ChatGPT

Hyper-Trading Automation System – Design & Implementation Plan

Project Overview & Goals

We aim to develop a **fully automated hyper-trading system** (Expert Advisor) for cryptocurrency markets on MetaTrader 5 (MT5). The system will autonomously ingest data, generate trading signals, execute orders, and manage risk – all with minimal latency and high reliability. The end deliverable is a production-ready GitHub repository with clean code, thorough documentation, tests, and CI/CD pipelines.

Primary performance targets:
- **Win Rate & Profitability:** Achieve a high win rate while maintaining a robust **Sharpe ratio** (reward-to-risk).
- **Drawdown Control:** Keep maximum drawdown below acceptable thresholds (e.g. <20%).
- **Low Latency:** Minimize latency from signal generation to order execution (target sub-second end-to-end).
- **Stable Operation:** 24/7 continuous trading with robust error handling and uptime.

System Components Overview

The hyper-trading system consists of several core modules working in tandem:

- **Data Ingestion:** Feeds for live price data (multi-exchange), historical data for backtesting, and news or sentiment data.
- **Feature Engineering:** Calculation of technical indicators (e.g. MACD, RSI, Bollinger Bands) and parsing news sentiment into numeric features.
- **Signal Generation:** Trading signals derived from strategies – combining rule-based indicator logic and possibly machine learning model predictions.
- **Execution Engine:** The MT5 Expert Advisor (written in MQL5) that places orders, sets stop-loss/take-profit, and handles position management with micro-lot sizing.
- **Risk & Portfolio Management:** Position sizing (capped risk per trade), enforcement of risk limits (max % equity at risk), and trade protective measures (SL/TP, trailing stops).
- **Monitoring & Logging:** Continuous logging of system status and trade actions, with potential integration into monitoring dashboards for live health checks.
- **Testing & CI/CD:** A framework for backtesting strategies on historical data, unit/integration tests for components, and automated CI/CD workflows (e.g. GitHub Actions) to ensure code quality and smooth deployment to the live environment.

These components together form a pipeline: data flows in, features are computed, strategies output signals, and the execution engine acts on them while managing risk (see **Architecture** section below for how these integrate).

Technology Stack & Dependencies

Programming Languages:
- **Python 3.x:** Used for data ingestion, feature engineering, machine learning, and orchestrating the strategy logic. Python offers rich libraries for data science and can interface with MetaTrader (via API or bridging).
- **MQL5 (MetaQuotes Language 5):** Used for coding the Expert Advisor that runs on MT5. MQL5 handles low-latency execution of trades on the trading platform side.

Key Libraries & Frameworks:
- **CCXT** (CryptoCurrency eXchange Trading library): For unified access to multiple exchange APIs (Binance, Coinbase, Kraken, etc.) to fetch live and historical price data. CCXT ensures broad exchange coverage and reliability.
- **Requests / Feedparser:** For pulling news feeds or RSS (e.g., using NewsAPI.org, CryptoPanic API, or CoinDesk RSS) to gather news headlines and social sentiment data.
- **Pandas, NumPy:** For data manipulation and indicator calculations (e.g., rolling means for moving averages). Potentially `ta-lib` or `pandas_ta` for technical indicators.
- **Machine Learning** (optional advanced signals): **TensorFlow** or **PyTorch** for model training/inference if we include an ML model to predict price movements. Additionally, libraries like **scikit-learn** or **Optuna** for feature scaling, model selection, and hyperparameter tuning.
- **Backtesting Framework: Backtrader** or **Zipline** integrated with MT5 price data. Backtrader can simulate strategy logic in Python, while we might also use MT5's Strategy Tester for EA-specific backtesting if needed.
- **CI/CD: GitHub Actions** for automating tests, linters (flake8/pylint for Python, and MQL5 code check if possible), and deployment scripts. We may use actions to compile the MQL5 EA and package releases.
- **Deployment Environment:** A cloud or VPS server (with Windows or Wine environment for MT5) will host the MT5 terminal and EA. For example, a Windows Server VPS with MetaTrader 5 installed, ensuring 24/7 uptime and network stability for live trading.

Note on Data Sources: We will prioritize free and reliable data sources. For instance, **Binance** and **Coinbase** (via CCXT) offer free REST APIs for price data, and NewsAPI.org provides a free tier for news data. This ensures the system doesn't depend on paid feeds that might introduce additional complexity or cost. If an API becomes unavailable, the design will allow falling back to alternative sources or locally cached data to maintain continuity.

