLE OF CONTENTS

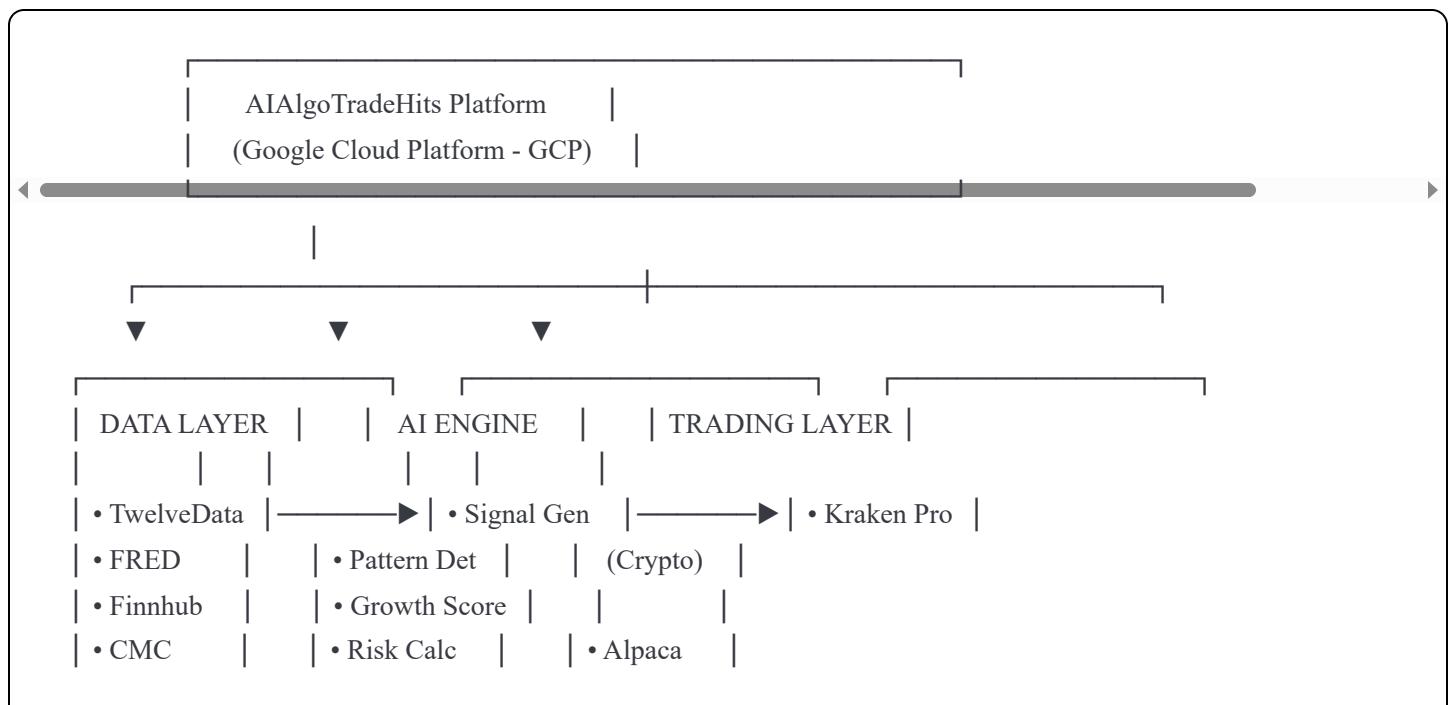
1. [Executive Summary](#)
  2. [Platform Architecture Overview](#)
  3. [Broker/Exchange Selection Guide](#)
  4. [Kraken Pro - Crypto Trading](#)
  5. [Alpaca - Stocks & ETFs Trading](#)
  6. [AI Signal Generation Engine](#)
  7. [Unified Trading Engine](#)
  8. [Risk Management System](#)
  9. [Fault Tolerant Execution](#)
  10. [API Capacity Maximization](#)
  11. [Configuration Management](#)
  12. [Deployment & Monitoring](#)
- 

## EXECUTIVE SUMMARY

|                                              |                                                           |
|----------------------------------------------|-----------------------------------------------------------|
| AIALGOTRADEHITS - AUTOMATED TRADING PLATFORM |                                                           |
| <input type="checkbox"/> CRYPTO TRADING      | <input checked="" type="checkbox"/> STOCKS & ETFs TRADING |
| Platform: KRAKEN PRO                         | Platform: ALPACA (Recommended)                            |

|                                                                |                                   |  |
|----------------------------------------------------------------|-----------------------------------|--|
| Assets: 200+ crypto pairs                                      | Assets: 10,000+ stocks, 500+ ETFs |  |
| Execution: Spot + Futures                                      | Execution: Commission-free        |  |
| API: REST + WebSocket                                          | API: REST + WebSocket             |  |
| Cost: Trading fees only                                        | Cost: FREE (no commissions)       |  |
| <br><b>⚠ KRAKEN DOES NOT SUPPORT STOCKS/ETFs</b>               |                                   |  |
| <b>ALPACA SUPPORTS:</b>                                        |                                   |  |
| - US Stocks                                                    |                                   |  |
| - US ETFs                                                      |                                   |  |
| - Crypto (limited)                                             |                                   |  |
| - Paper trading for testing                                    |                                   |  |
| <br><b>AI-POWERED SIGNAL GENERATION</b>                        |                                   |  |
| Data Source: TwelveData Pro (\$229/mo) - 1.15M credits/day     |                                   |  |
| AI Engine: Google Vertex AI (Gemini 2.5 Pro)                   |                                   |  |
| Indicators: 112 technical indicators                           |                                   |  |
| Signals: EMA crossovers, RSI, MACD, Growth Score, Trend Regime |                                   |  |
| Target Accuracy: 66-72% (from current 53% baseline)            |                                   |  |

## PLATFORM ARCHITECTURE OVERVIEW



• Kraken WS | • Gemini 2.5 | (Stocks/ETF)

BigQuery Data Warehouse  
 • Trade History • Performance  
 • Signals • Risk Metrics

## BROKER/EXCHANGE SELECTION GUIDE

### 🏆 RECOMMENDED SETUP FOR MAXIMUM AI-DRIVEN GAINS

| Asset Class | Recommended Platform | Why                                              | API Cost                 |
|-------------|----------------------|--------------------------------------------------|--------------------------|
| Crypto      | Kraken Pro           | Best liquidity, futures support, deep order book | Free (trading fees only) |
| US Stocks   | Alpaca               | Commission-free, excellent API, paper trading    | FREE                     |
| US ETFs     | Alpaca               | Commission-free, fractional shares               | FREE                     |
| Options     | Interactive Brokers  | Most comprehensive options API                   | \$0-10/mo                |
| Forex       | OANDA or Alpaca      | Tight spreads, good API                          | Free                     |

### Platform Comparison Matrix

| Feature     | Kraken     | Alpaca   | Coinbase | IBKR       | Tradier   |  |
|-------------|------------|----------|----------|------------|-----------|--|
|             | Pro        | Advanced |          |            |           |  |
| Crypto      | ✓ 200+     | ✓ 30+    | ✓ 100+   | ✓ Limited  | ✗         |  |
| US Stocks   | ✗          | ✓ 10K+   | ✗        | ✓ Global   | ✓         |  |
| ETFs        | ✗          | ✓ 500+   | ✗        | ✓ Global   | ✓         |  |
| Options     | ✗          | ✗        | ✗        | ✓ Best     | ✓         |  |
| Futures     | ✓ Crypto   | ✗        | ✗        | ✓ All      | ✗         |  |
| Commission  | 0.16-0.26% | FREE     | 0.5-1.5% | \$0.005/sh | \$0/trade |  |
| API Quality | ★★★★★      | ★★★★★    | ★★★★★    | ★★★★★      | ★★★★★     |  |

|                |      |           |      |      |           |      |      |     |
|----------------|------|-----------|------|------|-----------|------|------|-----|
| Paper Trading  |      |           | Yes  |      |           | Yes  |      |     |
| WebSocket      |      | Yes       |      | Yes  |           | Yes  |      | Yes |
| AI Integration | Good | Excellent | Good | Good | Excellent | Good | Good |     |

## KRAKEN PRO - CRYPTO AUTOMATED TRADING

### Overview

Kraken Pro is your primary platform for automated cryptocurrency trading. It supports spot trading, margin trading, and futures for 200+ crypto pairs.

