Building an Agentic RAG Model for Crypto Trading with Haystack, Weaviate, and OpenRouter.ai

Grok 3, xAI

July 7, 2025

Contents

1 Introduction

This guide provides a detailed roadmap for building an Agentic Retrieval-Augmented Generation (RAG) model for cryptocurrency trading using the **Haystack framework** (v2.0+), **Weaviate** as the document store, and **OpenRouter.ai** for multi-model LLM integration. The system leverages three vector databases to store data from CoinMarketCap APIs, Binance APIs with technical analysis, and news/social media platforms (e.g., X) with sentiment analysis. A robust trading strategy combining multiple technical indicators and risk management principles is integrated to maximize profit and minimize risk. The system empowers a frontend application for users to create intelligent AI trading agents that make data-driven decisions and execute trades on selected cryptocurrency pairs.

The system supports:

- Real-time market data analysis with technical indicators.
- Sentiment-driven trading strategies.
- Autonomous trade execution with user-defined parameters.
- A user-friendly React-based frontend for agent configuration and monitoring.

2 System Architecture

The architecture integrates Haystack's RAG pipeline, Weaviate for vector storage, OpenRouter.ai for generative AI, and a robust trading strategy.

2.1 Architecture Components

- **Data Sources**:
  - **Vector Database 1 (CoinMarketCap)**: Stores market data (price, market cap, volume).
  - **Vector Database 2 (Binance)**: Stores trading data with technical indicators (e.g., EMA, RSI, MACD, Bollinger Bands).
  - **Vector Database 3 (Sentiment)**: Stores news and X posts with sentiment scores.
- **Haystack RAG Pipeline**:
  - **Document Store**: Weaviate with three classes for data storage.
  - **Embedder**: `sentence-transformers/all-MiniLM-L6-v2`.
  - **Retriever**: `WeaviateHybridRetriever` for BM25 and dense search.
  - **Generator**: OpenRouter.ai's LLM (e.g., LLaMA 3.1 70B).
  - **Pipeline**: Combines components for query processing.
- **Haystack Agent**: Autonomous agents using tools for data retrieval, indicator computation, and trade execution.
- **Frontend Application**: React with Tailwind CSS and Chart.js for visualization.
- **Execution Layer**: Binance API for trade execution with safeguards.

2.2 Architecture Diagram (Text Description)

- **User Input**: Users configure agents via the frontend (e.g., BTC/USDT, 2% risk).
- **Haystack Agent**: Queries the RAG pipeline and executes trades based on a confluence strategy.
- **RAG Pipeline**:
  - **Query Processing**: Parses user goals (e.g., "Find oversold BTC/USDT with positive sentiment").
  - **Retriever**: Fetches data from Weaviate.
  - **Generator**: Synthesizes trading signals using OpenRouter.ai.
- **Weaviate Document Store**: Stores embeddings for all data sources.
- **Output**: Trading signals displayed on the frontend or executed via Binance API.

2.3 Robustness Recommendations

- Use a **microservices architecture** for scalability.
- Integrate **CoinGecko** as a backup for CoinMarketCap.
- Use **Kafka** for high-frequency data ingestion.
- Deploy on **AWS** with auto-scaling.

3 Trading Strategy

The trading strategy combines multiple technical indicators and risk management to maximize profit and reduce risk, emphasizing confluence for high-confidence decisions.

3.1 Indicator Categories

- **Trend Indicators**:
  - **EMA/SMA**: 50 EMA > 200 EMA for bullish trend.
  - **MACD**: Bullish/bearish crossovers for momentum shifts.
  - **Parabolic SAR**: Trailing stops and reversal points.
- **Momentum Indicators**:
  - **RSI**: Overbought (>70), oversold (<30).
  - **Stochastic Oscillator**: Short-term reversal signals.
  - **CCI**: Price deviations from historical norms.
- **Volatility Indicators**:
  - **Bollinger Bands**: Detect squeezes and breakouts.
  - **ATR**: Guide stop-loss levels based on volatility.
- **Volume Indicators**:
  - **OBV**: Confirm trend strength.

