Trading Signal System Spezifikationsdokument

1. Projektziel und Anforderungen

1.1 Überblick

Entwicklung eines Trading Signal Systems, das Finanznachrichten und Marktdaten in Echtzeit analysiert, um Handelsempfehlungen für kurzfristige Investitionen in Derivate zu generieren. Das System soll diese Signale über Telegram an den Benutzer übermitteln, der dann manuell über die Trade Republic App handeln kann.

1.2 Kernfunktionalitäten

- Kontinuierliche Überwachung von Finanznachrichten und Marktdaten (mit maximaler Latenz von einer Minute)
- Automatische Analyse von Nachrichten und technischen Indikatoren
- Generierung von Trading-Signalen mit Begründungen
- Benachrichtigung über Telegram
- Tracking und Evaluierung der Signal-Performance
- Unterbrechbare ML-Verarbeitung auf dem Windows PC
- Web-Dashboard zur Visualisierung der Signal-Performance

1.3 Einschränkungen

- Keine automatische Orderausführung
- Keine Trade Republic API-Nutzung
- Maximales Investment von 1.000€ pro Signal
- Betrieb auf lokaler Hardware (Raspberry Pi 5 und PC)

2. Hardware- und Softwarespezifikationen

2.1 Hardware

- **Raspberry Pi 5**: 24/7-Betrieb für kontinuierliche Datenerfassung und -analyse
  - Prozessor: ARM Cortex-A76 Quad-Core @ 2,4 GHz
  - RAM: 8GB
  - Speicher: 256GB SSD empfohlen
- **Windows PC**: Für ML-Modell-Training und komplexe Analysen während der Tageszeit
  - GPU: NVIDIA RTX 3090 Ti

- CPU: AMD Ryzen 5 5600X 6-Core
- RAM: 16GB
- Speicher: 128GB dedizierte Partition für ML-Modelle

2.2 Software-Stack

Die folgenden Versionen sind verifiziert kompatibel (Stand: 13. Juni 2025):

Kernbibliotheken

- Python 3.12 oder 3.13
- yfinance 0.2.63
- pandas 2.3.0
- scikit-learn 1.7.0
- PyTorch 2.7.1 (mit CUDA Support für RTX 3090 Ti)
- streamlit 1.45.1
- python-telegram-bot 22.1
- schedule 1.2.2
- SQLite (für Datenbank)

Entwicklungsumgebung

- Visual Studio Code mit Python-Extensions

2.3 Cloud-Ressourcen

- pCloud (2TB verfügbar) für wöchentliche Backups

3. Systemarchitektur

3.1 Komponenten-Übersicht

Das System besteht aus sechs Hauptkomponenten, die modular und unabhängig voneinander arbeiten:

1. **Datensammlung**: Kontinuierliche Erfassung von Marktdaten und Finanznachrichten
2. **Nachrichtenanalyse**: Sentiment-Analyse von Finanznachrichten mit NLP
3. **Technische Analyse**: Berechnung technischer Indikatoren und Muster
4. **Signal-Generierung**: Kombination von Nachrichtenanalyse und technischen Indikatoren
5. **Benachrichtigungssystem**: Telegram-Integration für Signalübermittlung
6. **Performance-Tracking**: Evaluierung und Visualisierung der Signalqualität

3.2 Verteilte Architektur

- **Raspberry Pi 5**: Kontinuierliche Datensammlung, Basisanalyse, Signalgenerierung, Benachrichtigungen
- **Windows PC**: ML-Modell-Training, komplexe Analysen, Web-Dashboard

3.3 Datenbanken

- SQLite für persistente Datenspeicherung
  - Marktdaten-Tabelle
  - Nachrichten-Tabelle
  - Signal-Tabelle
  - Performance-Tabelle

4. Komponentendetails

4.1 Datensammlung

- **Yahoo Finance API** (yfinance 0.2.63)
  - Historische Preisdaten
  - Echtzeit-Kursdaten (mit Latenz < 1 Minute)
  - Firmeninformationen und Kennzahlen
- **Nachrichtenquellen**
  - Finanzielle Pressemitteilungen
  - Unternehmensberichte
  - Wirtschaftsnachrichten

Technische Implementierung:

```
# Beispiel für Datensammlung mit yfinance 0.2.63
import yfinance as yf

def fetch_stock_data(ticker, period="1d", interval="1m"):
    stock = yf.Ticker(ticker)
    data = stock.history(period=period, interval=interval, auto_adjust=True)

    # Handling der neuen Multi-Index-Struktur in yfinance 0.2.63
    if isinstance(data.columns, pd.MultiIndex):
        data.columns = ['_'.join(col).strip() for col in data.columns.values]

    return data
```

