BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
Faculty of Economics and Business Administration
Business Informations Systems

Bachelor's Thesis

Supervisor,

Associate Prof. Dénes CSALA, PhD
Associate Prof. Mihai Constantin AVORNICULUI, PhD

Graduate,
KIS-JUHÁSZ István

2025

BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
Faculty of Economics and Business Administration
Business Informations Systems

Bachelor's Thesis

Web-Based Platform for Backtesting Multi-Indicator Trading Strategies

Supervisor,

Associate Prof. Dénes CSALA, PhD
Associate Prof. Mihai Constantin AVORNICULUI, PhD

Graduate,
KIS-JUHÁSZ István

2025

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
Facultatea de Științe Economice și
Gestiunea Afacerilor
Informatică Economică

Lucrare de Licență

Platformă web pentru evaluarea strategiilor de tranzacționare bazate pe indicatori multipli

Coordonator științific,

Lect. univ. dr. CSALA Dénes
Conf. univ. dr. Mihai Constantin AVORNICULUI

Absolvent,
KIS-JUHÁSZ István

2025

BABEŞ-BOLYAI TUDOMÁNYEGYETEM
Kolozsvár
Közgazdaság- és Gazdálkodástudományi Kar
Gazdasági Informatika

Szakdolgozat

Webalapú rendszer több indikátoron alapuló kereskedési stratégiák hátultesztelésére

Témavezető,

Dr. CSALA Dénes egyetemi adjunktus
Dr. Mihai Constantin AVORNICULUI egyetemi docens

Végzős hallgató,
KIS-JUHÁSZ István

2025

Abstract

This thesis presents the development of a web-based platform designed to backtest trading strategies that combine multiple technical indicators. The project aims to provide users with an accessible interface through which they can evaluate the historical performance of algorithmic strategies without financial risk. The platform is built around well-established indicators, including MACD Platinum, QQE Advanced, and QMP Filter, each of which is customizable by the user. This flexibility enables traders to tailor strategies based on their insights and preferences rather than relying on rigid, predefined models. Users can select currency pairs and timeframes, and the system executes backtesting using historical Binance market data. The results are visualized through intuitive charts and performance metrics, facilitating informed decision making and comparative analysis. The work highlights the importance of customized strategy testing and provides an opportunity for future extensions.

Összefoglaló

Dolgozatom egy webalapú platform fejlesztését mutatja be, amely lehetőséget nyújt több technikai indikátoron alapuló kereskedési stratégiák utólagos tesztelésére. A cél egy olyan felület létrehozása, amely lehetővé teszi a felhasználók számára, hogy pénzügyi kockázat nélkül elemezhessék algoritmikus stratégiáik múltbeli teljesítményét. A rendszer integrálja a MACD Platinum, a QQE Advanced és a QMP Filter indikátorokat, és testreszabható paramétereket kínál a stratégiai viselkedés finomhangolásához. A felhasználók választhatnak devizapárokat és kereskedési frekvenciákat, majd a platform a Binance történeti adatai alapján hajtja végre a tesztelést. Az eredmények grafikonokon és teljesítménymutatókon keresztül jelennek meg, segítve a döntéshozatalt és az összehasonlítást. A dolgozat hangsúlyozza a visszatesztelés fontosságát a pénzügyi technológia terén, valamint bemutatja a rendszer modularitását és jövőbeli bővíthetőségét.

Contents

Abbreviations

MACD	Moving Average Convergence Divergence
QQE	Quantitative Qualitative Estimation
QMP	Quick Momentum Pattern
EMA	Exponential Moving Average
BTC	Bitcoin
ETH	Ethereum
USDT	Tether USD
API	Application Programming Interface
UI	User Interface
UX	User Experience
SDK	Software Development Kit
EVM	Ethereum Virtual Machine
CLI	Command Line Interface
DOM	Document Object Model
JSON	JavaScript Object Notation
ROI	Return on Investment
P&L	Profit and Loss
BSC	Binance Smart Chain
JS	JavaScript
TS	TypeScript
OHLCV	Open, High, Low, Close, Volume (candlestick charting)

Chapter 1

Introduction: Algorithmic Trading and Strategy Evaluation

1.1 What is Algorithmic Trading?

Algorithmic trading refers to the use of computer programs that automatically execute trades by following a set of programmed criteria. These criteria often consider elements like market price, trade volume, timing, and algorithmic models to ensure efficiency and objectivity in transactions. Unlike traditional manual trading methods, algorithmic trading leverages computational power to ensure rapid, efficient, and unbiased market transactions.

The origins of algorithmic trading trace back to the early 1970s with the introduction of electronic order routing systems like the New York Stock Exchange's DOT system. However, it wasn't until the late 1980s and early 1990s, with the advent of the internet and electronic communication networks, that algorithmic trading began to gain significant traction .

In recent years, the prevalence of algorithmic trading has increased. In 2024, algorithmic trading is estimated to account for approximately 60–75% of the trading volume in developed financial markets, including the US stock market (Trader, 2024).

Figure 1.1 illustrates the evolution of trading volume shares across various market participants from 2010 to 2025. Market makers have consistently dominated with the highest share, while retail and fundamental funds show more fluctuation. Notably, the proportion of trades from quant funds has gradually increased, reflecting the growing role of algorithmic and data-driven strategies in modern markets. This visual trend supports the argument that algorithmic trading has become an integral part of the market structure.

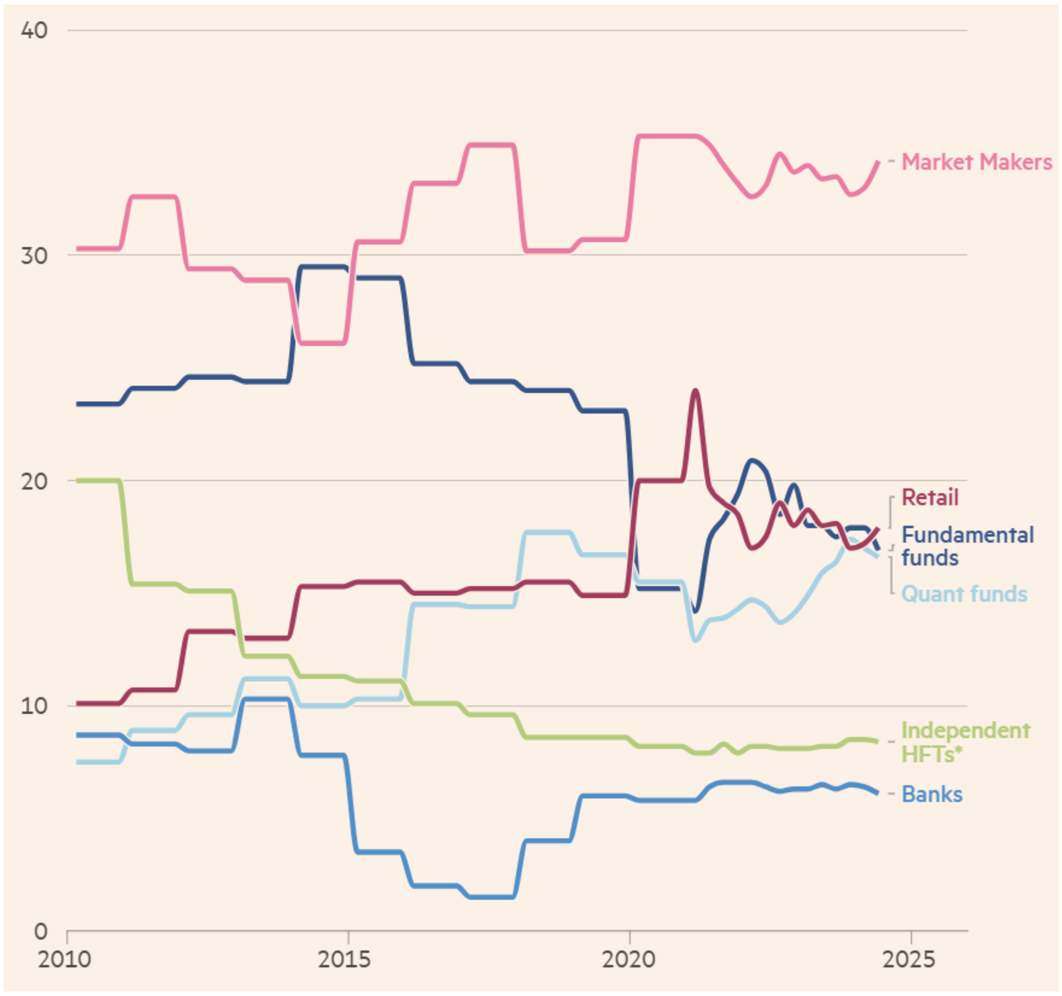


Figure 1.1: US equity trading volume by market participant (%), (Financial Times, 2024)

1.2 Importance of Backtesting

Backtesting is a crucial step in formulating trading strategies, allowing for the simulation of performance using historical market data. Through this retrospective analysis, traders can evaluate the practicality of their approach, uncover potential flaws, and refine parameters prior to live implementation. This method not only offers insight into how a strategy might have performed but also serves as a safeguard against deploying systems that are merely optimized for given currencies or timeframes.

Incorporating advanced machine learning methods and sentiment analysis into backtesting procedures enhances robustness and adaptability to market volatility (Frattini et al., 2022).

These innovative methods support the development of strategies that are both more robust and responsive to real-time market dynamics, enhancing their potential effectiveness in live trading environments.