  - **VWAP**: Identify institutional interest.
- **Support & Resistance**:
  - **Horizontal S/R**: Historical reaction levels.
  - **Fibonacci Retracement**: 0.618 level for pullbacks.
  - **Pivot Points**: Intraday decision-making.
- **Price Action & Patterns**:
  - **Candlestick Patterns**: Engulfing, pin bars for reversals.
  - **Chart Patterns**: Triangles, head and shoulders for breakouts.
- **Risk Management**:
  - **Position Sizing**: Limit to 1–2% of account per trade.
  - **Trailing Stop-Loss**: Lock in gains.
  - **Risk-to-Reward Ratio**: Minimum 1:2.

3.2 Confluence Strategy

Buy Setup:

- 50 EMA > 200 EMA (bullish trend).
- RSI crosses above 30 (momentum shift).
- Price bounces from 0.618 Fibonacci support.
- MACD bullish crossover.
- Volume spike (OBV increasing).

Sell/Exit Setup:

- RSI > 70 with bearish divergence.
- MACD bearish crossover.
- Price hits resistance with extended ATR.
- Parabolic SAR flips above price.

Risk Management:

- Stop-loss: Set at 2x ATR below entry.
- Take-profit: Target 2x risk or next Fibonacci level.
- Position size: 1% of account to limit exposure.

4 Setting Up Weaviate Document Stores

Weaviate serves as the document store for all three vector databases, using separate classes for CoinMarketCap, Binance, and sentiment data. Below is a detailed setup process.

4.1 Weaviate Installation and Configuration

- Install Weaviate locally or on a cloud provider (e.g., AWS).
- Use Docker for local setup:

```
docker run -d -p 8080:8080 --name weaviate semitechnologies/weaviate:
    latest
```

- Configure Weaviate with authentication (optional) and enable hybrid search.
- Set `embedding_dim=384` for `sentence-transformers/all-MiniLM-L6-v2`.

4.2 Vector Database 1: CoinMarketCap API Data

Purpose: Store market data for cryptocurrencies.

Schema:

```
{
  "class": "CoinMarketCap",
  "properties": [
    {"name": "crypto_id", "dataType": ["string"]},
    {"name": "symbol", "dataType": ["string"]},
    {"name": "price", "dataType": ["number"]},
    {"name": "market_cap", "dataType": ["number"]},
    {"name": "volume_24h", "dataType": ["number"]},
    {"name": "timestamp", "dataType": ["date"]},
    {"name": "embedding", "dataType": ["number[]"]}
  ],
  "vectorizer": "none"
}
```

Ingestion Code:

```
import requests
from haystack_integrations.document_stores.weaviate import
    WeaviateDocumentStore
from sentence_transformers import SentenceTransformer
from weaviate import Client

# Initialize Weaviate
client = Client("http://localhost:8080")
document_store = WeaviateDocumentStore(url="http://localhost:8080",
    embedding_dim=384)
model = SentenceTransformer('all-MiniLM-L6-v2')

# Create schema
client.schema.create_class({
    "class": "CoinMarketCap",
    "properties": [
        {"name": "crypto_id", "dataType": ["string"]},
        {"name": "symbol", "dataType": ["string"]},
        {"name": "price", "dataType": ["number"]},
        {"name": "market_cap", "dataType": ["number"]},
        {"name": "volume_24h", "dataType": ["number"]},
        {"name": "timestamp", "dataType": ["date"]},
        {"name": "embedding", "dataType": ["number[]"]}
    ],
```

```
    "vectorizer": "none"
})

# Fetch and store data
api_key = "YOUR_CMC_API_KEY"
url = "https://pro-api.coinmarketcap.com/v1/cryptocurrency/listings/latest"
headers = {"X-CMC_PRO_API_KEY": api_key}
response = requests.get(url, headers=headers).json()

documents = []
for crypto in response['data']:
    data = {
        "crypto_id": str(crypto['id']),
        "symbol": crypto['symbol'],
        "price": crypto['quote']['USD']['price'],
        "market_cap": crypto['quote']['USD']['market_cap'],
        "volume_24h": crypto['quote']['USD']['volume_24h'],
        "timestamp": crypto['last_updated']
    }
    embedding = model.encode(str(data)).tolist()
    documents.append({"content": str(data), "meta": data, "embedding":
       embedding})
document_store.write_documents(documents)
```

4.3 Vector Database 2: Binance API Data with Technical Analysis

Purpose: Store trading data with technical indicators.