### Account Setup

```
yaml
KRAKEN_SETUP:
  account_type: "Pro"
  verification_level: "Intermediate or Pro" # Required for API trading
```

```
api_key_permissions:
  - Query Funds
  - Query Open Orders & Trades
  - Query Closed Orders & Trades
  - Modify Orders
  - Cancel/Close Orders
  - Create & Modify Orders
# DO NOT enable: Withdraw Funds (security risk)
```

```
security:
  - Enable 2FA on account
  - IP whitelist for API keys
  - Separate API keys for production vs testing
```

### Kraken Trading Client (Full Implementation)

```
python
```

```
"""
Kraken Pro Automated Trading Client
Full fault-tolerant implementation for crypto trading
"""


```

```
import os
import time
import hmac
import hashlib
import base64
import urllib.parse
import asyncio
import aiohttp
import json
from typing import Dict, List, Optional, Tuple
from dataclasses import dataclass
from enum import Enum
from datetime import datetime
import logging
```

```
logger = logging.getLogger(__name__)
```

```
class OrderType(Enum):
    MARKET = "market"
    LIMIT = "limit"
    STOP_LOSS = "stop-loss"
    TAKE_PROFIT = "take-profit"
    STOP_LOSS_LIMIT = "stop-loss-limit"
    TAKE_PROFIT_LIMIT = "take-profit-limit"
    TRAILING_STOP = "trailing-stop"
    TRAILING_STOP_LIMIT = "trailing-stop-limit"
    SETTLE_POSITION = "settle-position"
```

```
class OrderSide(Enum):
    BUY = "buy"
    SELL = "sell"
```

```
@dataclass
class TradeSignal:
    """AI-generated trade signal"""
    symbol: str
    side: OrderSide
    confidence: float # 0.0 to 1.0
    entry_price: float
```

```

take_profit: float
stop_loss: float
position_size: float
timeframe: str
indicators: Dict
generated_at: datetime

@dataclass
class OrderResult:
    """Result of order execution"""
    success: bool
    order_id: Optional[str]
    txid: Optional[str]
    error: Optional[str]
    executed_price: Optional[float]
    executed_volume: Optional[float]
    fees: Optional[float]

class KrakenTradingClient:
    """
    Production-ready Kraken trading client with full fault tolerance.

    Features:
    - Automatic order placement with TP/SL
    - Position management
    - Risk controls
    - Nonce management for API auth
    - Circuit breaker pattern
    - Exponential backoff retry
    """

    REST_URL = "https://api.kraken.com"
    WS_URL = "wss://ws.kraken.com/"
    WS_AUTH_URL = "wss://ws-auth.kraken.com/"

    def __init__(self):
        self.api_key = os.getenv('KRAKEN_API_KEY')
        self.api_secret = os.getenv('KRAKEN_API_SECRET')
        self.last_nonce = 0
        self.nonce_lock = asyncio.Lock()

        # Rate limiting
        self.api_counter = 0
        self.counter_max = int(os.getenv('KRAKEN_COUNTER_MAX', 20))

```

```

self.counter_decay = float(os.getenv('KRAKEN_COUNTER_DECAY', 1.0))
self.last_counter_update = time.time()

# Circuit breaker
self.failure_count = 0
self.circuit_open = False
self.last_failure_time = None

# Position tracking
self.open_positions: Dict[str, Dict] = {}
self.pending_orders: Dict[str, Dict] = {}

# Risk settings
self.max_position_size_usd = float(os.getenv('MAX_POSITION_SIZE_USD', 1000))
self.max_daily_loss_usd = float(os.getenv('MAX_DAILY_LOSS_USD', 500))
self.max_concurrent_positions = int(os.getenv('MAX_CONCURRENT_POSITIONS', 5))
self.daily_pnl = 0.0

async def _get_nonce(self) -> str:
    """Generate strictly increasing nonce"""
    async with self.nonce_lock:
        current_nonce = int(time.time() * 1000)
        if current_nonce <= self.last_nonce:
            current_nonce = self.last_nonce + 1
        self.last_nonce = current_nonce
    return str(current_nonce)

def _sign_request(self, uri_path: str, data: Dict) -> str:
    """Generate Kraken API signature"""
    postdata = urllib.parse.urlencode(data)
    encoded = (str(data['nonce']) + postdata).encode()
    message = uri_path.encode() + hashlib.sha256(encoded).digest()

    mac = hmac.new(
        base64.b64decode(self.api_secret),
        message,
        hashlib.sha512
    )
    return base64.b64encode(mac.digest()).decode()

async def _check_rate_limit(self) -> bool:
    """Check and update rate limit counter"""
    current_time = time.time()
    elapsed = current_time - self.last_counter_update

```

```
    self.api_counter = max(0, self.api_counter - (elapsed * self.counter_decay))
    self.last_counter_update = current_time
```

```
if self.api_counter >= self.counter_max:
    return False
return True
```

```
async def _private_request(self, endpoint: str, data: Dict = None,
                           cost: int = 1) -> Dict:
```

```
    """
```

Make authenticated request to Kraken private API.

Includes full fault tolerance.

```
    """
```

```
# Check circuit breaker
```

```
if self.circuit_open:
    if time.time() - self.last_failure_time < 60:
        raise Exception("Circuit breaker open - trading suspended")
    self.circuit_open = False
    self.failure_count = 0
```

```
# Check rate limit
```

```
while not await self._check_rate_limit():
    await asyncio.sleep(1)
```

```
data = data or {}
data['nonce'] = await self._get_nonce()
```

```
uri_path = f"/0/private/{endpoint}"
signature = self._sign_request(uri_path, data)
```

```
headers = {
    'API-Key': self.api_key,
    'API-Sign': signature,
    'Content-Type': 'application/x-www-form-urlencoded'
}
```

```
max_retries = 3
for attempt in range(max_retries):
    try:
        async with aiohttp.ClientSession() as session:
            async with session.post(
                f'{self.REST_URL}{uri_path}',
                headers=headers,
                data=urlencode.parse.urlencode(data),
```

```
    timeout=aiohttp.ClientTimeout(total=30)

) as response:
    result = await response.json()

    if result.get('error'):
        error = result['error'][0] if result['error'] else 'Unknown'

        # Handle specific errors
        if 'EAPI:Invalid nonce' in error:
            self.last_nonce = int(time.time() * 1000) + 10000
            continue

        if 'EAPI:Rate limit exceeded' in error:
            await asyncio.sleep(5)
            continue

        if 'EOrder:Insufficient funds' in error:
            raise Exception(f"Insufficient funds: {error}")

        raise Exception(f"Kraken API error: {error}")

    self.api_counter += cost
    self.failure_count = 0
    return result.get('result', {})

except Exception as e:
    self.failure_count += 1
    if self.failure_count >= 5:
        self.circuit_open = True
        self.last_failure_time = time.time()
        logger.error(f"Circuit breaker opened due to failures: {e}")

    if attempt == max_retries - 1:
        raise

    await asyncio.sleep(2 ** attempt)

raise Exception("Max retries exceeded")

# =====
# TRADING OPERATIONS
# =====

async def place_order(
```

```
self,  
pair: str,  
side: OrderSide,  
order_type: OrderType,  
volume: float,  
price: float = None,  
leverage: int = None,  
take_profit: float = None,  
stop_loss: float = None,  
reduce_only: bool = False,  
validate_only: bool = False
```

```
) -> OrderResult:
```

```
"""
```

Place an order on Kraken.