4.2 Nachrichtenanalyse

- **Sentiment-Analyse mit PyTorch**
  - FinBERT für Finanznachrichten-Sentiment
  - Konfidenzwerte für positive/negative Stimmung
- **Entitätserkennung**
  - Identifikation von Unternehmen, Personen, Ereignissen
  - Zuordnung zu Trading-Instrumenten

4.3 Technische Analyse

- **Indikatoren**
  - Moving Averages (SMA, EMA)
  - Relative Strength Index (RSI)
  - MACD (Moving Average Convergence Divergence)
  - Bollinger Bands
- **Pattern Recognition**
  - Unterstützungs- und Widerstandsniveaus
  - Trendlinien und Trendbrüche
  - Candlestick-Muster

4.4 Signal-Generierung

- **Kriterien**
  - Kombination aus Nachrichten-Sentiment und technischen Indikatoren
  - Mindestkonfidenzbewertung von 0,7
  - Risikobewertung und Volatilitätsanalyse
- **Signal-Struktur**
  - Instrument (Ticker)
  - Richtung (Kaufen/Verkaufen)
  - Konfidenzniveau
  - Begründung
  - Erwarteter Zeitrahmen

4.5 Benachrichtigungssystem

- **Telegram-Bot**
  - Sofortige Signalbenachrichtigungen
  - Formatierte Nachrichtenstruktur

- Interaktive Befehle zur Statusabfrage

Implementierungsbeispiel:

```
# Telegram-Bot für Signalbenachrichtigungen
from telegram.ext import Updater, CommandHandler

def send_signal(bot_token, chat_id, message):
    """Sendet ein Trading-Signal an den spezifizierten Telegram-Chat."""
    updater = Updater(token=bot_token, use_context=True)
    dispatcher = updater.dispatcher

    # Signal formatieren
    formatted_message = f"""
 NEUES TRADING SIGNAL 
------------------------
 Symbol: {message['symbol']}
 Richtung: {message['direction']}
 Zeitstempel: {message['timestamp']}
 Konfidenzniveau: {message['confidence']}
 Begründung: {message['reason']}
------------------------
    """

    # Signal senden
    updater.bot.send_message(chat_id=chat_id, text=formatted_message)
```

4.6 Performance-Tracking

- **Datenbankstruktur**
  - Signalhistorie
  - Tatsächliche Marktbewegungen
  - Erfolgsraten und Genauigkeitsmetriken
- **Visualisierung**
  - Web-Dashboard mit Streamlit
  - Performance-KPIs und Trends
  - Signal-Erfolgsraten nach Kategorien

4.7 Unterbrechbare ML-Verarbeitung

- **Event-basierte Steuerung**
  - Signal-Handler für Unterbrechung und Fortsetzung
  - Checkpoint-System für Zustandsspeicherung
- **Windows Task Scheduler**
  - Automatischer Start um 09:00 Uhr
  - Graceful Shutdown um 17:00 Uhr

Implementierungsansatz:

```
# Unterbrechbare ML-Verarbeitung
import threading
import signal
import pickle
import os

class InterruptibleProcessor:
    def __init__(self, checkpoint_path='checkpoint.pkl'):
        self.pause_event = threading.Event()
        self.exit_event = threading.Event()
        self.checkpoint_path = checkpoint_path
        self.state = self._load_checkpoint() or {}

        # Signal-Handler registrieren
        signal.signal(signal.SIGINT, self._handle_interrupt)
        signal.signal(signal.SIGTERM, self._handle_terminate)

    def _handle_interrupt(self, signum, frame):
        """Toggle zwischen Pause und Fortsetzung."""
        if self.pause_event.is_set():
            print("Verarbeitung wird fortgesetzt...")
            self.pause_event.clear()
        else:
            print("Verarbeitung wird pausiert...")
            self.pause_event.set()
            self._save_checkpoint()

    def _handle_terminate(self, signum, frame):
        """Beendet die Verarbeitung ordnungsgemäß."""
        print("Beende Verarbeitung...")
        self._save_checkpoint()
        self.exit_event.set()

    def _save_checkpoint(self):
        """Speichert den aktuellen Zustand."""
        with open(self.checkpoint_path, 'wb') as f:
            pickle.dump(self.state, f)
        print(f"Checkpoint gespeichert: {self.checkpoint_path}")

    def _load_checkpoint(self):
        """Lädt den letzten gespeicherten Zustand."""
        if os.path.exists(self.checkpoint_path):
            with open(self.checkpoint_path, 'rb') as f:
                return pickle.load(f)
        return None

    def process(self, items):
        """Verarbeitet Items mit Unterbrechungsmöglichkeit."""
        for item in items:
            # Prüfen ob pausiert
            while self.pause_event.is_set():
                if self.exit_event.is_set():
                    return
                time.sleep(0.5)
```

```
            # Prüfen ob beendet
            if self.exit_event.is_set():
                return

            # Verarbeitung durchführen
            self._process_item(item)

    def _process_item(self, item):
        # Implementierung der tatsächlichen Verarbeitungslogik
        pass
```