Schema:

```
{
  "class": "Binance",
  "properties": [
    {"name": "pair", "dataType": ["string"]},
    {"name": "timestamp", "dataType": ["date"]},
    {"name": "open", "dataType": ["number"]},
    {"name": "high", "dataType": ["number"]},
    {"name": "low", "dataType": ["number"]},
    {"name": "close", "dataType": ["number"]},
    {"name": "volume", "dataType": ["number"]},
    {"name": "rsi", "dataType": ["number"]},
    {"name": "macd", "dataType": ["number"]},
    {"name": "bb_upper", "dataType": ["number"]},
    {"name": "bb_lower", "dataType": ["number"]},
    {"name": "sto_k", "dataType": ["number"]},
    {"name": "cci", "dataType": ["number"]},
    {"name": "atr", "dataType": ["number"]},
    {"name": "obv", "dataType": ["number"]},
    {"name": "vwap", "dataType": ["number"]},
    {"name": "fib_618", "dataType": ["number"]},
    {"name": "pivot_point", "dataType": ["number"]},
    {"name": "embedding", "dataType": ["number[]"]}
  ],
  "vectorizer": "none"
}
```

Ingestion Code:

```python
import ccxt
import pandas as pd
import pandas_ta as ta
from haystack_integrations.document_stores.weaviate import
    WeaviateDocumentStore
from sentence_transformers import SentenceTransformer
from weaviate import Client

# Initialize Weaviate
client = Client("http://localhost:8080")
document_store = WeaviateDocumentStore(url="http://localhost:8080",
    embedding_dim=384)
model = SentenceTransformer('all-MiniLM-L6-v2')

# Create schema
client.schema.create_class({
    "class": "Binance",
    "properties": [
        {"name": "pair", "dataType": ["string"]},
        {"name": "timestamp", "dataType": ["date"]},
        {"name": "open", "dataType": ["number"]},
        {"name": "high", "dataType": ["number"]},
        {"name": "low", "dataType": ["number"]},
        {"name": "close", "dataType": ["number"]},
        {"name": "volume", "dataType": ["number"]},
        {"name": "rsi", "dataType": ["number"]},
        {"name": "macd", "dataType": ["number"]},
        {"name": "bb_upper", "dataType": ["number"]},
        {"name": "bb_lower", "dataType": ["number"]},
        {"name": "sto_k", "dataType": ["number"]},
        {"name": "cci", "dataType": ["number"]},
        {"name": "atr", "dataType": ["number"]},
        {"name": "obv", "dataType": ["number"]},
        {"name": "vwap", "dataType": ["number"]},
        {"name": "fib_618", "dataType": ["number"]},
        {"name": "pivot_point", "dataType": ["number"]},
        {"name": "embedding", "dataType": ["number[]"]}
    ],
    "vectorizer": "none"
})

# Fetch Binance data
binance = ccxt.binance()
ohlcv = binance.fetch_ohlcv('BTC/USDT', timeframe='15m', limit=100)
df = pd.DataFrame(ohlcv, columns=['timestamp', 'open', 'high', 'low', '
    close', 'volume'])

# Compute indicators
df['rsi'] = ta.rsi(df['close'], length=14)
df['macd'], _, _ = ta.macd(df['close'])
df['bb_upper'], _, df['bb_lower'] = ta.bbands(df['close'], length=20)
df['sto_k'], _ = ta.stoch(df['high'], df['low'], df['close'])
df['cci'] = ta.cci(df['high'], df['low'], df['close'])
df['atr'] = ta.atr(df['high'], df['low'], df['close'])
df['obv'] = ta.obv(df['close'], df['volume'])
df['vwap'] = ta.vwap(df['high'], df['low'], df['close'], df['volume'])
df['fib_618'] = df['close'].rolling(20).max() - (df['close'].rolling(20).
```

```
    max() - df['close'].rolling(20).min()) * 0.618
df['pivot_point'] = (df['high'].shift(1) + df['low'].shift(1) + df['close'
    ].shift(1)) / 3

# Store in Weaviate
documents = []
for idx, row in df.iterrows():
    data = row.to_dict()
    data['pair'] = 'BTC/USDT'
    embedding = model.encode(str(data)).tolist()
    documents.append({"content": str(data), "meta": data, "embedding":
        embedding})
document_store.write_documents(documents)
```

4.4 Vector Database 3: News and Social Media Sentiment

Purpose: Store news and X posts with sentiment scores.