Args:

```
pair: Trading pair (e.g., 'XBTUSD', 'ETHUSD')  
side: 'buy' or 'sell'  
order_type: Order type (market, limit, etc.)  
volume: Order volume in base currency  
price: Limit price (required for limit orders)  
leverage: Leverage amount (2-5x typically)  
take_profit: Take profit price  
stop_loss: Stop loss price  
reduce_only: Only reduce existing position  
validate_only: Validate without placing
```

```
"""
```

# Pre-trade risk checks

```
if not await self._pre_trade_risk_check(pair, volume, side):
```

```
    return OrderResult(
```

```
        success=False,  
        order_id=None,  
        txid=None,  
        error="Failed risk checks",  
        executed_price=None,  
        executed_volume=None,  
        fees=None
```

```
)
```

```
data = {  
    'pair': pair,  
    'type': side.value,  
    'ordertype': order_type.value,  
    'volume': str(volume),
```

```
}

if price and order_type != OrderType.MARKET:
    data['price'] = str(price)

if leverage:
    data['leverage'] = str(leverage)

# Add close order for TP/SL
if take_profit or stop_loss:
    close_orders = []
    if take_profit:
        close_orders.append(f"limit:{take_profit}")
    if stop_loss:
        close_orders.append(f"stop-loss:{stop_loss}")
    data['close'] = ','.join(close_orders)

if reduce_only:
    data['reduce_only'] = 'true'

if validate_only:
    data['validate'] = 'true'

try:
    result = await self._private_request('AddOrder', data)

    order_id = result.get('txid', [None])[0]

    logger.info(f"Order placed: {side.value} {volume} {pair} @ {price or 'market'}")

    return OrderResult(
        success=True,
        order_id=order_id,
        txid=order_id,
        error=None,
        executed_price=price,
        executed_volume=volume,
        fees=None # Will be updated when order fills
    )

except Exception as e:
    logger.error(f"Order failed: {e}")
    return OrderResult(
        success=False,
```

```

        order_id=None,
        txid=None,
        error=str(e),
        executed_price=None,
        executed_volume=None,
        fees=None
    )

async def place_bracket_order(
    self,
    pair: str,
    side: OrderSide,
    volume: float,
    entry_price: float = None,
    take_profit_pct: float = 2.0,
    stop_loss_pct: float = 1.0,
    leverage: int = None
) -> Tuple[OrderResult, Optional[str], Optional[str]]:
    """
    Place a bracket order with entry, take-profit, and stop-loss.

```

Returns:

Tuple of (entry\_result, tp\_order\_id, sl\_order\_id)

"""

```

# Calculate TP/SL prices
if entry_price:
    if side == OrderSide.BUY:
        tp_price = entry_price * (1 + take_profit_pct / 100)
        sl_price = entry_price * (1 - stop_loss_pct / 100)
    else:
        tp_price = entry_price * (1 - take_profit_pct / 100)
        sl_price = entry_price * (1 + stop_loss_pct / 100)
else:
    # For market orders, we'll set TP/SL after entry fills
    tp_price = None
    sl_price = None

```

# Place entry order

```
order_type = OrderType.LIMIT if entry_price else OrderType.MARKET
```

```

entry_result = await self.place_order(
    pair=pair,
    side=side,
    order_type=order_type,
```

```

volume=volume,
price=entry_price,
leverage=leverage,
take_profit=tp_price,
stop_loss=sl_price
)

return entry_result, None, None # Kraken handles TP/SL with close param

async def cancel_order(self, txid: str) -> bool:
    """Cancel an open order"""
    try:
        result = await self._private_request('CancelOrder', {'txid': txid})
        logger.info(f"Order cancelled: {txid}")
        return True
    except Exception as e:
        logger.error(f"Failed to cancel order {txid}: {e}")
        return False

async def cancel_all_orders(self) -> int:
    """Cancel all open orders"""
    try:
        result = await self._private_request('CancelAll')
        count = result.get('count', 0)
        logger.info(f"Cancelled {count} orders")
        return count
    except Exception as e:
        logger.error(f"Failed to cancel all orders: {e}")
        return 0

# =====
# ACCOUNT & POSITION MANAGEMENT
# =====

async def get_balance(self) -> Dict[str, float]:
    """Get account balances"""
    result = await self._private_request('Balance')
    return {k: float(v) for k, v in result.items()}

async def get_trade_balance(self, asset: str = 'ZUSD') -> Dict:
    """Get trading balance including margin info"""
    result = await self._private_request('TradeBalance', {'asset': asset})
    return result

```

```

async def get_open_orders(self) -> Dict:
    """Get all open orders"""
    result = await self._private_request('OpenOrders')
    return result.get('open', {})

async def get_open_positions(self) -> Dict:
    """Get all open positions"""
    result = await self._private_request('OpenPositions')
    self.open_positions = result
    return result

async def get_trades_history(self, start: int = None, end: int = None) -> Dict:
    """Get trade history"""
    data = {}
    if start:
        data['start'] = start
    if end:
        data['end'] = end

    result = await self._private_request('TradesHistory', data, cost=2)
    return result.get('trades', {})

```

```

# =====
# RISK MANAGEMENT
# =====

```

```

async def _pre_trade_risk_check(self, pair: str, volume: float,
                                 side: OrderSide) -> bool:
    """
    Perform pre-trade risk checks.
    Returns True if trade is allowed.
    """

    # Check daily loss limit
    if self.daily_pnl < -self.max_daily_loss_usd:
        logger.warning(f'Daily loss limit reached: ${self.daily_pnl}')
        return False

    # Check max concurrent positions
    positions = await self.get_open_positions()
    if len(positions) >= self.max_concurrent_positions:
        logger.warning(f'Max concurrent positions reached: {len(positions)}')
        return False

    # Check position size

```

```

ticker = await self._get_ticker(pair)

if ticker:
    price = float(ticker.get('c', [0])[0])
    position_value_usd = volume * price

    if position_value_usd > self.max_position_size_usd:
        logger.warning(f"Position size ${position_value_usd} exceeds max ${self.max_position_size_usd}")
        return False

return True

async def _get_ticker(self, pair: str) -> Optional[Dict]:
    """Get current ticker for a pair"""
    async with aiohttp.ClientSession() as session:
        async with session.get(
            f'{self.REST_URL}/0/public/Ticker',
            params={'pair': pair}
        ) as response:
            result = await response.json()
            if result.get('result'):
                # Return first result
                return list(result['result'].values())[0]

    return None

async def emergency_close_all(self) -> List[OrderResult]:
    """
    EMERGENCY: Close all positions at market price.
    Use only in emergency situations.
    """

    logger.warning("EMERGENCY CLOSE ALL POSITIONS")

    results = []

    # Cancel all open orders first
    await self.cancel_all_orders()

    # Close all positions
    positions = await self.get_open_positions()

    for pos_id, pos_data in positions.items():
        pair = pos_data.get('pair')
        volume = float(pos_data.get('vol', 0))
        pos_type = pos_data.get('type') # 'buy' or 'sell'

```

```

# Close by placing opposite order
close_side = OrderSide.SELL if pos_type == 'buy' else OrderSide.BUY

result = await self.place_order(
    pair=pair,
    side=close_side,
    order_type=OrderType.MARKET,
    volume=volume,
    reduce_only=True
)
results.append(result)

return results

```

```

# Global Kraken client instance
kraken_trader = KrakenTradingClient()