Data Pipeline

Our data pipeline handles both **historical data collection** (for backtesting/model training) and **real-time streaming** (for live trading):

- **Historical Price Data:** We will use CCXT to fetch OHLCV price data for major cryptocurrency pairs (e.g., BTC/USD, ETH/USD) from multiple exchanges. This data will be stored in `data/` (possibly as CSV files or a lightweight database). Having multiple exchanges (Binance, Coinbase, Kraken, etc.) ensures redundancy and broad coverage; if one feed is down, another can be used. Historical data is crucial for strategy development and will feed into backtesting and model training.

- **Real-Time Price Feeds:** For live operation, the primary feed will come from the **MT5 platform** itself (connected to a broker that offers crypto CFDs or via a bridge EA connecting to exchanges). MT5's Python API can also stream tick data from the terminal into our Python environment if needed. Alternatively, CCXT can be polled for near-real-time prices (with some latency). The architecture will be flexible: the EA on MT5 can access price data directly from the trading server for low latency execution, while Python can subscribe to the same data via MT5 or a parallel API feed for analysis.

- **News & Sentiment Data:** We'll integrate one or two news APIs for sentiment analysis: for example, **NewsAPI.org** (aggregates headlines from many sources) and **CryptoPanic** (a crypto news aggregator with sentiment tags). We will schedule periodic fetches of news headlines and social media sentiment (if available via API), especially around significant market hours or events. The raw news data will be parsed (using `feedparser` for RSS or HTTP requests for JSON APIs) and then run through a sentiment scoring function (e.g., using a simple Natural Language Processing model or a sentiment lexicon). Each news item might produce a sentiment score between -1 (very negative) and +1 (very positive), which can be used as features for the trading signal logic.

- **Feature Engineering:** All raw data (prices, volume, news sentiment, possibly macro data if needed) will be processed into features. Technical indicators will be computed on price data streams – for example:

- *Moving Averages:* e.g., 50-period and 200-period moving averages for trend direction.
- *Momentum Indicators:* RSI (Relative Strength Index) for overbought/oversold conditions, MACD (Moving Average Convergence Divergence) for momentum shifts, Bollinger Bands for volatility.
- *Volatility Measures:* True range and Bollinger Band width to gauge volatility regimes.
- *Sentiment Features:* aggregated sentiment score over last N news items or a binary flag if a very positive/negative news was seen in last hour.

Feature computation will be implemented in a **utilities module** (e.g., `utils/features.py`). This module might use libraries like `pandas_ta` for convenience, but we will ensure we understand each feature's formula (for documentation and potential customization). For example, computing an RSI might look like:

```python
import pandas as pd

def compute_rsi(price_series: pd.Series, period: int = 14) -> pd.Series:
    delta = price_series.diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=period).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()
    rs = gain / loss
    rsi = 100 - (100 / (1 + rs))
    return rsi
```

(Example: RSI calculation on price series using Pandas)

- **Data Access Strategy:** Wherever possible, we use **free online data** directly (e.g., calling exchange APIs for current prices and NewsAPI for headlines). If an online source is not available or reliable enough, the system can fall back on locally stored data. For example, during backtesting or if an API limit is reached, we might use a cached dataset from `data/` directory. This two-tier approach (online first, local second) ensures reliability without incurring unnecessary data costs.

In summary, the data pipeline ensures that at any given time, the strategy has a rich, up-to-date set of features derived from both market data and sentiment, enabling well-informed trading decisions.

Trading Strategy & Signal Generation

The strategy module will produce trading signals (buy/sell/hold decisions) based on a combination of **rule-based logic** and optionally a **Machine Learning model** for predictive insight. We will incorporate well-known trading strategies as a baseline and then consider ML to adapt or enhance these strategies.

Rule-Based Strategies (Indicators): We will implement a set of popular indicator-based rules, such as:
- **Moving Average Crossover:** e.g., if a short-term MA (like 50-period) crosses above a long-term MA (200-period), generate a **buy** signal (trend bullish). A downward crossover would signal **sell**. Thresholds on how much separation is needed can be tuned.
- **RSI Overbought/Oversold:** if RSI drops below 30, it indicates an oversold market – trigger a potential **buy** (expecting mean reversion). If RSI exceeds 70 (overbought), trigger a **sell** or avoid new longs.
- **MACD Divergence:** use MACD line crossing above the signal line as a **bullish** signal, and vice versa for bearish. Also monitor if MACD histogram starts to tick up from negative territory as an early buy indication (and opposite for sell).
- **Bollinger Band Reversion:** if price touches the lower Bollinger Band while volatility is high and there's no bad news, it might indicate a **buy** opportunity (price too low relative to recent range); conversely touching the upper band might indicate **sell** or profit-taking.