Schema:

```
{
  "class": "Sentiment",
  "properties": [
    {"name": "source", "dataType": ["string"]},
    {"name": "text", "dataType": ["text"]},
    {"name": "crypto_mentioned", "dataType": ["string"]},
    {"name": "sentiment_score", "dataType": ["number"]},
    {"name": "timestamp", "dataType": ["date"]},
    {"name": "embedding", "dataType": ["number[]"]}
  ],
  "vectorizer": "none"
}
```

Ingestion Code:

```
from transformers import pipeline
from haystack_integrations.document_stores.weaviate import
    WeaviateDocumentStore
from sentence_transformers import SentenceTransformer
from weaviate import Client
import tweepy

# Initialize Weaviate
client = Client("http://localhost:8080")
document_store = WeaviateDocumentStore(url="http://localhost:8080",
    embedding_dim=384)
model = SentenceTransformer('all-MiniLM-L6-v2')
sentiment_analyzer = pipeline("sentiment-analysis", model="mrm8488/
    distilroberta-finetuned-financial-news-sentiment-analysis")

# Create schema
client.schema.create_class({
    "class": "Sentiment",
    "properties": [
        {"name": "source", "dataType": ["string"]},
        {"name": "text", "dataType": ["text"]},
        {"name": "crypto_mentioned", "dataType": ["string"]},
        {"name": "sentiment_score", "dataType": ["number"]},
```

```
        {"name": "timestamp", "dataType": ["date"]},
        {"name": "embedding", "dataType": ["number[]"]}
    ],
    "vectorizer": "none"
})

# Fetch X posts
client = tweepy.Client(bearer_token="YOUR_X_BEARER_TOKEN")
tweets = client.search_recent_tweets(query="#BTC", max_results=100).data

# Store in Weaviate
documents = []
for tweet in tweets:
    text = tweet.text
    sentiment = sentiment_analyzer(text)[0]
    data = {
        "source": "X",
        "text": text,
        "crypto_mentioned": "BTC",
        "sentiment_score": sentiment['score'] if sentiment['label'] == '
           positive' else -sentiment['score'],
        "timestamp": tweet.created_at.isoformat()
    }
    embedding = model.encode(text).tolist()
    documents.append({"content": text, "meta": data, "embedding": embedding
       })
document_store.write_documents(documents)
```

4.5 Robustness Recommendations

- Use Weaviate's **hybrid search** for optimal retrieval.
- Implement **data deduplication** using Weaviate's UUIDs.
- Validate API responses with schema checks.
- Schedule **incremental updates** with a cron job.

5 Building the Haystack RAG Pipeline

The Haystack RAG pipeline retrieves data from Weaviate and generates trading strategies using OpenRouter.ai.

5.1 Haystack Components

- **Document Store**: Weaviate with three classes.
- **Embedder**: `sentence-transformers/all-MiniLM-L6-v2`.
- **Retriever**: `WeaviateHybridRetriever` with top-k=5.
- **Generator**: OpenRouter.ai's LLaMA 3.1 70B via `OpenAIGenerator`.
- **Pipeline**: Combines retriever and generator with a custom prompt.

5.2 Pipeline Setup

- **Retriever**: Queries all Weaviate classes for relevant data.
- **Generator**: Synthesizes trading signals based on the confluence strategy.
- **Prompt Template**: Incorporates technical indicators and sentiment.