```

## ALPACA - STOCKS & ETFs AUTOMATED TRADING

### Why Alpaca for Stocks & ETFs

#### WHY ALPACA IS RECOMMENDED

- COMMISSION-FREE trading on stocks and ETFs
- EXCELLENT API designed for algorithmic trading
- PAPER TRADING environment for testing strategies
- REAL-TIME data included (no additional cost)
- FRACTIONAL SHARES supported
- NO MINIMUM balance requirement
- INSTANT settlement for day trading
- WebSocket streaming for real-time updates
- Well-documented Python SDK
- Crypto trading also available (30+ pairs)

## Alpaca Account Setup

```
yaml

ALPACA_SETUP:
  account_type: "Trading"
  website: "https://alpaca.markets"

account_types:
  paper:
    base_url: "https://paper-api.alpaca.markets"
    purpose: "Testing strategies with fake money"

  live:
    base_url: "https://api.alpaca.markets"
    purpose: "Real money trading"

api_keys:
  # Get from: https://app.alpaca.markets/brokerage/api-management
  - APCA_API_KEY_ID: "Your API Key ID"
  - APCA_API_SECRET_KEY: "Your Secret Key"

data_plans:
  free: "IEX data (15-min delayed)"
  sip: "$99/mo for real-time consolidated data"
```

## Alpaca Trading Client (Full Implementation)

```
python
```

....  
Alpaca Automated Trading Client  
For US Stocks and ETFs with full fault tolerance  
....

```
import os
import asyncio
import aiohttp
from typing import Dict, List, Optional, Tuple
from dataclasses import dataclass
from enum import Enum
from datetime import datetime, timedelta
import logging
```

```
logger = logging.getLogger(__name__)
```

```
class AlpacaOrderType(Enum):
    MARKET = "market"
    LIMIT = "limit"
    STOP = "stop"
    STOP_LIMIT = "stop_limit"
    TRAILING_STOP = "trailing_stop"
```

```
class AlpacaOrderSide(Enum):
    BUY = "buy"
    SELL = "sell"
```

```
class AlpacaTimeInForce(Enum):
    DAY = "day"
    GTC = "gtc" # Good til cancelled
    OPG = "opg" # Market on open
    CLS = "cls" # Market on close
    IOC = "ioc" # Immediate or cancel
    FOK = "fok" # Fill or kill
```

```
@dataclass
class AlpacaOrderResult:
    success: bool
    order_id: Optional[str]
    client_order_id: Optional[str]
    error: Optional[str]
    status: Optional[str]
    filled_qty: Optional[float]
    filled_avg_price: Optional[float]
```

```
class AlpacaTradingClient:
```

```
"""
```

Production-ready Alpaca trading client for stocks and ETFs.

Features:

- Commission-free stock/ETF trading
- Paper trading support
- Bracket orders with TP/SL
- Position management
- Real-time streaming

```
"""
```

```
def __init__(self, paper: bool = True):
```

```
    self.api_key = os.getenv('ALPACA_API_KEY_ID')
```

```
    self.api_secret = os.getenv('ALPACA_API_SECRET_KEY')
```

*# Use paper or live endpoint*

```
    if paper:
```

```
        self.base_url = "https://paper-api.alpaca.markets"
```

```
    else:
```

```
        self.base_url = os.getenv('ALPACA_BASE_URL', 'https://api.alpaca.markets')
```

```
    self.data_url = "https://data.alpaca.markets"
```

*# Rate limiting*

```
    self.requests_per_minute = 200
```

```
    self.request_count = 0
```

```
    self.last_reset = datetime.now()
```

*# Circuit breaker*

```
    self.failure_count = 0
```

```
    self.circuit_open = False
```

*# Risk settings*

```
    self.max_position_size_usd = float(os.getenv('MAX_STOCK_POSITION_USD', 5000))
```

```
    self.max_portfolio_risk_pct = float(os.getenv('MAX_PORTFOLIO_RISK_PCT', 2.0))
```

```
    self.max_daily_trades = int(os.getenv('MAX_DAILY_TRADES', 50))
```

```
    self.daily_trade_count = 0
```

```
def _get_headers(self) -> Dict:
```

```
    """Get authentication headers"""
    return {
```

```
        'APCA-API-KEY-ID': self.api_key,
```

```
'APCA-API-SECRET-KEY': self.api_secret,
'Content-Type': 'application/json'
}

async def _check_rate_limit(self):
    """Check and handle rate limiting"""
    now = datetime.now()
    if (now - self.last_reset).seconds >= 60:
        self.request_count = 0
        self.last_reset = now

    if self.request_count >= self.requests_per_minute:
        sleep_time = 60 - (now - self.last_reset).seconds
        await asyncio.sleep(sleep_time)
        self.request_count = 0
        self.last_reset = datetime.now()

    self.request_count += 1

async def _request(self, method: str, endpoint: str,
                  data: Dict = None) -> Dict:
    """Make authenticated request to Alpaca"""
    if self.circuit_open:
        raise Exception("Circuit breaker open - trading suspended")

    await self._check_rate_limit()

    url = f'{self.base_url}{endpoint}'

    max_retries = 3
    for attempt in range(max_retries):
        try:
            async with aiohttp.ClientSession() as session:
                if method == 'GET':
                    async with session.get(
                        url,
                        headers=self._get_headers(),
                        timeout=aiohttp.ClientTimeout(total=30)
                    ) as response:
                        if response.status == 429:
                            await asyncio.sleep(60)
                            continue
                        response.raise_for_status()
                return await response.json()
```

```
elif method == 'POST':
    async with session.post(
        url,
        headers=self._get_headers(),
        json=data,
        timeout=aiohttp.ClientTimeout(total=30)
    ) as response:
        if response.status == 429:
            await asyncio.sleep(60)
            continue

        if response.status >= 400:
            error_data = await response.json()
            raise Exception(f"Alpaca error: {error_data}")

    return await response.json()

elif method == 'DELETE':
    async with session.delete(
        url,
        headers=self._get_headers(),
        timeout=aiohttp.ClientTimeout(total=30)
    ) as response:
        if response.status == 204:
            return {'success': True}
        return await response.json()

except Exception as e:
    self.failure_count += 1
    if self.failure_count >= 5:
        self.circuit_open = True
        logger.error(f"Circuit breaker opened: {e}")

    if attempt == max_retries - 1:
        raise

    await asyncio.sleep(2 ** attempt)

raise Exception("Max retries exceeded")

# =====
# ACCOUNT OPERATIONS
# =====
```

```

async def get_account(self) -> Dict:
    """Get account information"""
    return await self._request('GET', '/v2/account')

async def get_positions(self) -> List[Dict]:
    """Get all open positions"""
    return await self._request('GET', '/v2/positions')

async def get_position(self, symbol: str) -> Dict:
    """Get position for specific symbol"""
    return await self._request('GET', f'/v2/positions/{symbol}')

async def close_position(self, symbol: str, qty: float = None,
                        percentage: float = None) -> Dict:
    """Close a position"""
    params = {}
    if qty:
        params['qty'] = str(qty)
    if percentage:
        params['percentage'] = str(percentage)

    endpoint = f'/v2/positions/{symbol}'
    if params:
        endpoint += '?' + '&'.join(f'{k}={v}' for k, v in params.items())

    return await self._request('DELETE', endpoint)

async def close_all_positions(self) -> Dict:
    """Close all positions"""
    return await self._request('DELETE', '/v2/positions')

# =====
# ORDER OPERATIONS
# =====

async def place_order(
    self,
    symbol: str,
    side: AlpacaOrderSide,
    order_type: AlpacaOrderType,
    qty: float = None,
    notional: float = None, # Dollar amount for fractional shares
    limit_price: float = None,

```

```
stop_price: float = None,  
time_in_force: AlpacaTimeInForce = AlpacaTimeInForce.DAY,  
take_profit: float = None,  
stop_loss: float = None,  
trail_percent: float = None,  
trail_price: float = None,  
extended_hours: bool = False
```

) -> AlpacaOrderResult:

"""\n

Place an order on Alpaca.