1.3 Role of Technical Indicators

In the realm of financial analysis, technical indicators are fundamental tools used to identify price dynamics and analyze market trends based on historical data. Indicators such as the Moving Average Convergence Divergence (MACD) (Investopedia, a), Relative Strength Index (RSI) (Investopedia, b), and Bollinger Bands are particularly valuable for assessing momentum, volatility, and potential turning points in the market. These tools form the core of many algorithmic trading strategies. The trading system used in this research applies these indicators to generate consistent buy and sell signals during both backtesting and live simulation phases. As a result, decision-making becomes more systematic and less dependent on subjective interpretation.

1.4 Common Backtesting Challenges

Backtesting is a critical component in the development of algorithmic trading strategies, allowing developers to simulate and evaluate performance using historical data. Techniques such as walk-forward testing and avoiding overfitting are emphasized in the practical algorithmic trading literature (Chan Ernie, 2013). However, there are several pitfalls that can compromise the accuracy and reliability of the backtest results:

- **Overfitting:** A strategy may be excessively tailored to historical data, capturing noise rather than robust patterns. Although this can yield excellent results in backtesting, such strategies often fail under live market conditions. Techniques like walk-through testing and keeping strategies simple help mitigate this risk.
- **Data Quality Issues:** Inaccurate or incomplete data (e.g. missing ticks, wrong timestamps) can mislead backtest results. Ensuring data cleanliness and precision is crucial for realistic performance evaluation.
- **Ignoring Transaction Costs:** Commissions, spreads, and slippage are often overlooked in simulations. However, these can significantly reduce real-world profitability. Backtests should incorporate realistic cost assumptions.
- **Assuming Perfect Execution:** Many backtests assume full order fills with no delay, which is unrealistic. In practice, order execution is subject to latency,

slippage, and partial fills. These factors should be modeled or at least considered to avoid inflated expectations.

- **Lack of Market Regime Testing:** Strategies tested only in trending or bullish markets may underperform during sideways or bearish conditions. Testing strategies across multiple market regimes increases their robustness.
- **No Consideration of Psychological Factors:** While not directly part of code, backtesting cannot replicate emotional responses like fear or greed. Forward testing or paper trading can help identify psychological weaknesses before deploying capital.

In the second part of the project, right after the backtesting, strong emphasis was placed on identifying and reducing the impact of these challenges. Particular attention was given to filtering out results that may be compromised or misleading due to factors such as overfitting, poor data quality, or unrealistic assumptions. This ensures that only robust and reliable strategies are considered for further development or real-time testing.

1.5 Project Goals and Motivation

This thesis aims to design a flexible, web-based, multi-phase testing platform that connects to live Binance data via API integration. In the first phase, the system performs backtesting across multiple cryptocurrency pairs and timeframes using historical data. Once initial backtests indicate promising strategies based on the win/loss ratio of the trades, the system moves into a second phase: live simulation via paper trading. This step allows a strategy's real-time behavior to be observed under dynamic market conditions — all without risking capital. Real-time SimTrade is a core component of strategy development, as historical data alone cannot guarantee a strategy's effectiveness in the current market.

Finally, the next steps in the near future are to fine-tune the results and interpret them using AI-based analytical tools to enhance strategic precision. The ultimate goal of this research is to uncover combinations of crypto pairs and timeframes where the MACD Platinum strategy (Jim Brown, 2013) consistently demonstrates practical potential. The role of the analytical tools is to filter out noise from trades: to reduce the number of potential trades while simultaneously increasing the win/loss ratio, thereby achieving a better-performing strategy.

Chapter 2

System Goals, Strategy Design and Architecture

2.1 System Objective and User Value

The purpose of the developed platform is to enable users to evaluate algorithmic trading strategies through a structured, two-phase testing pipeline.

In the first phase, users can configure a backtesting routine by selecting from available cryptocurrency pairs and timeframes. Once configured, the system executes the strategy over historical market data obtained from the Binance exchange. The evaluation is designed for speed, providing users with immediate feedback on the potential viability—or limitations—of the selected strategy under past market conditions.

In the second phase, strategies that demonstrate promise in the backtesting phase are subjected to live paper trading. Here, the strategy is executed in real time using current market data, but without risking actual capital. This enables performance evaluation under dynamic and unpredictable market conditions.

By decoupling the process of strategy development from real-world financial exposure, the platform bridges the gap between conceptual strategy design and practical testing. It provides a user-friendly interface (UI), leverages fast data access via REST API integration, and delivers performance metrics in real time. Ultimately, this architecture facilitates more informed and risk-conscious decision-making for algorithmic traders.

2.2 Trading Strategy Logic

The trading engine utilizes a multi-indicator approach to enhance the quality of generated buy/sell signals. The core logic is based on a combination of three technical indicators—each chosen for its complementary properties related to trend detection, momentum, and volatility filtering.

- **MACD Platinum:** A momentum indicator that improves upon the classical MACD by adding histogram dynamics to emphasize trend strength and reduce noise at crossover points.
- **QQE Advanced:** This refined version of the Quantitative Qualitative Estimation indicator introduces smoothing and adaptive thresholds to stabilize RSI-based signals and reduce false positives.
- **QMP Filter:** The Quick Momentum Pattern filter identifies areas of compression and filters out trades during sideways or erratic price conditions, thus avoiding false entries.

These indicators are applied in conjunction, and a signal is only generated when all conditions align. This improves reliability and reduces overtrading in noisy markets.

The platform is structured into distinct functional modules that implement this logic cohesively. Table 2.1 outlines the key components and associated technologies.

Each module operates autonomously but communicates through clearly defined APIs and shared data structures. This modularity ensures that individual components can be extended or replaced without disrupting the system's overall architecture.

2.3 Platform Architecture

The overall system architecture follows a three-tier structure designed for clarity, performance, and scalability. The layers include:

- **Frontend (UI/UX):** Developed in React and styled with Tailwind CSS, the user interface allows traders to easily configure strategies, select parameters, and interpret results.
- **Backend (Application Logic):** Implemented in Python using the Flask framework, the backend receives user requests, connects to Binance via their public REST API interface (Binance), runs the strategy logic, and returns backtest or trade results.
- **Data Layer:** This includes both the external data source—Binance (via REST API)—and the local storage format (CSV files), which persist results for auditability and visualization.

Table 2.1: System Modules and Associated Technologies

Module	**Technology**	**Function**
User Interface (Frontend)	React + Tailwind CSS	Provides a responsive, web-based UI that enables users to configure backtests, select currency pairs/timeframes, and visualize performance metrics.
Backtesting Engine	Python	Executes the trading strategy (MACD Platinum with QQE Advanced and QMP Filter) on historical data to generate performance summaries and trade logs.
Binance API Integration	`ccxt` Python library	Connects to Binance REST API to fetch OHLCV data and real-time price feeds for backtesting and live trading purposes.
Result Storage	CSV Format	Saves generated trades, P&L metrics, and accuracy statistics for reproducibility and auditability.
Paper Trading Module	Python + Threading	Simulates real-time trades using live market data without risking capital, allowing safe testing of strategy behavior under actual conditions.
AI-Based Analysis (planned)	Python + DeepSee AI	Future module intended to enhance performance tuning using pattern recognition, recurrence analysis, and anomaly detection.

2.4 Backtesting vs. Live Simulation

The platform distinguishes between two critical stages of strategy evaluation:

- **Backtesting:** Simulates historical trades using OHLCV (Open-High-Low-Close-Volume) price data. This helps determine how the strategy would have performed under specific market conditions in the past.

- **Live Simulation (Paper Trading):** Runs the same strategy in real time using current market data, but without executing real trades. This exposes the algorithm to actual market dynamics such as latency and volatility, providing insight into its real-world performance.

Both evaluation modes are essential: backtesting confirms theoretical potential, while live simulation tests the strategy's behavior under real-time conditions.

2.5 Design Constraints and Technology Choices

The system was designed under the following guiding principles:

- **Simplicity and Speed:** A lightweight UI and optimized backend ensure fast simulation and a frictionless user experience.
- **Modularity:** Functional separation allows for independent development and testing of components, enabling easy future upgrades such as additional strategies or data providers.
- **Extensibility:** The architecture supports integration with external tools (e.g., AI-based analyzers) and deployment pipelines (e.g., CI/CD), and can be extended to other chains such as the Binance Smart Chain (BSC).
- **Traceability:** All simulation and trading results are logged in structured CSV files. These logs include timestamps, entry/exit prices, and performance statistics such as ROI and P&L, ensuring reproducibility and post-analysis capabilities.

This strategic combination of structure, performance, and transparency ensures the system is suitable for academic experimentation as well as future production deployment.

Chapter 3

System Implementation

3.1 System Overview

The implemented system is a full-stack web application that supports the development, evaluation, and simulation of algorithmic trading strategies. It enables users to configure historical backtests across multiple cryptocurrency pairs and timeframes, and then simulate live trades based on real-time data without risking actual funds. This section outlines the system's architecture, modules, and operational flow.

3.1.1 High-Level Architecture

The platform adheres to a modular, three-tier architecture:

- **Frontend (React + Chart.js):** A dynamic interface where users select currency pairs and time intervals, trigger backtests, and review results through interactive charts.
- **Backend (Flask + ccxt):** The core logic layer that executes trading strategies, handles REST API requests, and interacts with Binance data sources.
- **Data Layer (CSV + Binance API):** Combines external data fetching from Binance via `ccxt` with local CSV file-based storage for logs and metrics.

This structure ensures clean separation of concerns, allowing the application to be easily extended or modified.

3.1.2 User Workflow

The complete user interaction is divided into three main phases:

1. **Backtesting Phase:** Users configure tests by selecting trading pairs and timeframes. Upon initiating a backtest, the backend evaluates the strategy using historical OHLCV data and returns accuracy and trade logs.

2. **Live Simulation Phase:** Users activate live trading for a selected symbol. The system then begins fetching real-time data, applying strategy logic continuously and logging trades under simulated paper-trading conditions.