Each of these rules will be coded in the `strategies/` module (for example, `strategies/indicator_signals.py`). The strategy can combine multiple conditions (e.g., take a trade only if **multiple indicators** align in the same direction, to reduce false signals).

Machine Learning Model (optional): In addition to or instead of pure rules, we may train an ML model to predict short-term price movements. For instance, a classification model could predict the probability of price going up by +X% vs down by –X% in the next N minutes. Key points for the ML approach:
- **Target Definition:** We need to define what the model predicts. One approach: a binary target for whether the price will rise by at least `r%` in the next `t` time horizon (e.g., next 30 minutes or 1 hour). Alternatively, predict a continuous outcome like the expected return or next period's price directly (regression).
- **Feature Set:** The input features would include technical indicators (the same ones used in rule-based strategies) over multiple time windows, recent price returns, volume changes, and even sentiment scores (e.g., average sentiment in last hour). This gives the model a holistic view of technical and fundamental state.
- **Model Choice:** We could start with a relatively simple model like a Random Forest or XGBoost (which often perform well on tabular data). If we have ample data, we might explore a deep learning model (like an LSTM for sequence data or a small neural network). However, complexity should be justified by performance gains, since simpler models are easier to interpret and less risky.
- **Training Procedure:** We will use a portion of historical data for training and set aside a test dataset for evaluation. To mimic live trading conditions, **walk-forward validation** will be used: train on an older period, test on a subsequent period, and repeat moving forward in time. This prevents lookahead bias.
- **Hyperparameter Tuning:** Tools like **Optuna** can help automate finding the best hyperparameters (e.g.,

tree depth, number of estimators for a Random Forest, or learning rate and layers for a neural network). Cross-validation on rolling windows of data will be applied to ensure the model generalizes.
- **Performance Metric:** We'll optimize the model for metrics correlated with trading success, such as accuracy of direction prediction, but more importantly **profit factor** or Sharpe ratio in backtesting (a model that's slightly less accurate but better at picking big wins over losses is preferable to one that's highly accurate on small moves but misses tail risks).

The ML model is **optional** in that the system will be functional with purely rule-based signals. The ML component can be integrated as an enhancement: for example, the rule-based strategy might trigger a trade *candidate*, and the ML model's output can be used to filter or confirm the trade. This hybrid approach leverages human insights (via rules) and data-driven adaptivity (via ML).

Signal Flow: In live trading, the Python part of the system will continuously generate signals based on the latest data and features. For example, every new tick or every minute: recalc indicators → check strategy rules → possibly call the ML model → produce a **signal object** (e.g., `{"symbol": "BTCUSD", "action": "BUY", "confidence": 0.8, "stop_loss": ..., "take_profit": ...}`). This signal will then be handed off to the execution engine.

System Architecture & Repository Structure

We will organize the project repository in a clear, modular structure to separate concerns and ensure maintainability. Below is the proposed repository layout and the role of each component:

```
hypertrader/
├── data/
│   ├── raw/              # Raw datasets or data dumps (historical prices, 
etc.)
│   └── processed/        # Processed data (feature matrices, cleaned datasets)
├── models/
│   ├── train.ipynb       # Jupyter notebook for model training experiments
│   └── saved_models/     # Serialized models (e.g., .h5 or .pkl files for ML 
models)
├── strategies/
│   ├── indicator_signals.py   # Functions for indicator-based rules
│   └── ml_strategy.py         # Code to load ML model and generate signals
├── execution/
│   ├── hypertrader.mq5    # MQL5 Expert Advisor code for live trading
│   └── bridge/           # (Optional) Python-MQL5 bridge code (e.g., ZeroMQ or 
file I/O)
├── utils/
│   ├── features.py       # Feature engineering utilities (technical 
indicators, etc.)
│   ├── sentiment.py      # Sentiment analysis and news fetching functions
│   └── config.py         # Configuration settings (API keys, thresholds, etc.)
├── tests/
```

```
│   ├── test_data.py       # Unit tests for data ingestion & integrity
│   ├── test_features.py   # Tests for feature calculations (indicators)
│   ├── test_strategy.py   # Tests for strategy logic (e.g., signals given mock data)
│   └── backtest.py        # Script or tests for backtesting the strategy on historical data
├── .github/
│   ├── workflows/
│   │   └── ci.yml         # GitHub Actions pipeline: linting, testing, deployment
│   ├── ISSUE_TEMPLATE.md  # Template for raising issues (bug reports, feature requests)
│   └── PULL_REQUEST_TEMPLATE.md  # Template to ensure informative PR descriptions
└── README.md              # Project overview, setup instructions, and usage guide
```