Args:

```
symbol: Stock ticker (e.g., 'AAPL', 'SPY')  
side: 'buy' or 'sell'  
order_type: market, limit, stop, stop_limit, trailing_stop  
qty: Number of shares (can be fractional)  
notional: Dollar amount (for fractional shares)  
limit_price: Limit price  
stop_price: Stop price  
time_in_force: Order duration  
take_profit: Take profit limit price (for bracket orders)  
stop_loss: Stop loss price (for bracket orders)  
trail_percent: Trailing stop percentage  
trail_price: Trailing stop price offset  
extended_hours: Allow extended hours trading
```

"""\n

# Risk checks

```
if not await self._pre_trade_risk_check(symbol, qty, notional, side):
```

```
    return AlpacaOrderResult(
```

```
        success=False,  
        order_id=None,  
        client_order_id=None,  
        error="Failed risk checks",  
        status=None,  
        filled_qty=None,  
        filled_avg_price=None
```

```
)
```

```
data = {  
    'symbol': symbol,  
    'side': side.value,  
    'type': order_type.value,  
    'time_in_force': time_in_force.value,  
}
```

```
if qty:
    data['qty'] = str(qty)
elif notional:
    data['notional'] = str(notional)

if limit_price:
    data['limit_price'] = str(limit_price)

if stop_price:
    data['stop_price'] = str(stop_price)

if trail_percent:
    data['trail_percent'] = str(trail_percent)

if trail_price:
    data['trail_price'] = str(trail_price)

if extended_hours:
    data['extended_hours'] = True

# Bracket order (entry with TP and SL)
if take_profit or stop_loss:
    data['order_class'] = 'bracket'

    if take_profit:
        data['take_profit'] = {'limit_price': str(take_profit)}

    if stop_loss:
        data['stop_loss'] = {'stop_price': str(stop_loss)}

try:
    result = await self._request('POST', '/v2/orders', data)

    self.daily_trade_count += 1

    logger.info(f"Order placed: {side.value} {qty or notional} {symbol}")

return AlpacaOrderResult(
    success=True,
    order_id=result.get('id'),
    client_order_id=result.get('client_order_id'),
    error=None,
    status=result.get('status'),
```

```
        filled_qty=float(result.get('filled_qty', 0)),
        filled_avg_price=float(result.get('filled_avg_price', 0)) if result.get('filled_avg_price') else None
    )

except Exception as e:
    logger.error(f"Order failed: {e}")
    return AlpacaOrderResult(
        success=False,
        order_id=None,
        client_order_id=None,
        error=str(e),
        status=None,
        filled_qty=None,
        filled_avg_price=None
    )
```

```
async def place_bracket_order(
```

```
    self,
    symbol: str,
    side: AlpacaOrderSide,
    qty: float,
    entry_price: float = None,
    take_profit_pct: float = 2.0,
    stop_loss_pct: float = 1.0
) -> AlpacaOrderResult:
```

```
    """
```

Place a bracket order with entry, take-profit, and stop-loss.

```
    """
```

```
# Get current price for market orders
```

```
if not entry_price:
    quote = await self.get_latest_quote(symbol)
    entry_price = float(quote.get('ap', 0)) or float(quote.get('bp', 0))
```

```
if side == AlpacaOrderSide.BUY:
```

```
    tp_price = round(entry_price * (1 + take_profit_pct / 100), 2)
    sl_price = round(entry_price * (1 - stop_loss_pct / 100), 2)
```

```
else:
```

```
    tp_price = round(entry_price * (1 - take_profit_pct / 100), 2)
    sl_price = round(entry_price * (1 + stop_loss_pct / 100), 2)
```

```
order_type = AlpacaOrderType.LIMIT if entry_price else AlpacaOrderType.MARKET
```

```
return await self.place_order(
    symbol=symbol,
```

```

        side=side,
        order_type=order_type,
        qty=qty,
        limit_price=entry_price if order_type == AlpacaOrderType.LIMIT else None,
        take_profit=tp_price,
        stop_loss=sl_price
    )
}

async def get_orders(self, status: str = 'open') -> List[Dict]:
    """Get orders by status"""
    return await self._request('GET', f'/v2/orders?status={status}')

async def get_order(self, order_id: str) -> Dict:
    """Get specific order by ID"""
    return await self._request('GET', f'/v2/orders/{order_id}')

async def cancel_order(self, order_id: str) -> bool:
    """Cancel an order"""
    try:
        await self._request('DELETE', f'/v2/orders/{order_id}')
        return True
    except:
        return False

async def cancel_all_orders(self) -> Dict:
    """Cancel all open orders"""
    return await self._request('DELETE', '/v2/orders')

# =====
# MARKET DATA
# =====

async def get_latest_quote(self, symbol: str) -> Dict:
    """Get latest quote for symbol"""
    async with aiohttp.ClientSession() as session:
        async with session.get(
            f'{self.data_url}/v2/stocks/{symbol}/quotes/latest',
            headers=self._get_headers()
        ) as response:
            data = await response.json()
            return data.get('quote', {})

async def get_latest_bar(self, symbol: str) -> Dict:
    """Get latest bar for symbol"""

```

```

async with aiohttp.ClientSession() as session:
    async with session.get(
        f'{self.data_url}/v2/stocks/{symbol}/bars/latest',
        headers=self._get_headers()
    ) as response:
        data = await response.json()
        return data.get('bar', {})

# =====
# RISK MANAGEMENT
# =====

async def _pre_trade_risk_check(self, symbol: str, qty: float,
                                 notional: float, side: AlpacaOrderSide) -> bool:
    """Pre-trade risk checks"""

    # Check daily trade limit
    if self.daily_trade_count >= self.max_daily_trades:
        logger.warning(f'Daily trade limit reached: {self.daily_trade_count}')
        return False

    # Calculate position value
    if notional:
        position_value = notional
    elif qty:
        quote = await self.get_latest_quote(symbol)
        price = float(quote.get('ap', 0)) or float(quote.get('bp', 0))
        position_value = qty * price
    else:
        return False

    # Check position size
    if position_value > self.max_position_size_usd:
        logger.warning(f'Position size ${position_value} exceeds max ${self.max_position_size_usd}')
        return False

    # Check buying power
    account = await self.get_account()
    buying_power = float(account.get('Buying Power', 0))

    if side == AlpacaOrderSide.BUY and position_value > buying_power:
        logger.warning(f'Insufficient buying power: ${buying_power}')
        return False

```

```
return True

async def emergency_liquidate(self) -> Dict:
    """
    EMERGENCY: Liquidate all positions.
    """
    logger.warning("EMERGENCY LIQUIDATION")

    # Cancel all orders first
    await self.cancel_all_orders()

    # Close all positions
    return await self.close_all_positions()

# Global Alpaca client instances
alpaca_paper = AlpacaTradingClient(paper=True)
alpaca_live = AlpacaTradingClient(paper=False)
```

## AI SIGNAL GENERATION ENGINE

### Overview

The AI Signal Generation Engine analyzes market data from TwelveData and generates trading signals with confidence scores for automated execution.

```
python
```

AI Signal Generation Engine

Uses TwelveData indicators + Gemini AI for signal generation

```
import os
import asyncio
from typing import Dict, List, Optional
from dataclasses import dataclass
from datetime import datetime
from enum import Enum
import vertexai
from vertexai.generative_models import GenerativeModel
from google.cloud import bigquery
```

```
class SignalType(Enum):
```

```
    STRONG_BUY = "strong_buy"
    BUY = "buy"
    HOLD = "hold"
    SELL = "sell"
    STRONG_SELL = "strong_sell"
```

```
@dataclass
```

```
class TradingSignal:
```

```
    symbol: str
    asset_type: str # 'crypto', 'stock', 'etf'
    signal_type: SignalType
    confidence: float # 0.0 to 1.0
    entry_price: float
    take_profit: float
    stop_loss: float
    position_size_pct: float
    timeframe: str
    reasoning: str
    indicators: Dict
    generated_at: datetime
```

```
class AISignalGenerator:
```

```
....
```

Generates trading signals using technical indicators and AI analysis.