5. Datenfluss

5.1 Hauptdatenfluss

1. Kontinuierliche Erfassung von Marktdaten über Yahoo Finance API
2. Speicherung in der SQLite-Datenbank
3. Analyse der Daten (technische Indikatoren und Nachrichten-Sentiment)
4. Generierung von Trading-Signalen bei eindeutigen Mustern
5. Benachrichtigung des Nutzers über Telegram
6. Tracking der Signal-Performance
7. Visualisierung im Web-Dashboard

5.2 Kommunikation zwischen Raspberry Pi und PC

- SQLite-Datenbank als zentraler Datenaustausch-Hub
- Checkpoint-System für ML-Modellzustände
- Automatische Synchronisation zu definierten Zeitpunkten

6. Web-Dashboard

6.1 Funktionen

- Übersicht über aktive Signale
- Historische Performance-Metriken
- Erfolgsraten nach Kategorien
- Detailansicht einzelner Signale
- Gesamtportfolio-Performance

6.2 Technische Umsetzung

- Streamlit 1.45.1 für Web-Interface
- Lokaler Zugriff im ersten Projektschritt
- Automatisches Start/Stop mit dem PC

Implementierungsbeispiel:

```
# Streamlit Dashboard für Signal-Tracking
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt

def load_signal_data():
    # Hier: Daten aus der SQLite-Datenbank laden
    pass

def main():
    st.title("Trading Signal Dashboard")

    # Seitenleiste mit Filteroptionen
    st.sidebar.header("Filter")
    timeframe = st.sidebar.selectbox(
        "Zeitraum",
        ["Heute", "Diese Woche", "Dieser Monat", "Gesamt"]
    )

    # Hauptbereich
    st.header("Performance-Übersicht")

    # KPI-Karten
    col1, col2, col3 = st.columns(3)
    with col1:
        st.metric(label="Erfolgsrate", value="67%", delta="5%")
    with col2:
        st.metric(label="Durchschn. Gewinn", value="2.4%", delta="-0.2%")
    with col3:
        st.metric(label="Aktive Signale", value="3", delta="1")

    # Performance-Chart
    st.subheader("Signal-Performance im Zeitverlauf")
    # Hier: Matplotlib oder Plotly-Chart mit Performance-Daten

    # Signal-Historie
    st.subheader("Signal-Historie")
    # Hier: DataFrame mit Signalen anzeigen

if __name__ == "__main__":
    main()
```

7. Backup-Strategie

7.1 Datenbank-Backup

- Wöchentliche Sicherung zu pCloud
- Automatische Rotation älterer Backups

7.2 ML-Modell-Sicherung

- Speicherung von Checkpoints nach jedem Training
- Wöchentliche Sicherung der besten Modelle

8. Implementierungsphasen

8.1 Phase 1: Grundsystem (Wochen 1-4)

- Raspberry Pi 5 Setup und Konfiguration
- Datenerfassung mit yfinance
- SQLite-Datenbankdesign
- Telegram-Bot-Integration
- Basis-Datenanalyse

8.2 Phase 2: ML-Pipeline (Wochen 5-8)

- Sentiment-Analyse mit PyTorch
- Technische Indikatoren-Implementierung
- Unterbrechbare ML-Pipeline
- Windows-Integration
- Performance-Tracking

8.3 Phase 3: Dashboard und Produktivbetrieb (Wochen 9-12)

- Streamlit-Dashboard-Entwicklung
- Backtesting und Performance-Validierung
- 24/7-Deployment
- Feinabstimmung der Signalgenerierung

9. Projekteinschränkungen und Sicherheitsüberlegungen

9.1 Technische Einschränkungen

- Keine fortgeschrittenen Sicherheitsfeatures in der ersten Ausbaustufe
- Lokaler Zugriff auf das Dashboard (kein Remote-Zugriff)
- Limitierte Nachrichtenquellen (primär Yahoo Finance)

9.2 Betriebseinschränkungen

- Maximale Positionsgröße von 1.000€ pro Signal
- Manuelle Ausführung aller Trades
- Mindestkonfidenzniveau für Signale

10. Offene Punkte und zukünftige Erweiterungen

10.1 Potenzielle Erweiterungen

- Integration weiterer Nachrichtenquellen
- Erweiterte ML-Modelle für Preisprognosen
- Remote-Zugriff auf das Dashboard
- Automatische Modelloptimierung

10.2 Bekannte Einschränkungen

- Latenz der Yahoo Finance API
- Limitierte Nachrichtenquellen
- Keine automatisierte Handelsmöglichkeit

Dieses Spezifikationsdokument dient als Grundlage für die Implementierung des Trading Signal Systems und kann im Projektverlauf weiter angepasst und verfeinert werden.

Erstellt: 13. Juni 2025