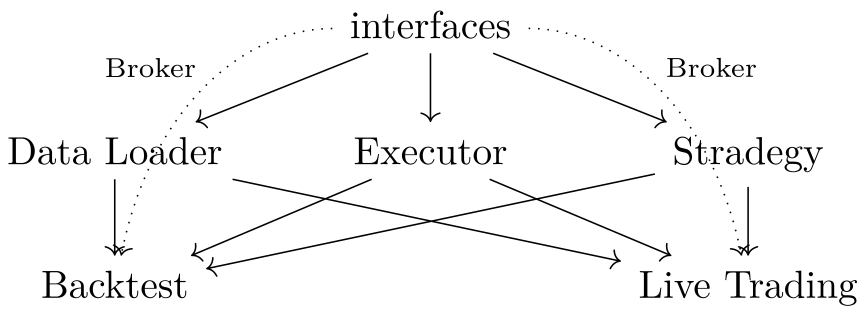
Architecture

The trading system is organized around a clear modular architecture that separates concerns into distinct components. At the foundation is **interfaces**, which defines abstract base classes for brokers, data feeds, and trading strategies. Concrete implementations plug into these interfaces: the **Data Loader** module fetches price candles (OHLCV) from Binance, either for historical backtesting or real-time execution. The **Strategy** module contains the trading logic (e.g., EMA crossovers), which receives market data and outputs BUY or SELL signals. Orders are executed through the **Executor**, which is broker-agnostic and simply calls a broker object and logs trades into CSV files. On top of this core engine sit two user-facing entry points: **Backtest**, which simulates trading on historical data using a lightweight broker adapter, and **Live Trading**, which streams real-time candles and executes trades on the Binance Testnet through a real broker implementation. This modular design enables swapping brokers or strategies without changing other code, making the system extendable, testable, and production-ready.



Strategy

The strategy implemented in this system is a **multi-timeframe EMA trend-following crossover strategy**. It operates on 15-minute candles for entries and exits while using 1-hour candles as a trend filter. First, the strategy computes short-term exponential moving averages (EMA10 and EMA20) on the 15-minute price data and long-term EMAs (EMA50 and EMA200) on the 1-hour timeframe. A long position is only considered when the long-term trend is bullish — defined by EMA50 on the 1-hour chart being higher than EMA200. Within that trend regime, an entry signal occurs when the short-term momentum flips upward, meaning EMA10 crosses above EMA20. After entering a position, the system continuously monitors for exit conditions: it exits the trade when the short-term momentum weakens, signaled by EMA10 crossing back below EMA20. Because the logic is based on mathematical comparisons on each candle, it returns signals as boolean conditions, allowing the Executor to act deterministically and removing emotional bias from the decision-making process.

Parity

Parity between backtesting and live trading is achieved by designing both systems to use the **same strategy logic, indicator calculations, and trade execution flow**, while only swapping the broker and data sources. The backtest runs historical candles through the exact same MultiTimeframeEMAStrategy class that live trading uses, ensuring signals are generated identically. In execution, the TradeExecutor provides a unified interface so both modes log trades the same way—into CSV—while the broker implementation is injected via an interface (IExchangeBroker). In a backtest, a lightweight BacktestBroker simulates price and fills orders instantly, whereas in live trading, a BinanceTestnetBroker places actual API test orders, but both expose identical methods such as `get_price()` and `place_order()`. This abstraction layer ensures that only the environment changes—historical dataset vs. real-time API—while the system behavior and decision logic remain consistent, enabling reliable, reproducible results when moving from simulation to real-world execution.