Signal Generation Process:

1. Fetch latest indicators from BigQuery
2. Apply rule-based filters (EMA crossover, RSI, etc.)

3. Score candidates with Growth Score algorithm
  4. Use Gemini AI for pattern recognition and sentiment
  5. Generate signals with confidence scores
- """

```

def __init__(self):
    self.bq_client = bigquery.Client()
    self.project_id = os.getenv('GCP_PROJECT_ID')
    self.dataset = os.getenv('BIGQUERY_DATASET')

    # Initialize Vertex AI
    vertexai.init(project=self.project_id, location='us-central1')
    self.model = GenerativeModel(os.getenv('VERTEX_AI_MODEL', 'gemini-2.5-pro'))

    # Signal thresholds
    self.min_confidence = float(os.getenv('MIN_SIGNAL_CONFIDENCE', 0.65))
    self.min_growth_score = float(os.getenv('MIN_GROWTH_SCORE', 60))

async def generate_crypto_signals(self, top_n: int = 10) -> List[TradingSignal]:
    """Generate signals for top crypto opportunities"""

    # Query for crypto candidates
    query = f"""
    WITH latest_data AS (
        SELECT
            symbol,
            close,
            ema_12,
            ema_26,
            rsi_14,
            macd,
            macd_signal,
            macd_histogram,
            adx,
            volume,
            growth_score,
            trend_regime,
            in_rise_cycle,
            -- Calculate signal strength
            CASE
                WHEN ema_12 > ema_26 AND rsi_14 > 50 AND macd > macd_signal THEN 'bullish'
                WHEN ema_12 < ema_26 AND rsi_14 < 50 AND macd < macd_signal THEN 'bearish'
                ELSE 'neutral'
            END as signal_direction,
    
```

```
    ROW_NUMBER() OVER (PARTITION BY symbol ORDER BY datetime DESC) as rn
  FROM `self.project_id`.{self.dataset}.crypto_analysis`
  WHERE DATE(datetime) >= DATE_SUB(CURRENT_DATE(), INTERVAL 1 DAY)
)
SELECT *
FROM latest_data
WHERE rn = 1
  AND growth_score >= {self.min_growth_score}
  AND signal_direction = 'bullish'
  AND in_rise_cycle = TRUE
ORDER BY growth_score DESC
LIMIT {top_n}
"""

```

```
results = self.bq_client.query(query).result()
```

```
signals = []
for row in results:
    signal = await self._generate_signal_for_row(row, 'crypto')
    if signal and signal.confidence >= self.min_confidence:
        signals.append(signal)

return signals
```

```
async def generate_stock_signals(self, top_n: int = 10) -> List[TradingSignal]:
    """Generate signals for top stock opportunities"""

```

```
query = f"""
WITH latest_data AS (
  SELECT
    symbol,
    company_name,
    sector,
    close,
    ema_12,
    ema_26,
    rsi_14,
    macd,
    macd_signal,
    adx,
    volume,
    growth_score,
    trend_regime,
    CASE
```

```

WHEN ema_12 > ema_26 AND rsi_14 BETWEEN 40 AND 70 AND macd > macd_signal THEN 'bullish'
WHEN ema_12 < ema_26 AND rsi_14 BETWEEN 30 AND 60 AND macd < macd_signal THEN 'bearish'
ELSE 'neutral'

END as signal_direction,
ROW_NUMBER() OVER (PARTITION BY symbol ORDER BY datetime DESC) as rn
FROM `self.project_id`.self.dataset.stock_analysis`
WHERE DATE(datetime) >= DATE_SUB(CURRENT_DATE(), INTERVAL 1 DAY)
)

SELECT *
FROM latest_data
WHERE rn = 1
AND growth_score >= {self.min_growth_score}
AND signal_direction = 'bullish'
AND adx > 20 -- Trending market
ORDER BY growth_score DESC
LIMIT {top_n}
"""

```

```
results = self.bq_client.query(query).result()
```

```

signals = []
for row in results:
    signal = await self._generate_signal_for_row(row, 'stock')
    if signal and signal.confidence >= self.min_confidence:
        signals.append(signal)

return signals

```

```
async def _generate_signal_for_row(self, row, asset_type: str) -> Optional[TradingSignal]:
    """Generate a trading signal for a single asset"""

```

```

indicators = {
    'ema_12': row.ema_12,
    'ema_26': row.ema_26,
    'rsi_14': row.rsi_14,
    'macd': row.macd,
    'macd_signal': row.macd_signal,
    'adx': row.adx,
    'growth_score': row.growth_score,
    'trend_regime': row.trend_regime,
}

```

```
# Calculate confidence based on indicator alignment
confidence = self._calculate_confidence(indicators)
```

```

if confidence < self.min_confidence:
    return None

# Determine signal type
if confidence >= 0.8:
    signal_type = SignalType.STRONG_BUY
elif confidence >= 0.65:
    signal_type = SignalType.BUY
else:
    signal_type = SignalType.HOLD

# Calculate TP/SL based on ATR or fixed percentage
entry_price = row.close
atr_pct = 2.0 # Default 2% for TP/SL

take_profit = entry_price * (1 + atr_pct / 100)
stop_loss = entry_price * (1 - (atr_pct * 0.5) / 100)

# Position size based on confidence
position_size_pct = min(5.0, confidence * 5.0) # Max 5% per position

# Get AI reasoning
reasoning = await self._get_ai_reasoning(row.symbol, indicators, asset_type)

return TradingSignal(
    symbol=row.symbol,
    asset_type=asset_type,
    signal_type=signal_type,
    confidence=confidence,
    entry_price=entry_price,
    take_profit=take_profit,
    stop_loss=stop_loss,
    position_size_pct=position_size_pct,
    timeframe='daily',
    reasoning=reasoning,
    indicators=indicators,
    generated_at=datetime.utcnow()
)

def _calculate_confidence(self, indicators: Dict) -> float:
    """Calculate confidence score based on indicator alignment"""
    score = 0.0
    max_score = 0.0

```

```

# EMA alignment (25% weight)
max_score += 25
if indicators['ema_12'] > indicators['ema_26']:
    score += 25

# RSI in optimal range (20% weight)
max_score += 20
rsi = indicators['rsi_14']
if 50 <= rsi <= 70:
    score += 20
elif 40 <= rsi < 50 or 70 < rsi <= 80:
    score += 10

# MACD above signal (20% weight)
max_score += 20
if indicators['macd'] > indicators['macd_signal']:
    score += 20

# ADX showing trend (15% weight)
max_score += 15
adx = indicators['adx']
if adx > 25:
    score += 15
elif adx > 20:
    score += 10

# Growth score (20% weight)
max_score += 20
growth_score = indicators['growth_score']
if growth_score >= 80:
    score += 20
elif growth_score >= 60:
    score += 15
elif growth_score >= 40:
    score += 10

return score / max_score

async def _get_ai_reasoning(self, symbol: str, indicators: Dict,
                             asset_type: str) -> str:
    """Get AI-generated reasoning for the signal"""

    prompt = f"""

```

Analyze this {asset\_type} trading opportunity for {symbol}:

Technical Indicators:

- EMA 12/26: {indicators['ema\_12']:.2f} / {indicators['ema\_26']:.2f}
- RSI 14: {indicators['rsi\_14']:.2f}
- MACD: {indicators['macd']:.4f} (Signal: {indicators['macd\_signal']:.4f})
- ADX: {indicators['adx']:.2f}
- Growth Score: {indicators['growth\_score']:.1f}
- Trend Regime: {indicators['trend\_regime']}

Provide a brief (2-3 sentences) analysis explaining why this is a good trading opportunity.