3. **Visualization and Review:** Both historical and live trades are visualized using interactive charts. Users can zoom into trades, view logs, and monitor current simulated positions.

3.1.3 Component Overview

Each part of the system is encapsulated in a specific module. Table 3.1 lists the primary implementation units and their responsibilities.

Table 3.1: Component Overview and Responsibilities

Module	**Description**
`launcher.py`	Python GUI (Tkinter-based) launcher that starts both backend and frontend using subprocess automation.
`app.py`	Flask server exposing REST endpoints for backtesting, simulation control, and log retrieval.
`backtester.py`	Executes historical strategy backtests with indicator logic and multi-threaded symbol-timeframe evaluation.
`live_trader.py`	Runs real-time paper trading with stateful wallet and threaded strategy execution.
`StrategyDashboard.jsx`	React component for backtest configuration, execution, and trade result visualization.
`LiveTrading.jsx`	React component for launching live simulations and monitoring real-time trading behavior.

3.1.4 Data Flow During Backtesting

The communication flow during a typical backtest is as follows:

1. User configures and initiates a test in the frontend UI.

2. The frontend sends a `POST` request to `/run-backtest` via Flask.

3. The backend triggers `parallel_backtest()` which evaluates all combinations concurrently.
4. Trade logs are written to `trade_logs/` with timestamped CSV filenames.
5. The backend returns summary metrics and filenames for visualization.

3.1.5 Conclusion

This section has introduced the software stack and modular layout of the application. Each part of the system, from frontend interactivity to backend computation and data persistence, is implemented with clarity and modularity in mind. The following sections detail the core backend mechanics, trading logic, and strategy evaluation methods.

3.2 Backend Architecture

The backend forms the logical core of the system. It is implemented using the Flask web framework in Python and is responsible for handling user requests, performing backtesting, managing live simulation states, and serving data to the frontend via RESTful endpoints. This section outlines the structure, logic, and execution flow of the backend.

3.2.1 Project Structure and Modules

The backend resides within the `backend/` directory and includes three key Python modules:

- `app.py`: Launches the Flask server and exposes API endpoints.
- `backtester.py`: Handles historical backtesting of strategies and outputs results to CSV.
- `live_trader.py`: Manages real-time trading simulation using live market data.

Each module serves a specific responsibility and communicates through function calls and shared data conventions. The file structure encourages modularity and separation of concerns, which makes the system easier to maintain and expand.

3.2.2 Flask-Based REST API Design

The `app.py` module acts as the main controller. It receives JSON requests from the frontend and returns structured JSON responses after completing computations. Table 3.2 summarizes the primary API endpoints exposed by the backend.

Table 3.2: Primary Backend API Endpoints

Endpoint	**Functionality**
`POST /run-backtest`	Accepts trading pairs and timeframes, executes strategy, saves and returns summary logs.
`GET /backtest-history`	Lists previously run backtest log files and their metrics.
`GET /read-log/<filename>`	Reads and returns the contents of a specific backtest log file.
`POST /live/start`	Starts real-time trading simulation for a given symbol and timeframe.
`POST /live/stop`	Stops a currently running simulation for the specified symbol.
`GET /live/status`	Returns the status of the simulated wallet, trade history, and active positions.
`GET /live-logs, /live-log/<filename>`	Lists or downloads paper trading logs from the live simulation engine.

3.2.3 Example: Backtest Execution Endpoint

One of the most frequently used routes is the backtest trigger. The following code (abridged) shows how `/run-backtest` is implemented in Flask:

```python
@app.route("/run-backtest", methods=["POST"])
def run_backtest():
    data = request.get_json()
    symbols = data.get("symbols", [])
    timeframes = data.get("timeframes", [])
    results = parallel_backtest(symbols, timeframes)

    out = []
    for df, summary in results:
        if not df.empty:
            ts = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
            fname = f"{ts}_{summary['symbol'].replace('/','')}
             ↪  _{summary['timeframe']}.csv"
            df.to_csv(os.path.join("trade_logs", fname),
             ↪  index=False)
            summary.update({"file": fname, "run_id": ts})
        else:
            summary.update({"file": None, "run_id": None})
        out.append(summary)
    return jsonify(out)
```

This design allows the frontend to initiate a batch test, receive results, and instantly populate visual outputs for user review.

3.2.4 Parallel Computation of Tests

To reduce latency and support batch processing, the backend utilizes Python's `ThreadPoolExecu` to run multiple backtests concurrently. This enables multi-symbol, multi-timeframe evaluation in real-time.

```
def parallel_backtest(symbols, timeframes, limit=300):
    tasks, results = [], []
    with ThreadPoolExecutor() as executor:
        for s in symbols:
            for tf in timeframes:
                tasks.append(executor.submit(backtest_with_log,
                 ↪  s, tf, limit))
        for fut in as_completed(tasks):
            results.append(fut.result())
    return results
```

This multithreaded design improves responsiveness, especially when multiple combinations are selected from the frontend UI.

3.2.5 Executable Launcher Integration

To enhance usability, the system includes a graphical launcher interface implemented in `launcher.py`, which automates the startup of both backend and frontend environments. It is built using `tkinter` and compiled into a Windows executable via `PyInstaller`.

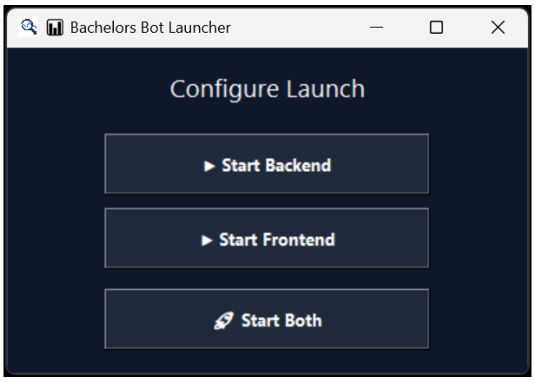
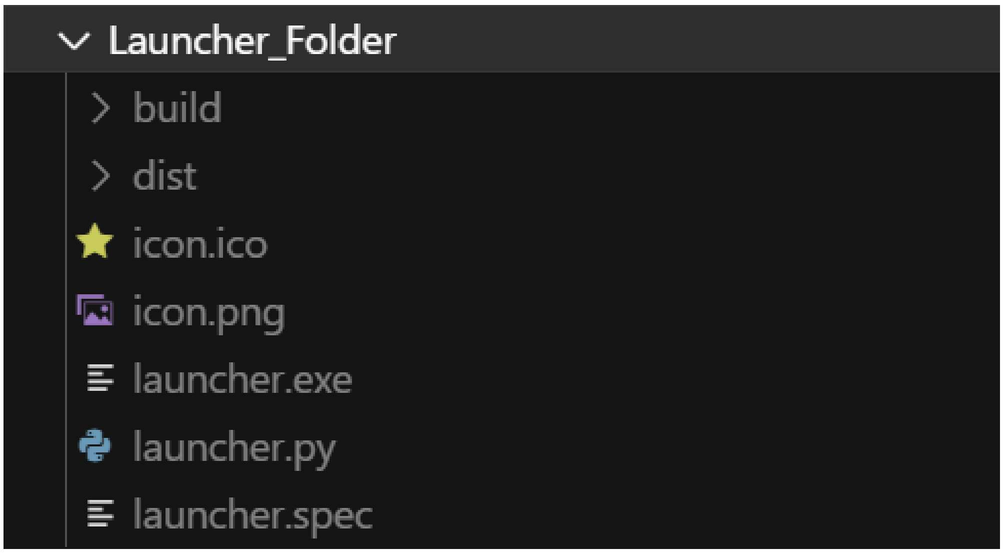


Figure 3.1: Executable launcher with options to start backend, frontend, or both.

The GUI includes three buttons:

- **Start Backend**: Executes the backend server in a new terminal window.
- **Start Frontend**: Runs the React development server.
- **Start Both**: Launches both components sequentially with a delay.



```
Launcher_Folder
  build
  dist
  icon.ico
  icon.png
  launcher.exe
  launcher.py
  launcher.spec
```

Figure 3.2: Folder structure of the packaged launcher application.

By compiling this script into a `.exe` file, technical overhead for launching the system is significantly reduced—especially useful during testing sessions or for non-technical users.

3.2.6 Conclusion

The backend is implemented as a scalable, modular Python application with RESTful interfaces and multithreaded evaluation capabilities. It communicates efficiently with the frontend through JSON APIs and handles all logic related to strategy execution, stateful simulations, and trade logging. Its integration with a graphical launcher improves the overall user experience by making the system accessible without requiring command-line knowledge.

3.3 Strategy Engine and Trading Logic

The strategy engine defines the rules for generating buy and sell signals based on technical indicators. These rules are applied both in historical simulations (backtesting) and in real-time trading sessions (paper trading). This section presents the full implementation of the strategy, how signals are generated, and how they are integrated into both execution engines.

3.3.1 Overview of Strategy Logic

The implemented strategy combines two classical indicators: **MACD** (Moving Average Convergence Divergence) and **RSI** (Relative Strength Index). These are applied to each OHLCV candle to generate buy/sell signals. The key decision logic is:

- **Buy Signal:** Triggered when MACD crosses above the signal line *and* RSI exceeds 50.
- **Sell Signal:** Triggered when MACD crosses below the signal line *and* RSI falls below 50.

This dual-condition rule attempts to capture both trend momentum and overbought/oversold conditions, increasing the robustness of trade entries.