Folder/Module Descriptions:
- `data/`**:** Contains subfolders for raw and processed data. For example, raw CSV files downloaded from exchanges, and processed files like feature datasets ready for modeling. This separation helps in reproducibility (raw data is the source of truth; processed can be regenerated via code).
- `models/`**:** Contains the machine learning training notebook and saved model files. The `train.ipynb` will document the process of training any ML model (including data preprocessing and results). The final chosen model can be saved (e.g., as `model_final.pkl`) for the live system to load.
- `strategies/`**:** Houses the strategy logic. We separate indicator-based logic (`indicator_signals.py`) and ML-based logic (`ml_strategy.py`), but they will likely share features. For example, `indicator_signals.py` might have functions like `def moving_average_signal(data) -> Signal`, etc. These strategy functions can be called in combination or sequence by a master orchestration function that decides the final action.
- `execution/`**:** Contains everything related to executing trades on MetaTrader 5. The core is the **MQL5 Expert Advisor** code (`hypertrader.mq5`). This EA will run on the MT5 platform and is responsible for reading signals (from the Python side) and executing orders. We might also include a subfolder `bridge/` if we implement a communication bridge (for example, a Python script using ZeroMQ or writing to a file). If using ZeroMQ, we may include a lightweight library or code in MQL5 to connect to a local socket. The EA will handle order placement, modifications, and ensure that each trade obeys risk parameters (lot size calculation and attaching SL/TP).
- `utils/`**:** General utilities. `features.py` will have code to compute indicators and possibly a wrapper to get all features for a given symbol. `sentiment.py` will include functions to call news APIs (complete with API keys stored in config or environment variables), parse responses, and compute sentiment scores. We may also include `config.py` (or a YAML/JSON config file) to centralize configurable settings: e.g., list of symbols to trade, risk per trade, indicator thresholds, API keys, etc. This modular approach allows adjusting parameters without altering core logic.
- `tests/`**:** Contains automated tests. We will write **unit tests** for pure functions (like indicator calculations to ensure correctness) and **integration tests** for more complex interactions (like generating a signal from a sample data sequence, or simulating the EA logic on a dummy account). The `backtest.py` might not be a typical unit test; it could be a script that uses the strategy on historical data and reports performance

metrics (sort of an offline evaluation). This can be run periodically or in CI to guard against regressions in strategy performance.
- `.github/workflows/ci.yml`**:** This is a CI pipeline configuration (GitHub Actions). It will likely do steps like: set up Python environment, install dependencies, run linters (ensuring code style), run all tests, and perhaps compile the MQL5 EA code. For compiling the EA, we might use MetaTrader's command-line compiler (MetaEditor) on a Windows runner, or simply ensure the `.mq5` file has no errors by a static check. If possible, the CI could also package the EA (output `.ex5` file) and attach it as an artifact or deploy to the repo for easy download.
- **Issue/PR Templates:** We include these to ensure good project management practices. For example, the issue template might prompt for descriptions, reproduction steps for bugs, etc., and the PR template will remind contributors to describe what and why of changes, link issues, and note any checklist items (like tests added or documentation updated). This is part of making the repository "production-ready" and friendly to collaboration.
- `README.md`**:** The README will serve as the user's guide to the project – summarizing the strategy, how to set up and run the system, and perhaps example results from backtesting. It will also detail how to deploy the EA on MetaTrader, and any configuration required (like putting API keys in environment variables or config files).

This structured architecture ensures clarity: for instance, one can work on improving the ML model in isolation under `models/` without touching the execution code, or update the EA logic in `hypertrader.mq5` without altering how features are computed. The separation also aids testing and continuous integration, since each part can be tested on its own.