Focus on the technical setup and potential risks.

.....

try:

```
    response = self.model.generate_content(prompt)
    return response.text.strip()
except Exception as e:
    return f"Technical analysis indicates bullish momentum with aligned indicators."
```

# Global signal generator

```
signal_generator = AISignalGenerator()
```

## UNIFIED TRADING ENGINE

### Automated Trading Orchestrator

python

Unified Trading Engine

Coordinates AI signals with execution across Kraken and Alpaca

```
import asyncio
from typing import Dict, List
from datetime import datetime, time
import pytz
from enum import Enum
```

```
class TradingEngine:
```

Main trading engine that:

1. Receives signals from AI Signal Generator
2. Validates against risk parameters
3. Routes to appropriate broker (Kraken for crypto, Alpaca for stocks)
4. Manages positions and monitors P&L
5. Handles emergency situations

```
def __init__(self):
```

```
    self.kraken = kraken_trader
    self.alpaca = alpaca_live # Use alpaca_paper for testing
    self.signal_gen = signal_generator
```

```
    # Trading state
```

```
    self.is_running = False
    self.positions: Dict[str, Dict] = {}
    self.daily_pnl = 0.0
    self.trade_history: List[Dict] = []
```

```
    # Configuration
```

```
    self.max_daily_loss = float(os.getenv('MAX_DAILY_LOSS_USD', 500))
    self.max_positions = int(os.getenv('MAX_TOTAL_POSITIONS', 10))
    self.trading_hours_only = os.getenv('TRADING_HOURS_ONLY', 'true').lower() == 'true'
```

```
    # Market hours (Eastern Time)
```

```
    self.market_open = time(9, 30)
    self.market_close = time(16, 0)
    self.eastern = pytz.timezone('America/New_York')
```

```
async def start(self):
```

```
    """Start the trading engine"""
```

```
self.is_running = True
logger.info("Trading engine started")

while self.is_running:
    try:
        await self._trading_loop()
    except Exception as e:
        logger.error(f"Trading loop error: {e}")
        await asyncio.sleep(60)

async def stop(self):
    """Stop the trading engine gracefully"""
    self.is_running = False
    logger.info("Trading engine stopping...")

async def _trading_loop(self):
    """Main trading loop"""

    # Check if we should be trading
    if not self._should_trade():
        await asyncio.sleep(60)
        return

    # Check daily loss limit
    if self.daily_pnl <= -self.max_daily_loss:
        logger.warning(f"Daily loss limit reached: ${self.daily_pnl}")
        await asyncio.sleep(300) # Wait 5 minutes
        return

    # Generate signals
    crypto_signals = await self.signal_gen.generate_crypto_signals(top_n=5)
    stock_signals = await self.signal_gen.generate_stock_signals(top_n=5)

    all_signals = crypto_signals + stock_signals

    # Process signals
    for signal in all_signals:
        if len(self.positions) >= self.max_positions:
            break

        # Skip if already in position
        if signal.symbol in self.positions:
            continue
```

```
# Execute signal
await self._execute_signal(signal)

# Update open positions
await self._update_positions()

# Wait before next iteration
await asyncio.sleep(60) # Check every minute

def _should_trade(self) -> bool:
    """Check if we should be trading now"""
    # Crypto trades 24/7
    # Stocks only during market hours

    if not self.trading_hours_only:
        return True

    now = datetime.now(self.eastern)
    current_time = now.time()

    # Weekend check for stocks
    if now.weekday() >= 5:
        return True # Still trade crypto on weekends

    # Market hours for stocks
    if self.market_open <= current_time <= self.market_close:
        return True

    return True # Always allow crypto

async def _execute_signal(self, signal: TradingSignal):
    """Execute a trading signal"""

    logger.info(f"Executing signal: {signal.signal_type.value} {signal.symbol}")

    # Calculate position size
    account_value = await self._get_total_account_value()
    position_value = account_value * (signal.position_size_pct / 100)

    if signal.asset_type == 'crypto':
        # Execute on Kraken
        volume = position_value / signal.entry_price

        result = await self.kraken.place_bracket_order(
```

```

        pair=self._to_kraken_pair(signal.symbol),
        side=OrderSide.BUY,
        volume=volume,
        entry_price=None, # Market order
        take_profit_pct=(signal.take_profit / signal.entry_price - 1) * 100,
        stop_loss_pct=(1 - signal.stop_loss / signal.entry_price) * 100
    )

else:
    # Execute on Alpaca (stocks/ETFs)
    qty = position_value / signal.entry_price

    result = await self.alpaca.place_bracket_order(
        symbol=signal.symbol,
        side=AlpacaOrderSide.BUY,
        qty=qty,
        take_profit_pct=(signal.take_profit / signal.entry_price - 1) * 100,
        stop_loss_pct=(1 - signal.stop_loss / signal.entry_price) * 100
    )

if result.success:
    self.positions[signal.symbol] = {
        'signal': signal,
        'order_id': result.order_id,
        'entry_time': datetime.utcnow(),
        'entry_price': signal.entry_price,
        'current_price': signal.entry_price,
        'pnl': 0.0
    }

    self._log_trade(signal, result, 'ENTRY')
else:
    logger.error(f"Failed to execute signal for {signal.symbol}: {result.error}")

async def _update_positions(self):
    """Update all open positions"""
    for symbol, position in list(self.positions.items()):
        try:
            signal = position['signal']

            if signal.asset_type == 'crypto':
                # Check Kraken positions
                kraken_positions = await self.kraken.get_open_positions()
                # Update position data...

```

```

else:
    # Check Alpaca positions
    try:
        alpaca_position = await self.alpaca.get_position(symbol)
        current_price = float(alpaca_position.get('current_price', 0))
        unrealized_pnl = float(alpaca_position.get('unrealized_pnl', 0))

        position['current_price'] = current_price
        position['pnl'] = unrealized_pnl

    except:
        # Position closed (TP or SL hit)
        del self.positions[symbol]

    except Exception as e:
        logger.error(f"Error updating position {symbol}: {e}")

async def _get_total_account_value(self) -> float:
    """Get total account value across all brokers"""
    total = 0.0

    # Kraken balance
    try:
        balance = await self.kraken.get_trade_balance()
        total += float(balance.get('eb', 0)) # Equivalent balance
    except:
        pass

    # Alpaca balance
    try:
        account = await self.alpaca.get_account()
        total += float(account.get('portfolio_value', 0))
    except:
        pass

    return total

def _to_kraken_pair(self, symbol: str) -> str:
    """Convert symbol to Kraken pair format"""
    # BTC/USD -> XBTUSD
    symbol = symbol.upper().replace('/', '')
    if symbol.startswith('BTC'):
        symbol = 'XBT' + symbol[3:]