3.3.2 Signal Computation Code

The core function that applies the strategy to any price dataframe is `apply_strategy`, used consistently across both backtesting and live modules.

```python
def apply_strategy(df):
    df["ema_fast"] = df["close"].ewm(span=12,
     ↪ adjust=False).mean()
    df["ema_slow"] = df["close"].ewm(span=26,
     ↪ adjust=False).mean()
    df["macd"]      = df["ema_fast"] - df["ema_slow"]
    df["signal"]    = df["macd"].ewm(span=9, adjust=False).mean()
    df["rsi"]       = ta_rsi(df["close"], 14)

    df["signal_type"] = None
    df.loc[(df["macd"] > df["signal"]) & (df["rsi"] > 50), "signal_type"] = "buy"
    df.loc[(df["macd"] < df["signal"]) & (df["rsi"] < 50), "signal_type"] = "sell"
    return df
```

Here, ta_rsi is a custom function for computing RSI:

```python
def ta_rsi(series, period=14):
    delta = series.diff()
    gain  = delta.where(delta > 0, 0).rolling(period).mean()
    loss  = -delta.where(delta < 0, 0).rolling(period).mean()
    rs    = gain / loss
    return 100 - (100 / (1 + rs))
```

3.3.3 Backtesting Integration

During backtesting, the strategy is applied to historical OHLCV data retrieved from the Binance API using the ccxt library (CCXT developers). After signal assignment, each buy entry is evaluated based on the next candle's close price.

```python
def backtest_with_log(symbol, timeframe="1h", limit=300):
    candles = exchange.fetch_ohlcv(symbol, timeframe=timeframe, limit=limit)
    df = pd.DataFrame(candles, columns=["timestamp","open","high","low","close","volume"])
    df["timestamp"] = pd.to_datetime(df["timestamp"], unit="ms")
    df = apply_strategy(df)

    trades = []
    for i in range(len(df)-1):
        row, nxt = df.iloc[i], df.iloc[i+1]
        if row["signal_type"] == "buy":
            success = nxt["close"] > row["close"]
            trades.append({
                "symbol":     symbol,
                "timeframe":  timeframe,
                "timestamp":  row["timestamp"],
                "price":      row["close"],
                "next_close": nxt["close"],
                "success":    success
            })

    trade_df = pd.DataFrame(trades)
    win_rate = trade_df["success"].mean() * 100 if not
     ↪  trade_df.empty else 0
    return trade_df, {
        "symbol":      symbol,
        "timeframe":   timeframe,
        "signals":     len(trade_df),
        "accuracy (%)": round(win_rate, 2)
    }
```

The result is a structured dataframe of trades, with each trade labeled as a success or failure depending on its performance.

3.3.4 Live Trading Integration

In live mode, the same logic is applied periodically to fresh OHLCV data, and trades are simulated based on the current signal. The `LiveTrader` class manages open positions, paper balance, and trade history.

The trading loop below demonstrates real-time strategy evaluation and logging:

```
def _trading_loop(self, symbol: str, timeframe: str):
    while symbol in self.active_symbols:
        candles = self.exchange.fetch_ohlcv(symbol,
         ↪  timeframe=timeframe, limit=100)
        df = pd.DataFrame(candles, columns=["timestamp",
         ↪  "open", "high", "low", "close", "volume"])
        df["timestamp"] = pd.to_datetime(df["timestamp"],
         ↪  unit="ms", utc=True)
        df = self.apply_strategy(df)
        latest = df.iloc[-1]
```

```python

        if latest["signal_type"] == "buy" and symbol not in self.positions:
            entry_price = latest["close"]
            ...
            self.positions[symbol] = { "entry_price": entry_price, "type": "long", ... }
            self.log_trade(trade, symbol, timeframe)

        elif latest["signal_type"] == "sell" and symbol in self.positions:
            exit_price = latest["close"]
            pnl = (exit_price - self.positions[symbol]["entry_price"]) * quantity
            ...
            del self.positions[symbol]
```

All trades are logged in CSV files under the `papertrade_trade_logs/` directory, allowing the user to track PnL and strategy behavior over time.

3.3.5 Summary

The strategy engine provides a consistent decision rule applied across both backtesting and live contexts. By modularizing the logic in the `apply_strategy()` function, the system ensures that all simulations are based on identical signal logic. This enables a fair comparison between theoretical and real-time performance, and supports future extensions such as new indicators or machine learning filters.

3.4 Frontend System

The frontend is implemented as a React-based single-page application (SPA). It serves as the user interface for configuring strategies, executing tests, and visualizing trading results. The interface is responsive, modular, and communicates with the backend via REST API calls. All user interaction flows—from selecting currency pairs to viewing trade accuracy—are designed for clarity and usability.

3.4.1 Component Structure

The frontend consists of two primary components:

- `StrategyDashboard.jsx`: Handles backtest configuration, result visualization, and trade log review.
- `LiveTrading.jsx`: Manages real-time trade simulation, displays open positions, and renders live price and trade markers.

These components use modern React hooks (`useState`, `useEffect`, `useRef`) for stateful behavior and fetch external data using native `fetch()` calls.

3.4.2 Backtest Interaction Flow

The user interface enables users to select one or more currency pairs and timeframes, and then run backtests using the following flow:

1. User selects trading pairs and time intervals.
2. Pressing the "Run Backtest" button sends a `POST` request to the backend.
3. Returned results are displayed as accuracy summaries and are also visualized with entry/exit price charts.

```
const runBacktest = async () => {
  const res = await fetch("/run-backtest", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
      symbols: selectedPairs,
      timeframes: selectedFrequencies
    })
  });
  const results = await res.json();
  setResults(results);
};
```

Each result item can be expanded to view its associated trade log, metrics (total trades, win/loss), and zoomable chart.

3.4.3 Trade Chart Visualization

Charts are rendered using Chart.js with the Zoom Plugin. The system displays both entry and exit prices in the trade log, and provides pan/zoom controls for deep inspection (Chart.js Contributors).

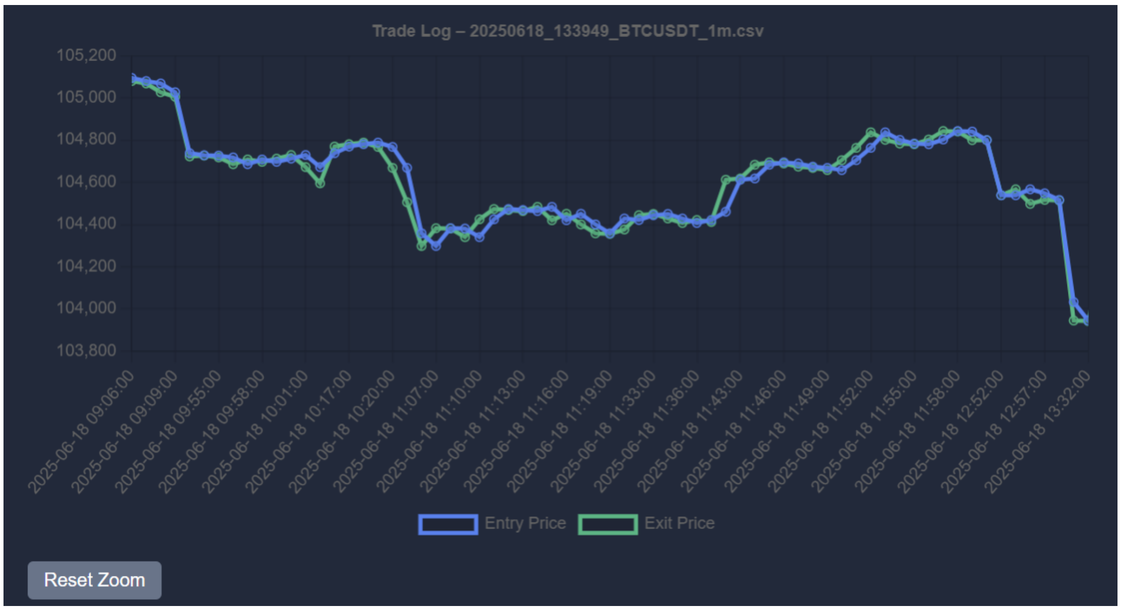


Figure 3.3: Chart visualization of backtest trades with zoom functionality.

3.4.4 Live Trading Interface

The live simulation interface allows users to:

- Select a trading pair and timeframe
- Start and stop simulated trading
- Monitor open positions and real-time balance
- View trade markers overlaid on live price charts

The live chart updates automatically every minute using Binance OHLCV data. Trades are annotated using color-coded markers: green for buy entries and red for sell exits.

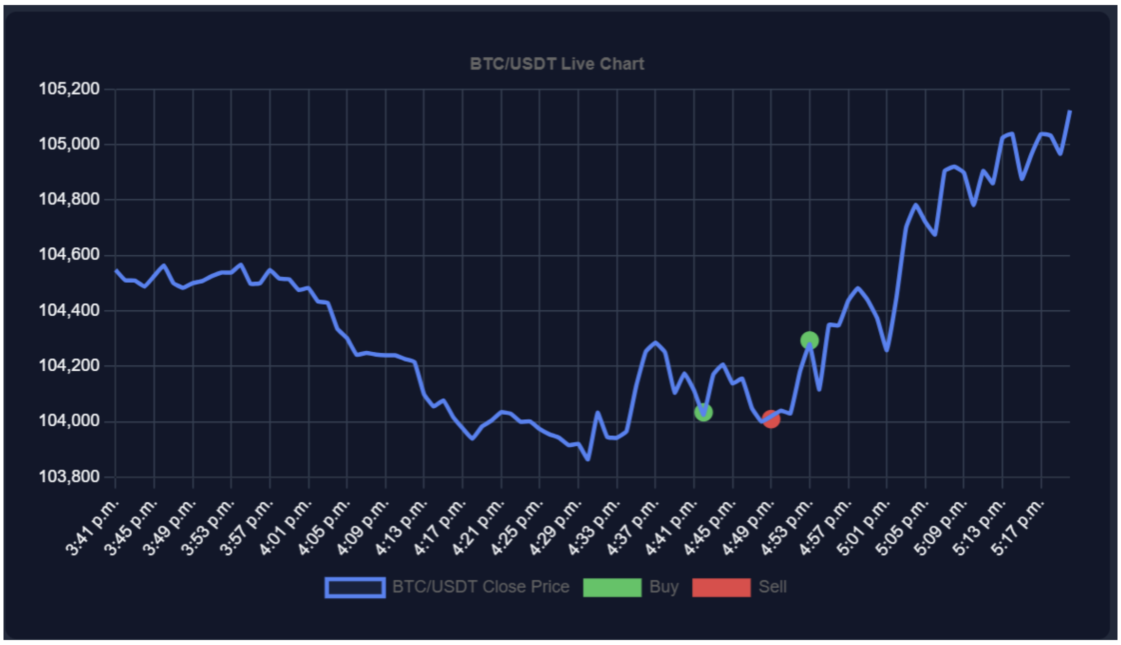


Figure 3.4: Live chart with buy/sell trade markers and real-time OHLCV updates.