Step-by-Step Implementation Plan

To build the system, we will proceed in logical stages. Each step will be implemented as an atomic commit (with a descriptive message), ensuring the project is built up incrementally and tested along the way:

1. **Environment Setup:**
2. Initialize a Python virtual environment and create a `requirements.txt` listing all necessary Python packages (CCXT, pandas, etc.).
3. Ensure MetaTrader 5 is set up on the development machine or CI environment (including MetaEditor for compiling the EA).
4. Verify access to exchange APIs and news APIs (obtain API keys if needed, e.g., for NewsAPI.org).

5. *Commit example:* `"chore: initial project setup with Python env and dependencies"`.

6. **Data Ingestion Module:**

7. Create `utils/features.py` (or a new `data/fetch_data.py`) to fetch historical data via CCXT. Start with major pairs like BTC/USD and ETH/USD on at least one exchange (e.g., Binance). Include functions to fetch recent data as well.
8. Store fetched data in `data/raw/` as CSV for reproducibility.
9. Implement simple unit tests in `test_data.py` to ensure the data fetching returns expected columns and data ranges.

10. *Commit example:* `"feat: add data ingestion script for historical price data from exchanges"`.

11. **Feature Engineering Utilities:**

12. Implement functions to calculate key technical indicators in `utils/features.py`. Leverage libraries if available, otherwise implement manually as shown in earlier RSI example. Functions might include `compute_moving_average`, `compute_macd`, `compute_rsi`, etc., each returning either a pandas Series or a numeric value given recent data.
13. Add unit tests in `test_features.py` to validate these computations against known values or small sample data (for instance, compare our RSI output with an expected value on a tiny dataset).

14. *Commit example:* `"feat: implement technical indicator calculations (MA, RSI, MACD, BBands)"`.

15. **Sentiment Analysis Module:**

16. Create `utils/sentiment.py` with functions to fetch news. For example, `fetch_news_headlines()` that calls NewsAPI or CryptoPanic and returns a list of headlines with timestamps.
17. Implement a simple sentiment scoring function. Perhaps use a pre-trained sentiment model (like TextBlob or `vaderSentiment` for quick polarity scores) to score each headline. Alternatively, define a dictionary of positive/negative words for crypto (e.g., "hack" negative, "ETF approval" positive, etc.).
18. This module should output a summary sentiment metric that can be merged with the trading features (e.g., average sentiment in last 15 minutes, or a score of the most extreme recent news).
19. Add tests to ensure the news fetcher handles empty or error responses gracefully and that sentiment scoring yields values in expected range.

20. *Commit example:* `"feat: add news fetcher and sentiment analysis utility"`.

21. **Strategy Logic Implementation:**

22. Begin with the rule-based strategy in `strategies/indicator_signals.py`. Define functions for each rule (as mentioned earlier), and a main function `generate_signal(data_window)` that applies the rules to recent features and decides whether to buy, sell, or hold. This function can return a custom `Signal` object or simple dict (with fields like action, symbol, stop_loss, take_profit).
23. If multiple indicators are used, decide on a simple logic to combine them (e.g., all conditions must align, or weight them). This could be as straightforward as:

```python
if buy_condition1 and buy_condition2 and sentiment_score > 0:
    signal = "BUY"
elif sell_condition1 and sell_condition2 and sentiment_score < 0:
    signal = "SELL"
else:
    signal = "HOLD"
```