Pipeline Code:

```python
from haystack import Pipeline
from haystack_integrations.components.retrievers.weaviate import WeaviateHybridRetriever
from haystack.components.generators import OpenAIGenerator
from haystack_integrations.document_stores.weaviate import WeaviateDocumentStore

# Initialize components
document_store = WeaviateDocumentStore(url="http://localhost:8080", embedding_dim=384)
retriever = WeaviateHybridRetriever(document_store=document_store, top_k=5)
generator = OpenAIGenerator(
    api_key="YOUR_OPENROUTER_API_KEY",
    api_base_url="https://openrouter.ai/api/v1",
    model="meta-llama/llama-3.1-70b-instruct"
)

# Create pipeline
pipeline = Pipeline()
pipeline.add_component("retriever", retriever)
pipeline.add_component("generator", generator)
pipeline.connect("retriever.documents", "generator.documents")

# Query example
query = "Find trading opportunities for BTC/USDT with oversold RSI and positive sentiment"
result = pipeline.run({
    "retriever": {"query": query},
    "generator": {
        "prompt": """
        Given the following data:
        {documents}
        Suggest a trading strategy for BTC/USDT using:
        - 50 EMA > 200 EMA for trend
        - RSI crossing above 30
        - Price near 0.618 Fibonacci support
        - MACD bullish crossover
        - Volume spike (OBV increasing)
        Set stop-loss at 2x ATR below entry and take-profit at 2x risk.
        """
    }
})
print(result["generator"]["replies"])
```

6 Developing the Haystack Agent

The Haystack Agent implements the confluence strategy and executes trades.

6.1 Agent Setup

- **Tools**: Retrieve data, compute indicators, execute trades.
- **Logic**: Combines RAG outputs with strategy rules.
- **Execution**: Uses Binance API with risk management.

Agent Code:

```
from haystack import Pipeline
from haystack.components.generators import OpenAIGenerator
from haystack.agents import Agent, Tool
import ccxt
import pandas as pd
import pandas_ta as ta

# Trade execution tool
def place_trade(symbol, action, price, amount, stop_loss, take_profit):
    binance = ccxt.binance({'apiKey': 'YOUR_API_KEY', 'secret': '
        YOUR_API_SECRET'})
    try:
        if action == "buy":
            order = binance.create_limit_buy_order(symbol, amount, price)
            # Set stop-loss and take-profit (simplified)
            return f"Placed␣buy␣order:␣{order}"
        elif action == "sell":
            order = binance.create_limit_sell_order(symbol, amount, price)
            return f"Placed␣sell␣order:␣{order}"
    except Exception as e:
        return f"Error:␣{e}"

trade_tool = Tool(
    name="TradeExecutor",
    description="Executes␣trades␣on␣Binance␣with␣stop-loss␣and␣take-profit"
        ,
    function=place_trade
)

# Indicator computation tool
def compute_indicators(symbol, timeframe):
    binance = ccxt.binance()
    ohlcv = binance.fetch_ohlcv(symbol, timeframe, limit=100)
    df = pd.DataFrame(ohlcv, columns=['timestamp', 'open', 'high', 'low', '
        close', 'volume'])
    df['ema50'] = ta.ema(df['close'], length=50)
    df['ema200'] = ta.ema(df['close'], length=200)
    df['rsi'] = ta.rsi(df['close'], length=14)
    df['macd'], _, _ = ta.macd(df['close'])
    df['obv'] = ta.obv(df['close'], df['volume'])
    df['atr'] = ta.atr(df['high'], df['low'], df['close'])
    return df.iloc[-1].to_dict()

indicator_tool = Tool(
    name="IndicatorCalculator",
    description="Computes␣technical␣indicators␣for␣a␣trading␣pair",
    function=compute_indicators
)

# Initialize agent
```

```
generator = OpenAIGenerator(
    api_key="YOUR_OPENROUTER_API_KEY",
    api_base_url="https://openrouter.ai/api/v1",
    model="meta-llama/llama-3.1-70b-instruct"
)
agent = Agent(generator=generator, tools=[trade_tool, indicator_tool])

# Run agent
result = agent.run(
    query="For BTC/USDT on 15m timeframe, check if 50 EMA > 200 EMA, RSI > 30, and positive sentiment. If true, buy 0.01 BTC at market price with 2x ATR stop-loss.",
    prompt="Use IndicatorCalculator to check conditions and TradeExecutor to place the trade if valid."
)
print(result)
```

7 Building the Frontend Application

Tech Stack:

- Framework: React with Tailwind CSS.
- Visualization: Chart.js for candlestick charts and indicators.
- Backend: FastAPI for Haystack integration.

Features:

- Configure trading pairs, risk levels, and strategy parameters.
- Display real-time price charts, indicators, and sentiment.
- Approve or automate trades.
- Backtest strategies on historical data.