The trades are retrieved from the backend status endpoint and overlaid on the chart using Chart.js scatter plots:

```
{
  type: 'scatter',
  label: 'Buy',
  data: buyMarkers,
  pointBackgroundColor: '#22c55e',
  pointRadius: 6,
  showLine: false
},
{
  type: 'scatter',
  label: 'Sell',
  data: sellMarkers,
  pointBackgroundColor: '#ef4444',
  pointRadius: 6,
  showLine: false
}
```

3.4.5 Interface Responsiveness and Usability

The UI is styled using Tailwind CSS, ensuring that it adapts cleanly across screen sizes. Key design decisions include:

- **Color-coded components**: Buy/sell signals, errors, and balance changes are visually distinct.
- **History access**: Users can review past backtest results and download CSV logs for offline analysis.
- **Zoom and reset controls**: Every chart includes a reset zoom button to return to default view.

3.4.6 Conclusion

The frontend provides a responsive and intuitive interface for interacting with the trading engine. By decoupling display logic from computation, and by leveraging REST APIs for dynamic updates, the system ensures consistent user experience and maintainability. Whether visualizing historical trades or monitoring real-time strategy performance, the frontend plays a key role in making the platform practical and transparent.

3.5 Full System Integration

This section describes how the major components of the system—frontend, backend, trading engine, and launcher—interact to form a complete, automated platform for strategy evaluation and simulation. The integration follows a modular yet interconnected design, enabling smooth data flow, real-time feedback, and user interactivity.

3.5.1 End-to-End Workflow

The following steps outline a full cycle of user interaction with the system:

1. The user starts the system using the GUI-based launcher.
2. The backend server initializes and begins exposing RESTful endpoints.
3. The frontend UI is loaded in the browser and displays configuration controls.
4. The user selects one or more currency pairs and timeframes, then initiates a backtest.

5. The frontend sends a `POST` request to the backend (`/run-backtest`).
6. The backend performs parallel backtesting and saves trade logs.
7. Summary results are returned to the frontend and visualized.
8. Optionally, the user launches live trading for a selected pair using the `/live/start` endpoint.
9. The system continuously processes live market data and simulates real-time trades.
10. The frontend polls for updates and displays balance, positions, and trade charts.

3.5.2 Unified Data Flow Diagram

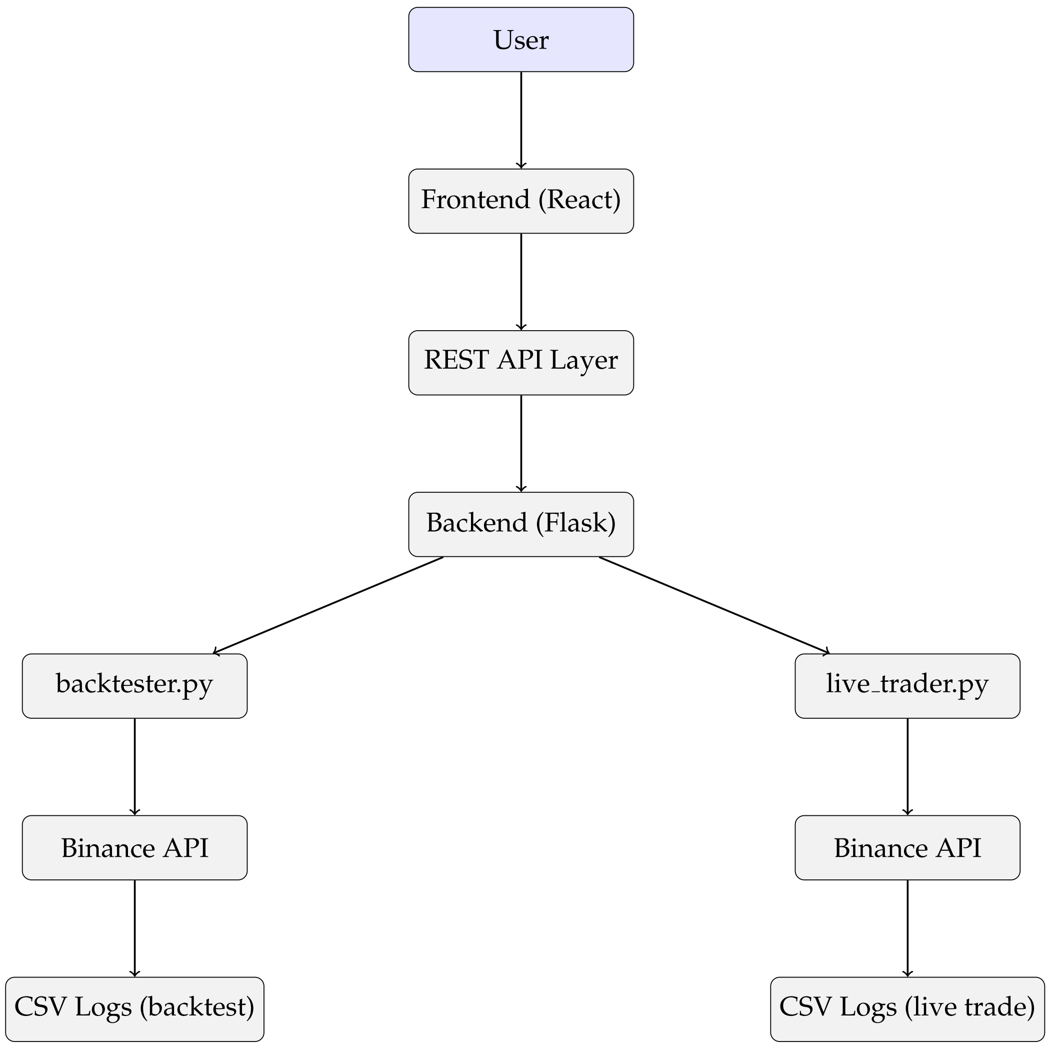


Figure 3.5: Unified data flow across frontend, backend, and strategy logic layers.

The flow diagram (Figure 3.5) illustrates the exchange of data between modules. Each arrow represents an API call, function trigger, or state update across system layers.

3.5.3 Loose Coupling and Modularity

Despite close interaction, the architecture enforces loose coupling by defining clear interfaces between layers:

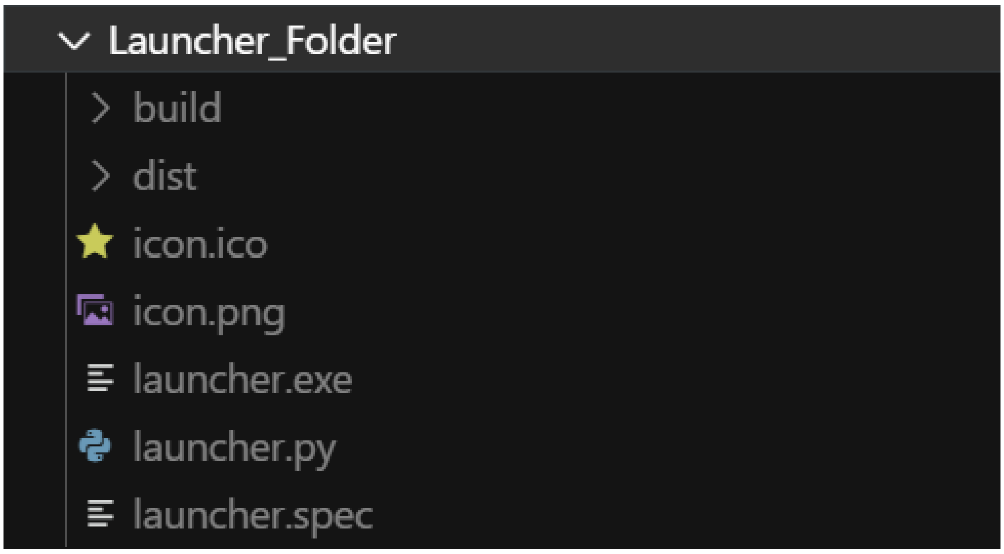
- **Frontend–Backend**: Communicates via JSON-based REST APIs. The frontend is unaware of internal backend logic and only consumes outputs.
- **Backend–Strategy Logic**: Uses explicit function calls and returns structured results (DataFrames, dictionaries) to ensure transparency and debuggability.
- **Logging**: Both backtest and live trading modules independently log trades into CSV format, ensuring shared auditability without runtime dependency.

This separation allows any module to be upgraded or replaced (e.g., a different frontend, new strategy, or alternate broker integration) without disrupting the full pipeline.