```

```
return symbol

def _log_trade(self, signal: TradingSignal, result, action: str):
    """Log trade to history and BigQuery"""
    trade = {
        'symbol': signal.symbol,
        'asset_type': signal.asset_type,
        'action': action,
        'signal_type': signal.signal_type.value,
        'confidence': signal.confidence,
        'entry_price': signal.entry_price,
        'take_profit': signal.take_profit,
        'stop_loss': signal.stop_loss,
        'order_id': result.order_id,
        'timestamp': datetime.utcnow().isoformat()
    }
```

```
self.trade_history.append(trade)
logger.info(f"Trade logged: {trade}")
```

```
# =====
# EMERGENCY CONTROLS
# =====
```

```
async def emergency_stop(self):
    """
    EMERGENCY: Stop all trading and close all positions
    """
    logger.warning("EMERGENCY STOP ACTIVATED")
```

```
    self.is_running = False
```

```
    # Close all crypto positions
    await self.kraken.emergency_close_all()
```

```
    # Close all stock positions
    await self.alpaca.emergency_liquidate()
```

```
    logger.warning("All positions closed")
```

```
# Global trading engine
trading_engine = TradingEngine()
```

---

# RISK MANAGEMENT SYSTEM

## Risk Parameters Configuration

```
yaml
```

```
# Risk Management Configuration (.env)
# =====

# Position Limits
MAX_POSITION_SIZE_USD=1000      # Max single position
MAX_PORTFOLIO_RISK_PCT=2.0       # Max risk per trade
MAX_TOTAL_POSITIONS=10          # Max concurrent positions
MAX_CRYPTO_POSITIONS=5           # Max crypto positions
MAX_STOCK_POSITIONS=5            # Max stock positions

# Loss Limits
MAX_DAILY_LOSS_USD=500          # Stop trading after this loss
MAX_WEEKLY_LOSS_USD=1500         # Weekly loss limit
MAX_MONTHLY_LOSS_USD=3000        # Monthly loss limit
MAX_DRAWDOWN_PCT=10              # Max portfolio drawdown

# Signal Quality
MIN_SIGNAL_CONFIDENCE=0.65       # Minimum confidence to trade
MIN_GROWTH_SCORE=60               # Minimum growth score

# Time Limits
MAX_TRADE_DURATION_HOURS=72      # Auto-close after 72 hours
TRADING_HOURS_ONLY=true          # Only trade during market hours

# Emergency
ENABLE_KILL_SWITCH=true          # Enable emergency stop
KILL_SWITCH_LOSS_PCT=5             # Trigger kill switch at 5% daily loss
```

---

# ENVIRONMENT CONFIGURATION

## Complete .env.prod Configuration

```
bash
```

```
# =====
# AIALGOTRADEHITS PRODUCTION CONFIGURATION
# =====

# GCP Configuration
GCP_PROJECT_ID=cryptobot-462709
GCP_REGION=us-central1
BIGQUERY_DATASET=crypto_trading_data
BIGQUERY_LOCATION=US

# =====
# BROKER API KEYS
# =====

# Kraken Pro (Crypto)
KRAKEN_API_KEY=${KRAKEN_API_KEY_PROD}
KRAKEN_API_SECRET=${KRAKEN_API_SECRET_PROD}
KRAKEN_WS_URL=wss://ws.kraken.com/
KRAKEN_REST_URL=https://api.kraken.com
KRAKEN_COUNTER_MAX=20
KRAKEN_COUNTER_DECAY=1.0

# Alpaca (Stocks/ETFs)
ALPACA_API_KEY_ID=${ALPACA_API_KEY_ID_PROD}
ALPACA_API_SECRET_KEY=${ALPACA_API_SECRET_KEY_PROD}
ALPACA_BASE_URL=https://api.alpaca.markets
ALPACA_DATA_URL=https://data.alpaca.markets
ALPACA_PAPER_URL=https://paper-api.alpaca.markets

# =====
# DATA API KEYS
# =====

# TwelveData (Primary Data Source)
TWELVEDATA_API_KEY=${TWELVEDATA_API_KEY_PROD}
TWELVEDATA_BASE_URL=https://api.twelvedata.com
TWELVEDATA_DAILY_CREDITS=1152000
TWELVEDATA_TARGET_CREDITS=576000

# FRED (Economic Data)
FRED_API_KEY=${FRED_API_KEY}
FRED_BASE_URL=https://api.stlouisfed.org/fred
```

```
# FinnHub (News & Sentiment)
FINNHUB_API_KEY=${FINNHUB_API_KEY_PROD}
FINNHUB_BASE_URL=https://finnhub.io/api/v1

# CoinMarketCap (Crypto Metadata)
CMC_API_KEY=${CMC_API_KEY_PROD}
CMC_BASE_URL=https://pro-api.coinmarketcap.com

# =====
# AI CONFIGURATION
# =====

VERTEX_AI_MODEL=gemini-2.5-pro
VERTEX_AI_FALLBACK=gemini-2.0-pro
AI_TEMPERATURE=0.1
AI_MAX_TOKENS=8192

ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY_PROD}
CLAUDE_MODEL=claude-sonnet-4-20250514

# =====
# TRADING PARAMETERS
# =====

# Position Limits
MAX_POSITION_SIZE_USD=1000
MAX_STOCK_POSITION_USD=5000
MAX_TOTAL_POSITIONS=10
MAX_DAILY_TRADES=50

# Risk Limits
MAX_DAILY_LOSS_USD=500
MAX_PORTFOLIO_RISK_PCT=2.0
MAX_DRAWDOWN_PCT=10

# Signal Quality
MIN_SIGNAL_CONFIDENCE=0.65
MIN_GROWTH_SCORE=60

# =====
# FAULT TOLERANCE
# =====

RETRY_MAX_ATTEMPTS=5
```

```
RETRY_BASE_DELAY=1.0
RETRY_MAX_DELAY=60.0
CIRCUIT_FAILURE_THRESHOLD=5
CIRCUIT_TIMEOUT=60.0

# =====
# MONITORING
# =====

ENABLE_KILL_SWITCH=true
ALERT_ON_TRADE=true
LOG_LEVEL=INFO
```

## DEPLOYMENT & MONITORING

### Cloud Functions Deployment

```
bash
```

```

# Deploy Trading Engine
gcloud functions deploy trading-engine \
--gen2 --runtime=python311 \
--region=us-central1 \
--memory=2GB --timeout=540s \
--min-instances=1 \
--set-env-vars-file=.env.prod \
--trigger-http

# Deploy Signal Generator
gcloud functions deploy signal-generator \
--gen2 --runtime=python311 \
--region=us-central1 \
--memory=4GB --timeout=540s \
--set-env-vars-file=.env.prod \
--trigger-http

# Create Scheduler for Signal Generation
gcloud scheduler jobs create http trading-signals-job \
--location=us-central1 \
--schedule="*/5 * * * *" \
--time-zone="America/New_York" \
--uri="https://signal-generator-xxxxx-uc.a.run.app" \
--http-method=POST

```

## QUICK START GUIDE

### Step 1: Set Up Broker Accounts

1. KRAKEN PRO (for Crypto):
  - Sign up at: <https://www.kraken.com>
  - Complete identity verification
  - Enable 2FA
  - Create API keys with trading permissions
  - Fund account with USD or crypto
  
2. ALPACA (for Stocks/ETFs):
  - Sign up at: <https://alpaca.markets>
  - No minimum balance required

- Create API keys
- Start with Paper Trading to test

## Step 2: Configure Environment

```
bash

# Copy environment template
cp .env.example .env.prod

# Edit with your API keys
nano .env.prod
```

## Step 3: Test with Paper Trading

```
python

# Use paper trading accounts first
alpaca_client = AlpacaTradingClient(paper=True)

# Test a trade
result = await alpaca_client.place_order(
    symbol='AAPL',
    side=AlpacaOrderSide.BUY,
    order_type=AlpacaOrderType.MARKET,
    qty=1
)
print(result)
```

## Step 4: Go Live

```
python

# Switch to live trading
alpaca_client = AlpacaTradingClient(paper=False)

# Start trading engine
await trading_engine.start()
```