Frontend Code (React):

```
import React, { useState, useEffect } from 'react';
import { Line } from 'react-chartjs-2';
import axios from 'axios';

const Dashboard = () => {
  const [priceData, setPriceData] = useState([]);
  const [indicators, setIndicators] = useState({});

  useEffect(() => {
    // Fetch price and indicator data
    axios.get('/api/price/BTCUSDT').then(response => {
      setPriceData(response.data);
    });
    axios.get('/api/indicators/BTCUSDT').then(response => {
      setIndicators(response.data);
    });
  }, []);

  const chartData = {
    labels: priceData.map(d => d.timestamp),
```

```
    datasets: [
      { label: 'Price', data: priceData.map(d => d.close), borderColor: '
         blue' },
      { label: 'EMA 50', data: priceData.map(d => d.ema50), borderColor: '
         green' },
      { label: 'EMA 200', data: priceData.map(d => d.ema200), borderColor:
         'red' }
    ]
  };

  return (
    <div className="p-4">
      <h2 className="text-2xl font-bold">BTC/USDT Dashboard</h2>
      <Line data={chartData} />
      <div>RSI: {indicators.rsi?.toFixed(2)}</div>
      <div>MACD: {indicators.macd?.toFixed(2)}</div>
    </div>
  );
};

export default Dashboard;
```

8 Integration and Testing

Integration:

- Connect Haystack pipeline and agent to FastAPI.
- Use Binance testnet API for trade testing.

Testing:

- Validate data ingestion with schema checks.
- Test retriever accuracy with queries like "RSI < 30 and sentiment > 0.7".
- Simulate trades to ensure strategy compliance.

9 Deployment

9.1 Deployment Setup

- **Weaviate**: Deploy on AWS EC2 using Docker.
- **Haystack Backend**: Deploy FastAPI on AWS ECS with Docker.
- **Frontend**: Host React app on AWS Amplify with Cloudflare CDN.
- **Message Queue**: Use Kafka on AWS MSK for data ingestion.

9.2 Deployment Steps

1. **Provision Infrastructure**:
   - Launch EC2 instance for Weaviate (e.g., t3.medium).
   - Set up ECS cluster for FastAPI (use Fargate for serverless).

- Configure Amplify for React frontend.
- Deploy Kafka on MSK for data pipelines.

2. **Configure Weaviate**:
   - Run Weaviate Docker container with persistent storage (EBS).
   - Set environment variables for authentication and hybrid search.
3. **Deploy Haystack Backend**:
   - Create Docker image for FastAPI with Haystack dependencies.
   - Push to AWS ECR and deploy to ECS.
   - Configure load balancer (ALB) for API endpoints.
4. **Deploy Frontend**:
   - Build React app and push to Amplify.
   - Configure Cloudflare for CDN and DDoS protection.
5. **Set Up Monitoring**:
   - Use Prometheus and Grafana on EC2 for metrics.
   - Monitor API latency, trade execution, and Weaviate performance.
6. **Security**:
   - Encrypt API keys with AWS KMS.
   - Use IAM roles for ECS and Amplify.
   - Enable WAF on Cloudflare.

9.3 Maintenance

- Schedule Weaviate backups using EBS snapshots.
- Monitor OpenRouter.ai and Binance API rate limits.
- Update Haystack and LLM models regularly.

10 Advanced Features

- **On-Chain Integration**: Use Etherscan for wallet movements.
- **Backtesting**: Simulate strategies with historical Binance data.
- **Explainability**: Display reasons for agent decisions.

11 Regulatory Compliance

- Ensure KYC/AML compliance for trade execution.
- Include risk disclaimers in the frontend.
- Redirect to `https://help.x.com/en/using-x/x-premium` for subscriptions and `https://x.ai/api` for API pricing.

12 Conclusion

This guide provides a robust framework for building an Agentic RAG model for crypto trading using Haystack, Weaviate, and OpenRouter.ai. The confluence-based trading strategy ensures high-confidence decisions, while detailed deployment steps enable production readiness.

Next Steps:

1. Deploy Weaviate and set up data ingestion.
2. Build and test the Haystack pipeline.
3. Develop the React frontend.
4. Deploy on AWS and monitor performance.