24. Next, integrate the **ML model** (if we proceed with one). Use `models/train.ipynb` to experiment and train a model on historical data. Once satisfied, save the model (e.g., `models/saved_models/model.pkl`). Then in `strategies/ml_strategy.py`, load this model and create a function `ml_signal(latest_features)` that returns a signal or confidence. This can be integrated into `generate_signal` (for example, only take a trade if the ML model confidence is above 60% for that direction).
25. Write tests in `test_strategy.py` to simulate scenarios. For example, feed a rising price sequence into `generate_signal` and check that a buy signal is emitted; feed a flat or choppy sequence and ensure mostly hold or occasional signals. If ML is included, test the `ml_signal` with some sample feature input to ensure the pipeline works (perhaps using a stub or simple trained model for test).
26. *Commit example:* `"feat: add trading strategy logic (indicator rules and ML hook)"`.
27. **Expert Advisor (Execution Engine) Development:**
28. Write the MQL5 EA in `execution/hypertrader.mq5`. This is a critical component that connects the Python strategy to actual trades on MT5. Key tasks for the EA:
    - **Initialization:** load configuration (symbol list, lot size or risk parameters). Possibly connect to Python if using a bridging method.
    - **OnTick or Timer Event:** Each time a new market tick comes in (or on a short timer), the EA should retrieve the latest trading signal from the Python side. How this is done depends on the chosen integration:
    - *File-Based:* The Python script could continuously write the latest signal to a file (e.g., `signal.json` or a CSV) in a shared directory. The EA can then read this file each tick to get `action` and `SL/TP` values.
    - *ZeroMQ Socket:* For a more real-time link, implement a ZeroMQ client in Python that sends signals, and in MQL5 use a DLL or known library (like the **dwx-zeromq connector** from Darwinex) to receive messages. This requires some setup and is more complex, but reduces latency.
    - *MT5 Python API:* Alternatively, run the Python script *inside* the same machine where MT5 is, and use MetaTrader's built-in Python integration to call Python functions or have Python connect via the `MetaTrader5` Python package to place orders directly. (However, using the EA to place orders might be simpler for control and risk logic.)
    - **Placing Orders:** If the signal says "BUY" or "SELL", the EA should compute the lot size based on the 2-3% risk rule. This involves calculating lot such that (lot * trade_value * stop_distance) ≈ 2% of equity. It can retrieve account equity via `AccountInfoDouble(ACCOUNT_BALANCE)` or similar, then use known formula for risk position sizing. For example, if equity is $10,000 and we risk 2% ($200) on a trade, and stop-loss is 5% away from entry, then the position size should be $200/0.05 = $4,000 worth of the asset (which corresponds to lot size = 0.4 if 1.0 lot = $10,000 for the asset).
    - **Order Execution:** Use the MQL5 trade functions to send orders. For example, using the `CTrade` class:

```
// Pseudo-code inside hypertrader.mq5
#include <Trade/Trade.mqh>
CTrade trade;
void OnTick() {
    Signal sig = ReadSignalFromFile();  // custom function to get
Python signal
    if(sig.action == "BUY") {
        double sl_price = sig.stop_loss;
        double tp_price = sig.take_profit;
        trade.Buy(sig.volume, _Symbol, 0, sl_price, tp_price);
    } else if(sig.action == "SELL") {
        // similarly, place a sell order
        trade.Sell(sig.volume, _Symbol, 0, sl_price, tp_price);
    }
    // No action on "HOLD" signals.
}
```

(Example snippet: EA reading a signal and executing a Buy. The volume (lot size) and SL/TP come from the strategy signal or are calculated within the EA.)

- **Trade Management:** After opening a trade, the EA will rely on the built-in stop-loss and take-profit for exits, but we can also implement a **trailing stop** mechanism: on each tick, if a trade is in profit above a threshold, adjust its SL to lock in profits. Additionally, ensure only one trade per symbol if desired (or allow multiple if strategy calls for scaling, depending on netting vs hedging mode in MT5).
- **Safety Checks:** Include checks to avoid duplicate orders on the same signal (perhaps tag each signal with an ID or timestamp and store the last processed ID). Also handle cases like market closed or no price (shouldn't happen with crypto, since 24/7, but in case of connection loss).
- **Logging:** Within the EA, use `Print()` statements to log important events to the MT5 log (which can later be monitored). For instance, log when a trade is opened or closed, and any error returns from trade operations.

29. Testing the EA is a bit tricky but we will do a combination of strategy tester runs (manually in MT5) and integration tests in Python by simulating the signals and making sure the file/bridge communication works. For example, an integration test can drop a fake `signal.json` and then call a debug function in the EA (through MT5 Strategy Tester) to see if it picks it up. We may also create a **mock trading environment** in Python that mimics the EA's risk calculations to verify our formulas yield expected lot sizes.

30. *Commit example:* `"feat: add MetaTrader5 Expert Advisor for execution and bridging with Python signals"`.

