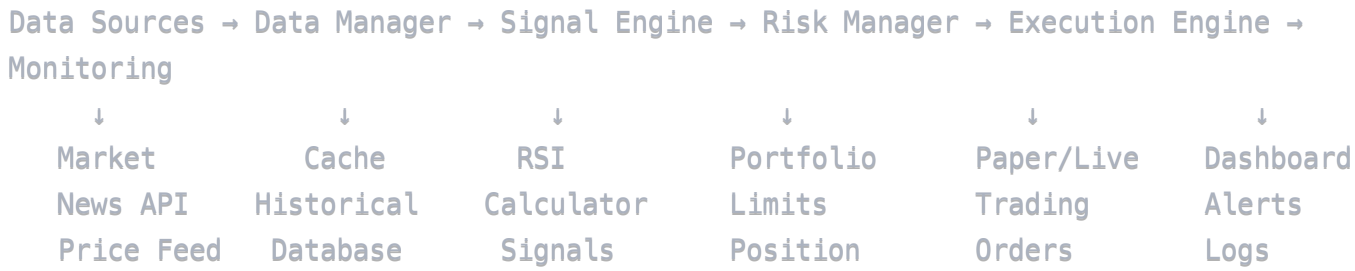
# RSI Trading System Workflow

## Core Architecture Overview




## File Structure and Language Assignments

### 1. Data Layer (Python)

`data_manager.py` - Main data orchestration

- **Language**: Python (asyncio for concurrent feeds)
- **Purpose**: Coordinate all data sources, handle connection failures
- **Key Functions**:
  - Manage multiple API connections
  - Route data to appropriate processors
  - Handle reconnection logic
  - Data validation and cleaning

`market_data_feed.py` - Real-time price data

- **Language**: Python (websocket/REST API clients)
- **Purpose**: Connect to market data providers (Alpha Vantage, Yahoo Finance, IEX Cloud)
- **Key Functions**:
  - WebSocket connections for real-time prices
  - REST API fallbacks
  - Data normalization across providers
  - Rate limiting compliance

`news_sentiment.py` - News and sentiment analysis

- **Language**: Python (NLTK/TextBlob for basic NLP)

- **Purpose**: Process financial news for sentiment overlay
- **Key Functions**:
  - News API integration (NewsAPI, Finnhub)
  - Basic sentiment scoring
  - Keyword filtering for relevance
  - Sentiment caching

### 2. Storage Layer (Python + SQLite)

`database_manager.py` - Data persistence

- **Language**: Python (SQLAlchemy ORM)
- **Purpose**: Handle all database operations
- **Key Functions**:
  - SQLite database management
  - Historical data storage
  - Signal logging
  - Performance tracking

`cache_manager.py` - In-memory data storage

- **Language**: Python (Redis-py or custom dict structures)
- **Purpose**: High-speed data access for real-time calculations
- **Key Functions**:
  - Circular buffer management
  - RSI component caching
  - Recent price storage
  - Signal state management

### 3. Analysis Engine (Python + NumPy)

`rsi_calculator.py` - Core RSI computation

- **Language**: Python (NumPy for vectorized operations)
- **Purpose**: Calculate RSI indicators efficiently
- **Key Functions**:
  - Incremental RSI updates

- Multi-stock batch processing
- Historical RSI calculation
- Custom period support

`signal_generator.py` - Trading signal logic

- **Language**: Python
- **Purpose**: Generate buy/sell signals from RSI data
- **Key Functions**:
  - Long-short signal detection
  - Threshold management
  - Signal filtering and validation
  - Multi-timeframe confirmation

`technical_indicators.py` - Additional indicators

- **Language**: Python (NumPy/Pandas)
- **Purpose**: Support indicators for signal confirmation
- **Key Functions**:
  - Moving averages
  - Volume analysis
  - Volatility measures
  - Trend confirmation

### 4. Risk Management (Python)

`risk_manager.py` - Portfolio risk controls

- **Language**: Python
- **Purpose**: Implement position sizing and risk limits
- **Key Functions**:
  - Position sizing algorithms
  - Exposure limits
  - Drawdown monitoring
  - Risk-adjusted returns

`portfolio_manager.py` - Position tracking

- **Language**: Python
- **Purpose**: Track all positions and performance
- **Key Functions**:
  - Position tracking
  - P&L calculation
  - Performance metrics
  - Portfolio rebalancing