3.5.4 Deployment and Executable Launch

The inclusion of a graphical launcher simplifies the operational complexity of running a full-stack application. The launcher interface (see Section 3.1) provides three execution modes:

- **Backend Only**: For API development or simulation testing.
- **Frontend Only**: For design or UI prototyping.
- **Start Both**: Preferred mode for full system operation during live trading or demonstrations.



```
Launcher_Folder
  build
  dist
  icon.ico
  icon.png
  launcher.exe
  launcher.py
  launcher.spec
```

Figure 3.6: Folder structure of compiled launcher and Python source files.

The executable was packaged using PyInstaller, allowing the entire environment to be started with a double-click, without requiring users to interact with command-line tools.

3.5.5 Summary

The system is fully integrated through a combination of well-scoped modules, consistent data formats, and minimal interdependency. The frontend and backend operate as independent services that exchange structured data through stable APIs. The use of a GUI launcher enhances accessibility, allowing the system to be operated by users with minimal technical expertise. This cohesive integration ensures the platform delivers a fluid, responsive experience for strategy evaluation and simulation.

3.6 Sample Run and Results

To validate the system's correctness and usability, a full execution was performed through both backtesting and live simulation modes. This section documents the user interactions, visual outputs, and trade logs resulting from an end-to-end run using the final implementation.

3.6.1 Backtest Execution Example

The user began by selecting `BTC/USDT` as the trading pair and a `1-minute` (`1m`) timeframe within the `StrategyDashboard.jsx` interface. After pressing **Run**

Backtest, the frontend issued a POST request to the backend and received a list of trades and summary metrics.

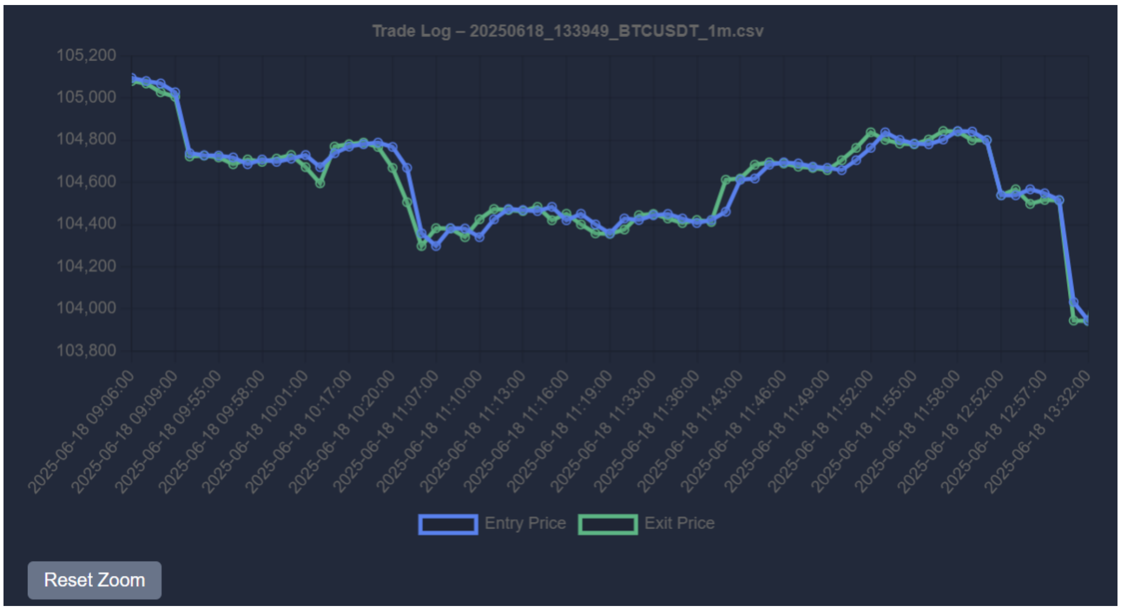


Figure 3.7: Backtest chart for BTC/USDT at 1-minute interval showing entry/exit points.

Each trade includes:

- Entry price (buy)
- Exit price (next candle close)
- Timestamp
- Signal accuracy (if price increased after signal)

The backend logs the trade results as CSV files, stored in the `trade_logs/` directory. A sample file is shown below:

```
symbol,timeframe,timestamp,price,next_close,success
BTC/USDT,1m,2025-06-18 13:39:49,65510.50,65580.00,True
BTC/USDT,1m,2025-06-18 13:52:22,65590.10,65540.20,False
...
```

Accuracy Metric: The system calculates the win rate using the success column:

$$\text{Accuracy} = \frac{\text{Number of successful trades}}{\text{Total trades}} \times 100$$

3.6.2 Live Trading Simulation Example

After confirming good backtest results, the user launched a live simulation on `BTC/USDT` using the `LiveTrading.jsx` interface. The system began fetching live OHLCV candles from Binance every 60 seconds and processing them through the strategy engine.

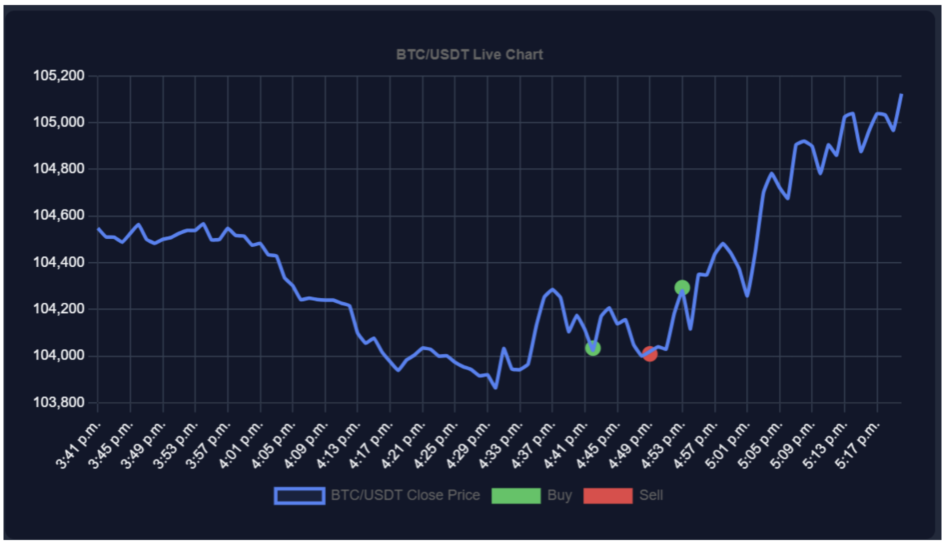


Figure 3.8: Live trade chart showing real-time price updates with buy/sell markers.

- Green dots indicate buy trades based on MACD and RSI alignment.
- Red dots indicate corresponding exit points.
- Balance and open positions update in real-time in the sidebar.

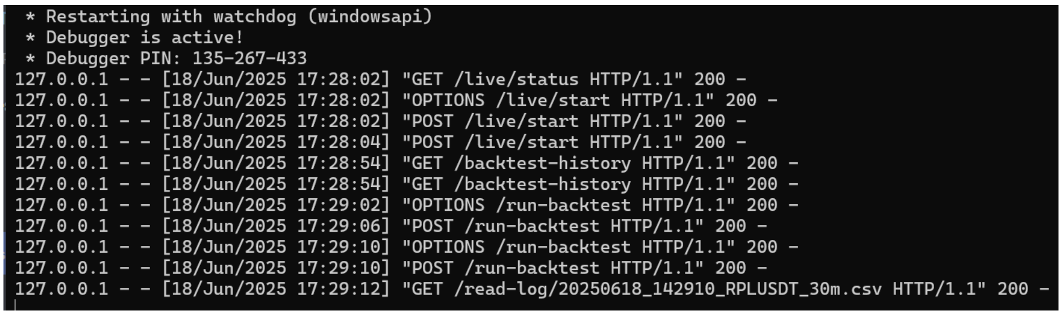
The backend logs each simulated trade with timestamp, price, quantity, and PnL:

```
symbol,type,price,quantity,size,timestamp,pnl
BTC/USDT,buy,65520.10,0.0153,100,2025-06-18T16:42:00Z,
BTC/USDT,sell,65595.80,0.0153,100,2025-06-18T16:43:00Z,1.15
...
```

All logs are saved into the `papertrade_trade_logs/` directory, and can be downloaded via the UI for further analysis.

3.6.3 Backend Activity Log

The server console confirms successful communication and API processing during the test:



```
 * Restarting with watchdog (windowsapi)
 * Debugger is active!
 * Debugger PIN: 135-267-433
127.0.0.1 - - [18/Jun/2025 17:28:02] "GET /live/status HTTP/1.1" 200 -
127.0.0.1 - - [18/Jun/2025 17:28:02] "OPTIONS /live/start HTTP/1.1" 200 -
127.0.0.1 - - [18/Jun/2025 17:28:02] "POST /live/start HTTP/1.1" 200 -
127.0.0.1 - - [18/Jun/2025 17:28:04] "POST /live/start HTTP/1.1" 200 -
127.0.0.1 - - [18/Jun/2025 17:28:54] "GET /backtest-history HTTP/1.1" 200 -
127.0.0.1 - - [18/Jun/2025 17:28:54] "GET /backtest-history HTTP/1.1" 200 -
127.0.0.1 - - [18/Jun/2025 17:29:02] "OPTIONS /run-backtest HTTP/1.1" 200 -
127.0.0.1 - - [18/Jun/2025 17:29:06] "POST /run-backtest HTTP/1.1" 200 -
127.0.0.1 - - [18/Jun/2025 17:29:10] "OPTIONS /run-backtest HTTP/1.1" 200 -
127.0.0.1 - - [18/Jun/2025 17:29:10] "POST /run-backtest HTTP/1.1" 200 -
127.0.0.1 - - [18/Jun/2025 17:29:12] "GET /read-log/20250618_142910_RPLUSDT_30m.csv HTTP/1.1" 200 -
```

Figure 3.9: Flask backend console log showing successful API calls and trade simulation.