31. **Backtesting Module:**

32. Using the historical data and strategies, implement a backtesting script (`tests/backtest.py`). This script will load historical price data, step through it (candle by candle or tick by tick), and apply the strategy logic (the same `generate_signal` function) to simulate trades. We can use **Backtrader** for a full-featured solution: wrap our strategy as a Backtrader strategy class, or simply manually simulate trades in a loop for our custom logic.
33. Compute performance metrics from the backtest: total return, max drawdown, Sharpe ratio, win rate, etc., and output these. This will help in evaluating strategy variants and ensuring the system meets the objectives.
34. The backtest can be run as part of tests (though it might be too slow for every CI run if using a lot of data; perhaps run on a subset in CI).
35. *Commit example:* `"feat: add backtesting harness and evaluate baseline strategy performance"`.
36. **Continuous Integration & Deployment Setup:**
37. Configure GitHub Actions in `.github/workflows/ci.yml`. For example, use a matrix to test on multiple OS (at least Ubuntu for Python tests, and Windows for compiling the EA). Steps include:
    - Checkout code, set up Python, install dependencies.
    - Run `flake8` or `pylint` to ensure code style.
    - Run `pytest` to execute our tests.
    - On a Windows runner: run the MetaTrader5 MetaEditor in command-line to compile `hypertrader.mq5` (MetaEditor has a `/compile` flag). This will catch any syntax errors in MQL5 at least. Save the compiled `.ex5` file as an artifact or output.
38. Set up repository **branch protections** (if in scope) so that all tests must pass before merge. The PR template will remind reviewers to check that CI passes.
39. If deployment is to be automated: possibly add a step to the CI that, on pushing a version tag, it connects to the VPS (maybe via FTP/SSH) and updates the EA or restarts the bot. This might be a stretch goal; initially, deployment can be manual (copying the EA file to the server and running the Python script).
40. *Commit example:* `"chore: add GitHub Actions CI pipeline for testing and compilation"`.
41. **Documentation & Repository Polish:**
42. Write a comprehensive `README.md` (if not already drafted) covering how to install and run the system. Include configuration instructions (like how to set up the MT5 terminal, where to put the EA file, and how to supply API keys securely via environment variables or a config file).
43. Include usage examples, e.g., how to run the backtest script, and sample output of a trade. Possibly add diagrams or workflow visuals for clarity.
44. Ensure the issue/PR templates are in place to guide future contributions.
45. *Commit example:* `"docs: add README with setup, usage, and contribution guidelines"`.

Throughout the implementation, we will commit frequently with clear messages as illustrated. This not only tracks progress but also makes code reviews easier by isolating each feature. We will also open Pull

Requests for major features (data ingestion, strategy, EA, etc.), even if working solo, to document the rationale and get feedback (if any collaborators or future maintainers).

Risk Management & Performance Tuning

Managing risk is a cornerstone of this trading system, and we'll enforce strict controls to protect the trading account:

- **Position Sizing & Risk Cap:** Each trade will risk at most **2–3% of account equity**. We'll implement a function to calculate lot size based on this risk and the trade's stop-loss distance. For example, if the account equity is $10,000 and risk per trade is 2% ($200), and the strategy indicates a stop-loss 5% away from entry price, the volume is set such that 5% move causes $200 loss. This dynamic sizing means positions are smaller for wider stops and larger for tighter stops, keeping risk constant. We will also enforce a minimum lot (as exchanges/brokers require) and maybe a max cap for extremely volatile conditions.

- **Stop Loss & Take Profit:** Every order sent by the EA will include a **stop-loss (SL)** and **take-profit (TP)** level as determined by the strategy. For instance, the strategy might set SL at the last swing low (for a buy) or a fixed percentage. TP could be based on a risk:reward ratio (e.g. 2:1 reward-to-risk) or a technical level (like nearest resistance/support). By setting these immediately at order execution, we guard against sudden adverse moves or losing network connectivity.

- **Trailing Stop (Optional):** The EA can adjust the stop-loss once a trade is in profit by a certain amount. For example, after 1% in profit, move SL to breakeven; after 2% profit, trail the SL 1% behind current price. This helps lock in profits on winning trades in trending markets. This feature will be configurable (can be turned on/off or adjusted).

- **Multiple Positions & Exposure:** We will decide whether the EA can have multiple positions simultaneously. Likely, to limit risk, the system will **only hold one position per symbol** (no hedging or stacking trades on the same pair) and possibly limit total open risk across all symbols (e.g., if trading multiple pairs, ensure combined risk doesn't exceed, say, 5% at any time). If one trade is open and another signal comes for a different symbol, it can be taken as long as it adheres to the risk cap. If another signal comes for the *same* symbol in the opposite direction, the EA might ignore it or use it as a signal to exit/flip the trade depending on strategy design.