### 5. Execution Layer (Python)

`order_manager.py` - Order handling

- **Language**: Python
- **Purpose**: Manage order lifecycle
- **Key Functions**:
  - Paper trading simulation
  - Order validation
  - Fill simulation
  - Order status tracking

`broker_interface.py` - Broker connectivity (future)

- **Language**: Python (broker-specific APIs)
- **Purpose**: Connect to actual brokers for live trading
- **Key Functions**:
  - Broker API integration
  - Authentication handling
  - Order submission
  - Account monitoring

### 6. Configuration and Control (Python + JSON)

`config.py` - System configuration

- **Language**: Python
- **Purpose**: Centralized configuration management
- **Key Functions**:

- Parameter management
- Environment settings
- API credentials
- Trading parameters

`settings.json` - Configuration file

- **Language**: JSON
- **Purpose**: Store adjustable parameters
- **Contents**:
  - RSI thresholds
  - Stock universe
  - Risk parameters
  - API endpoints

### 7. Monitoring and UI (Python + HTML)

`main_controller.py` - System orchestrator

- **Language**: Python (asyncio for concurrent operations)
- **Purpose**: Main event loop and system coordination
- **Key Functions**:
  - Start/stop all components
  - Error handling and recovery
  - System health monitoring
  - Graceful shutdown

`web_dashboard.py` - Monitoring interface

- **Language**: Python (Flask/FastAPI + HTML templates)
- **Purpose**: Real-time system monitoring
- **Key Functions**:
  - Live position display
  - Performance charts
  - System status
  - Manual controls

`dashboard.html` - Web interface

- **Language**: HTML/CSS/JavaScript
- **Purpose**: User interface for monitoring
- **Features**:
  - Real-time charts (Chart.js)
  - Position tables
  - System logs
  - Control buttons

### 8. Utilities and Logging (Python)

`logger.py` - Logging system

- **Language**: Python (logging module)
- **Purpose**: Comprehensive system logging
- **Key Functions**:
  - Multi-level logging
  - File rotation
  - Performance logging
  - Error tracking

`utils.py` - Helper functions

- **Language**: Python
- **Purpose**: Common utility functions
- **Key Functions**:
  - Date/time utilities
  - Math helpers
  - Validation functions
  - Format converters

## Execution Flow

### 1. System Startup

```python
main_controller.py
├── Load config.py and settings.json
├── Initialize database_manager.py
├── Start cache_manager.py
├── Launch market_data_feed.py
├── Start news_sentiment.py
└── Begin signal_generator.py loop
```

### 2. Real-time Processing Loop

```python
# Every market data update:
market_data_feed.py → cache_manager.py → rsi_calculator.py → signal_generator.py → ris
```

### 3. Monitoring and Control

```python
web_dashboard.py serves dashboard.html
├── Display real-time positions
├── Show performance metrics
├── System health status
└── Manual override controls
```

## Development Priority Order

1. **Phase 1**: Data layer and basic RSI calculation
   - `market_data_feed.py`
   - `cache_manager.py`
   - `rsi_calculator.py`
2. **Phase 2**: Signal generation and basic risk management
   - `signal_generator.py`
   - `risk_manager.py`
   - `portfolio_manager.py`
3. **Phase 3**: Execution and monitoring
   - `order_manager.py`

- `web_dashboard.py`
- `main_controller.py`

4. **Phase 4**: Enhancement and optimization
   - `news_sentiment.py`
   - `technical_indicators.py`
   - Performance optimization

## Hardware Optimization Notes

- **Memory Management**: Use generators and iterators where possible
- **CPU Efficiency**: Leverage NumPy vectorization for batch calculations
- **Storage**: Use SQLite for persistence, in-memory structures for speed
- **Concurrency**: asyncio for I/O-bound operations, multiprocessing for CPU-bound tasks
- **GPU Utilization**: Consider CuPy for large-scale parallel RSI calculations across many stocks

## Dependencies

**Core Libraries**:

- `numpy` - Numerical computations
- `pandas` - Data manipulation
- `asyncio` - Concurrent programming
- `aiohttp` - Async HTTP client
- `websockets` - Real-time data feeds
- `sqlalchemy` - Database ORM
- `flask/fastapi` - Web dashboard
- `nltk/textblob` - Basic NLP

**Optional Enhancements**:

- `cupy` - GPU acceleration
- `redis-py` - Advanced caching
- `plotly/dash` - Advanced charting
- `numba` - JIT compilation for speed