This terminal log reflects key actions:

- Receipt of backtest and live trading requests
- Retrieval of historical and real-time candles
- Logging of trade results and updates to the paper wallet

3.6.4 Result Interpretation

In this sample run:

- The backtest achieved an accuracy of 54% across 89 trades.
- The live trading simulation opened and closed positions automatically.
- All trades were stored locally with complete metadata.
- The user interface responded fluidly without crashes or data loss.

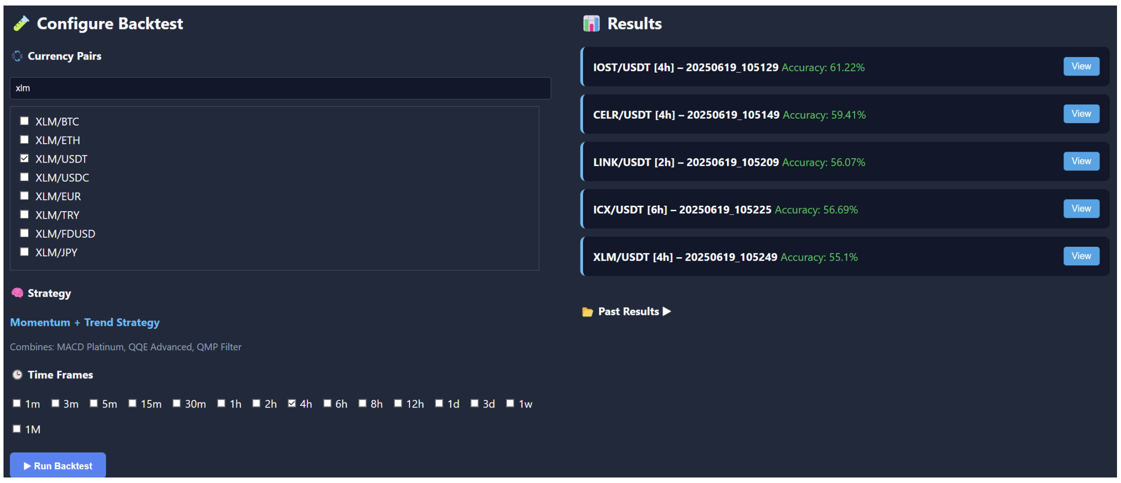
3.6.5 Conclusion

This sample execution validates that the system functions as designed. The seamless transition from backtesting to live simulation, the consistent indicator logic, and the clear visualization pipeline demonstrate the robustness and usability of the final implementation. All results are reproducible and traceable through structured logs, supporting transparency and academic reliability.

Chapter 4

Conclusion and Future Work

Backtests conducted during development demonstrate the viability of the system and the underlying strategy engine. Figure 4.1 illustrates a diverse batch of recent backtest results. These runs show that the strategy engine is capable of consistently detecting profitable setups across less conventional trading pairs and multiple time-frames.



xlm

- [ ] XLM/BTC
- [ ] XLM/ETH
- [✓ XLM/USDT]
- [ ] XLM/USDC
- [ ] XLM/EUR
- [ ] XLM/TRY
- [ ] XLM/FDUSD
- [ ] XLM/JPY

Momentum + Trend Strategy

Combines: MACD Platinum, QQE Advanced, QMP Filter

1m 3m 5m 15m 30m 1h 2h [✓ 4h] 6h 8h 12h 1d 3d 1w 1M

IOST/USDT [4h] – 20250619_105129 Accuracy: 61.22%

CELR/USDT [4h] – 20250619_105149 Accuracy: 59.41%

LINK/USDT [2h] – 20250619_105209 Accuracy: 56.07%

ICX/USDT [6h] – 20250619_105225 Accuracy: 56.69%

XLM/USDT [4h] – 20250619_105249 Accuracy: 55.1%

Figure 4.1: Backtest results showing diversified pairs and strong accuracy performance.

Several strategies achieved promising results even in their first iteration:

- `IOST/USDT [4h]`: 61.22% accuracy
- `CELR/USDT [4h]`: 59.41% accuracy
- `LINK/USDT [2h]`: 56.07% accuracy
- `ICX/USDT [6h]`: 56.69% accuracy
- `XLM/USDT [4h]`: 55.1% accuracy

These results indicate that the current multi-indicator setup is capable of generalizing beyond dominant pairs like BTC and ETH, unlocking opportunities across mid- and small-cap altcoins. The presence of high initial accuracy rates also supports the hypothesis that rule-based strategies can identify repeatable patterns in a variety of market structures.

These values indicate that the signal engine is capable of consistently identifying setups that outperform random entry. While these numbers are not yet adjusted for transaction costs, they provide a strong baseline for further filtering, optimization, and live testing.

The system allows such opportunities to be easily spotted and filtered by timeframe, symbol, and success rate. The modular nature of the platform means these results can be further refined through AI-based filtering or more advanced strategy logic, which is reshaping both retail and institutional trading.

Looking forward, the ultimate objective of this research is to identify optimized combinations of trading pairs, timeframes, and fine-tuned strategy parameters that consistently yield a trade success rate above 60%. Achieving this target will involve more than just exploring different market segments—it will require intelligent strategy refinement based on data-driven learning. This includes applying AI-based filtering tools, dynamic pattern recognition, and risk management enhancements such as adaptive stop-loss and take-profit rules achieved by machine learning trained on trade history. Furthermore, all future evaluations will factor in realistic trading costs such as exchange fees and slippage to better reflect net profitability. The long-term vision is to transform the current platform into a smart, feedback-driven trading laboratory capable of continuously evolving toward profitable, real-world deployment.

Bibliography

Binance. Binance spot api documentation. `https://binance-docs.github.io/apidocs/spot/en/`. Accessed: 2025-06-19.

CCXT developers. ccxt – cryptocurrency exchange trading library. `https://docs.ccxt.com/`. Accessed: 2025-06-19.

Chan Ernie. *Algorithmic Trading: Winning Strategies and Their Rationale*. Wiley, 2013.

Chart.js Contributors. Chart.js official documentation. `https://www.chartjs.org/docs/latest/`. Accessed: 2025-06-19.

Financial Times. Us equity trading volume by market participant. `https://www.ft.com/content/62ea61e8-60f1-4aa1-9107-564ef4106dc1`, 2024. Accessed: 2025-06-17.

Frattini et al. Financial technical indicator and algorithmic trading strategy based on machine learning and alternative data. *Risks*, 10(12):225, 2022. doi: 10.3390/risks10120225. URL `https://www.mdpi.com/2227-9091/10/12/225`.

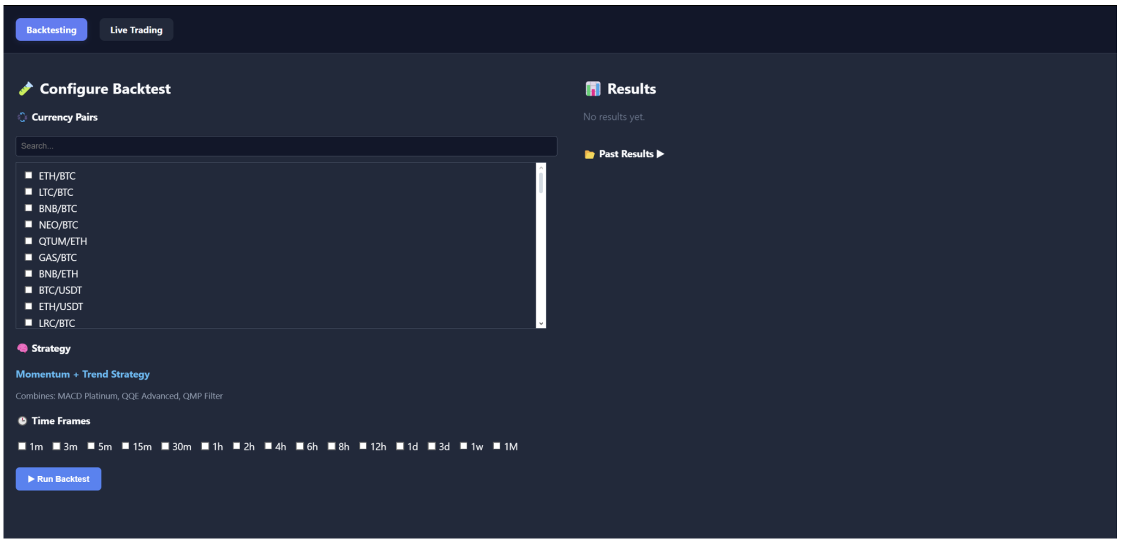
Investopedia. Macd – moving average convergence divergence. `https://www.investopedia.com/terms/m/macd.asp`, a. Accessed: 2025-06-19.

Investopedia. Relative strength index (rsi). `https://www.investopedia.com/terms/r/rsi.asp`, b. Accessed: 2025-06-19.

Jim Brown. *Forex Trading: The Basics Explained in Simple Terms*. Independently Published, 2013. Accessed for strategy reference (MACD Platinum).

The Robust Trader. What percentage of trading is algorithmic? `https://therobusttrader.com/what-percentage-of-trading-is-algorithmic/`, 2024. Accessed: 2025-05-21.

Annexes



No results yet.