- **Monitoring & Alerts:** To keep track of performance and any issues:

- The system will maintain a log of each trade (entry time, price, size, exit time, P/L, reason for exit). This could be simply written to a CSV or database. This log helps in post-analysis and auditing the strategy behavior over time.

- For live monitoring, we plan to integrate with a tool like **Prometheus/Grafana**. For example, the Python part could expose a Prometheus metric (via an HTTP endpoint) for things like current portfolio value, number of open trades, latency of last signal, etc. A Grafana dashboard can then plot these in real-time. This goes beyond essential requirements, but is valuable for a production system. At minimum, we'll set up email or Telegram alerts for critical events (like if a trade is executed, or if an error/exception occurs in the Python script or EA).

- **Performance Optimization:** Given the system may execute rapidly, we must ensure efficiency:

- Python computations (indicators, model inference) should be vectorized where possible (using NumPy/Pandas) to handle incoming ticks without lag. If using an ML model, loading it into memory once and reusing it for inference each time avoids reload overhead.
- The communication between Python and MQL should be as lightweight as possible. For instance, if using file-based signals, ensure the file I/O is minimal (perhaps just a single small JSON). If using sockets, keep messages concise.

- The EA code in MQL5 should execute quickly on each tick – avoid heavy loops or blocking operations. MQL5 is fast for trading operations, but any waiting for Python should be asynchronous if possible. In worst case, if Python becomes slow or unresponsive, the EA should have a timeout or default behavior (e.g., if no new signal received, either do nothing or maybe close positions if that's part of fail-safe).

- **Resilience & Error Handling:** Anticipate things like network outages, API failures, or broker server issues. The system should handle these gracefully: for example, if price feed is lost temporarily, the EA might pause trading (and not send blind orders). If the Python script crashes or becomes unresponsive, the EA could either keep running with last known strategy state or close out to avoid unmanaged positions. We will incorporate retries for API calls, and sanity checks on data (e.g., if an indicator returns NaN or extreme value, ignore that cycle).

- **Regulatory and Compliance Considerations:** Since this system trades cryptocurrencies, traditional regulatory constraints are limited (no pattern day trading rule, etc.), but we must ensure compliance with exchange terms of service and any regional restrictions. For example, if using Binance API, adhere to rate limits and required authentication. Also, if the system is scaled to manage others' money, there might be licensing considerations (outside our current scope). For now, we assume this is proprietary trading on the user's own account, so the main rule is to trade safely within the account's risk tolerance.

By incorporating these risk management measures, the strategy will not only aim for profit but also safeguard against catastrophic losses and unpredictable market events. Fine-tuning will continue during testing – for instance, adjusting the 2% risk to 1% if we find the strategy performs better with smaller positions, or tightening stop-losses if drawdowns are higher than expected.

Conclusion & Next Steps

With the above plan, we have a roadmap to develop the hyper-trading automation system in a methodical way. The project emphasizes reliability, clarity, and safety at every step, from using robust data sources to enforcing risk limits. The next steps would be to address a few open questions and then proceed to coding:

- **Exchange & News API Choices:** We will use **Binance** and **Coinbase Pro** as primary exchanges via CCXT (both have high uptime and liquidity). For news, **NewsAPI.org** (for general crypto news) and **CryptoPanic** (for aggregated sentiment) will be integrated. These choices balance breadth of coverage with reliable free access.
- **Server/Hosting Environment:** The EA will run on a **Windows VPS** (since MT5 is a Windows application). A popular choice is an AWS EC2 Windows instance or a specialized Forex VPS with low

latency. The Python components can run on the same machine (ensuring fast local communication with MT5). We'll schedule the Python script to start with the OS (so that if the VPS reboots, the trading system comes back online).

- **Clarifying Risk Limits:** Aside from the 2-3% per trade, no additional specific risk limits were given, so we will implement that as the primary rule. Of course, the user can configure this percentage. We also assume no leverage beyond what the broker account uses by default, and we won't engage in anything exotic like margin borrowing outside the account's scope.

If any of the above choices need adjustment (for instance, a different exchange or a different hosting setup), we should clarify that before coding. Otherwise, the plan is clear. We will proceed to implement each module as outlined, ensuring to document and test along the way, ultimately delivering a robust trading bot repository ready for real-world deployment.