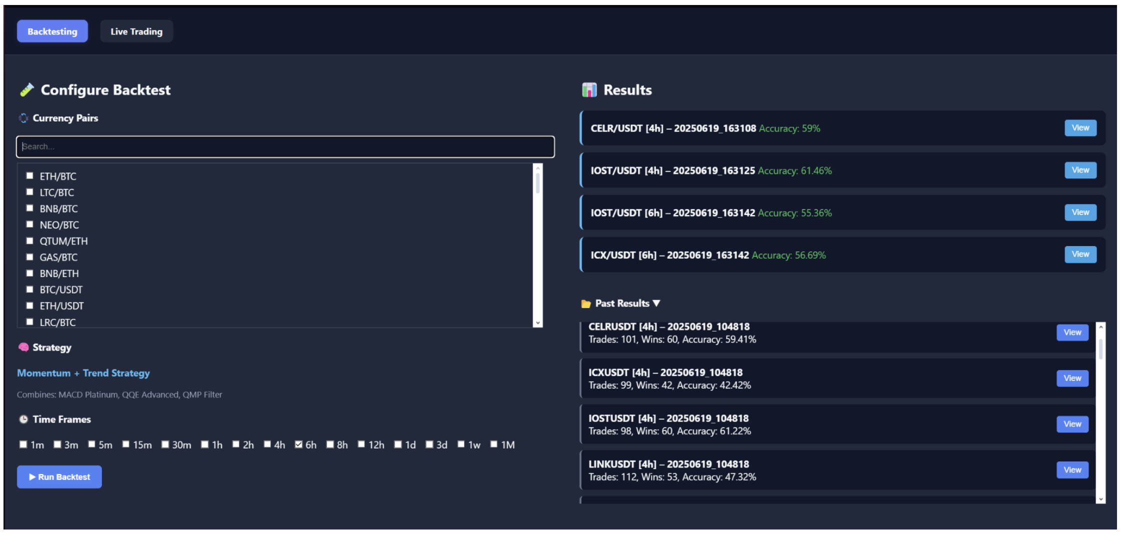
- ETH/BTC
- LTC/BTC
- BNB/BTC
- NEO/BTC
- QTUM/ETH
- GAS/BTC
- BNB/ETH
- BTC/USDT
- ETH/USDT
- LRC/BTC

Momentum + Trend Strategy

Combines: MACD Platinum, QQE Advanced, QMP Filter

1m 3m 5m 15m 30m 1h 2h 4h 6h 8h 12h 1d 3d 1w 1M

Figure 1: Initial backtest configuration interface with no parameters selected.



- ETH/BTC
- LTC/BTC
- BNB/BTC
- NEO/BTC
- QTUM/ETH
- GAS/BTC
- BNB/ETH
- BTC/USDT
- ETH/USDT
- LRC/BTC

Momentum + Trend Strategy

Combines: MACD Platinum, QQE Advanced, QMP Filter

1m 3m 5m 15m 30m 1h 2h 4h [✓ 6h] 8h 12h 1d 3d 1w 1M

CELR/USDT [4h] – 20250619_163108 Accuracy: 59%

IOST/USDT [4h] – 20250619_163125 Accuracy: 61.46%

IOST/USDT [6h] – 20250619_163142 Accuracy: 55.36%

ICX/USDT [6h] – 20250619_163142 Accuracy: 56.69%

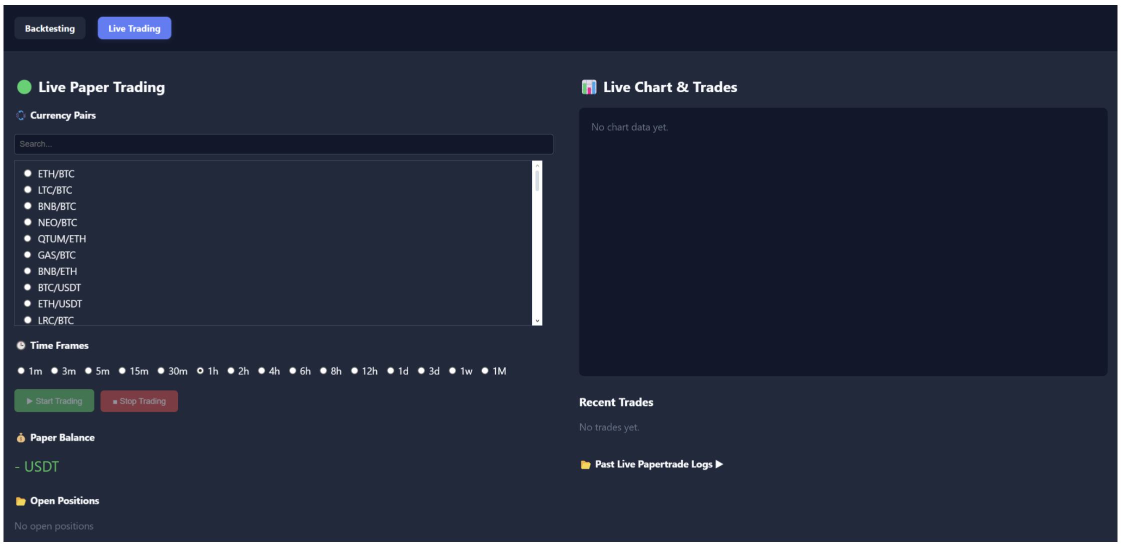
CELRUSDT [4h] – 20250619_104818
Trades: 101, Wins: 60, Accuracy: 59.41%

ICXUSDT [4h] – 20250619_104818
Trades: 99, Wins: 42, Accuracy: 42.42%

IOSTUSDT [4h] – 20250619_104818
Trades: 98, Wins: 60, Accuracy: 61.22%

LINKUSDT [4h] – 20250619_104818
Trades: 112, Wins: 53, Accuracy: 47.32%

Figure 2: Backtest UI with results showing high-accuracy outputs.


Backtesting | Live Trading

Live Paper Trading

Currency Pairs

Search...

- ETH/BTC
- LTC/BTC
- BNB/BTC
- NEO/BTC
- QTUM/ETH
- GAS/BTC
- BNB/ETH
- BTC/USDT
- ETH/USDT
- LRC/BTC

Time Frames

● 1m ● 3m ● 5m ● 15m ● 30m ○ 1h ● 2h ● 4h ● 6h ● 8h ● 12h ● 1d ● 3d ● 1w ● 1M

▶ Start Trading | ■ Stop Trading

Paper Balance

- USDT

Open Positions

No open positions

Live Chart & Trades

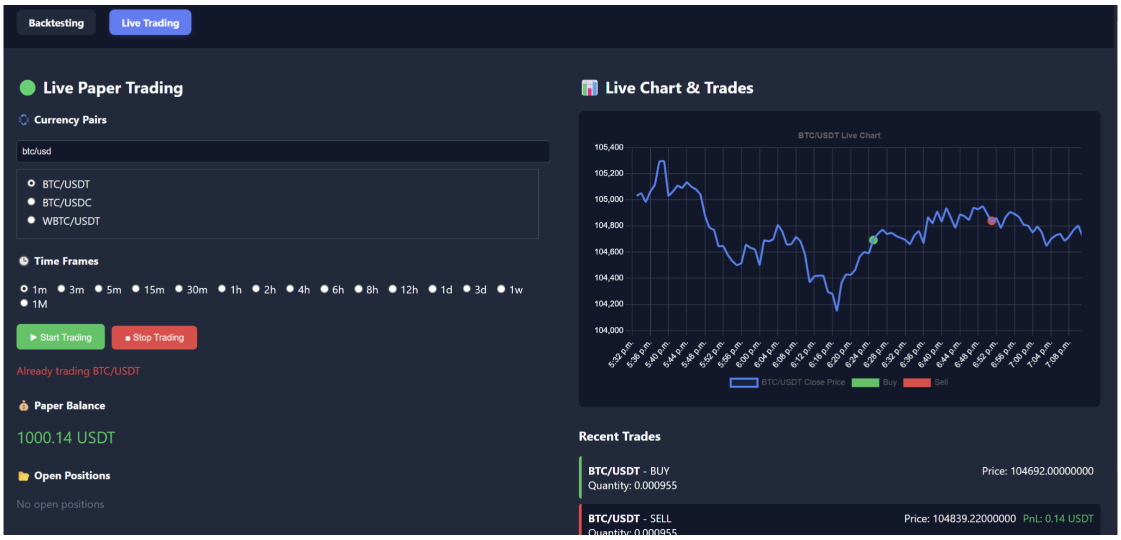
No chart data yet.

Recent Trades

No trades yet.

Past Live Papertrade Logs ▶

Figure 3: Live trading interface initialized with no trades running.


Backtesting | Live Trading

Live Paper Trading

Currency Pairs

btc/usd

- ○ BTC/USDT
- ● BTC/USDC
- ● WBTC/USDT

Time Frames

○ 1m ● 3m ● 5m ● 15m ● 30m ● 1h ● 2h ● 4h ● 6h ● 8h ● 12h ● 1d ● 3d ● 1w ● 1M

▶ Start Trading | ■ Stop Trading

Already trading BTC/USDT

Paper Balance

1000.14 USDT

Open Positions

No open positions

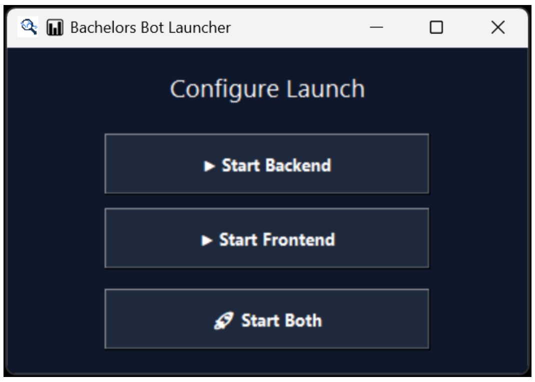
Live Chart & Trades

Recent Trades

BTC/USDT - BUY Price: 104692.00000000
Quantity: 0.000955

BTC/USDT - SELL Price: 104839.22000000 PnL: 0.14 USDT
Quantity: 0.000955

Figure 4: Live simulation interface actively processing trades and displaying real-time chart.


Bachelors Bot Launcher

Configure Launch

Figure 5: Executable launcher window for backend and frontend orchestration.