General Concept:

Macro: 1day

* Predicting up or down + magnitude (if possible) on a 24hour period.
  + Ref($close, -2)/Ref($close, -1) – 1
* LGMBoost Quantile Model
  + Q10
  + Q50
  + Q90
* GDelt Features for Regime Detection?
* Bitcoin Dominance: Market Cap
* Crypto Fear & Greed Index

TODO: evaluate if two layers is necessary? Maybe pipe to existing trading bot strategy…

Micro: 60min

* Interpreting directional / macro signals
  + TODO: 4 weeks of order book data
* Position Strategy: Long / Short
  + TODO: Try other env to see if we can get backtest info of some sort
* Trading Framework Integration:
  + Nautilus
  + Hummingbird
  + Fenix

Appendix A: Nautilus Integration

If you want to integrate a custom data source into NautilusTrader, you'll need to create a custom data adapter. NautilusTrader's architecture leverages adapters to interface with different data providers and trading venues

.

Here's a high-level overview of the process:

1. **Understand NautilusTrader's data model:** Familiarize yourself with NautilusTrader's core data objects like Instruments, Bars, and Ticks. Your adapter will need to translate your custom data into this format.
2. **Determine the type of data client:** NautilusTrader provides different client types like LiveDataClient for real-time data and HistoricalDataClient for backtesting. Choose the appropriate type based on your custom data source.
3. **Implement the data handling logic:** Within your adapter, you'll need to write code that connects to your data source, retrieves the data, and transforms it into NautilusTrader's data objects.
4. **Define handlers for data updates:** NautilusTrader strategies use data handlers to react to incoming data. Your adapter will be responsible for pushing the converted data to these handlers.
5. **Consider performance optimizations:** If you're dealing with large datasets, explore ways to optimize data loading and processing. NautilusTrader's use of Rust components and Parquet format for data storage can offer significant performance benefits.

In short

* **Data Client:** You'll likely need to implement a custom DataClient or extend an existing one (like LiveDataClient or HistoricalDataClient) to handle your specific data source.
* **Data Conversion:** Your adapter will need to convert the data from your source's format into NautilusTrader's data objects (e.g., Instruments, Bars, Ticks).
* **Data Handling:** Strategies in NautilusTrader use data handlers (methods like on\_bar, on\_trade) to receive and process data updates. Your adapter will need to trigger these handlers with your custom data.
* **Performance:** If you're dealing with large datasets, consider optimizing your data loading and processing for efficient backtesting and live trading. NautilusTrader leverages Rust and Parquet for performance, and you can take advantage of these features.

**Example:** NautilusTrader provides an integration with Databento, a market data provider, using a custom adapter. This adapter leverages the Databento API to retrieve historical and real-time market data, and then converts that data into Nautilus objects for use in trading strategies.

**Important:** You may also need to consider aspects like error handling, authentication with your data source, and potentially creating custom instrument providers for your specific needs, [according to NautilusTrader documentation](https://nautilustrader.io/docs/nightly/concepts/adapters/).

Appendix B: Hummingbird API

Concept: Make calls to update Hummingbird Strategies based on execution of predictive models.

**Hummingbot API**

A comprehensive RESTful API framework for managing trading operations across multiple exchanges. The Hummingbot API provides a centralized platform to aggregate all your trading functionalities, from basic account management to sophisticated automated trading strategies.

**What is Hummingbot API?**

The Hummingbot API is designed to be your central hub for trading operations, offering:

* **Multi-Exchange Account Management**: Create and manage multiple trading accounts across different exchanges
* **Portfolio Monitoring**: Real-time balance tracking and portfolio distribution analysis
* **Trade Execution**: Execute trades, manage orders, and monitor positions across all your accounts
* **Automated Trading**: Deploy and control Hummingbot instances with automated strategies
* **Strategy Management**: Add, configure, and manage trading strategies in real-time
* **Complete Flexibility**: Build any trading product on top of this robust API framework

Whether you're building a trading dashboard, implementing algorithmic strategies, or creating a comprehensive trading platform, the Hummingbot API provides all the tools you need.

**System Dependencies**

The Hummingbot API requires two essential services to function properly:

**1. PostgreSQL Database**

Stores all trading data including:

* Orders and trade history
* Account states and balances
* Positions and funding payments
* Performance metrics

**2. EMQX Message Broker**

Enables real-time communication with trading bots:

* Receives live updates from running bots
* Sends commands to control bot execution
* Handles real-time data streaming

<https://hummingbot.org/hummingbot-api/quickstart/#complete-example>

**Next Steps**[**¶**](https://hummingbot.org/hummingbot-api/quickstart/#next-steps)

Now that you've completed the quickstart, explore more advanced features:

* [**Bot Management**](https://hummingbot.org/hummingbot-api/#bot-orchestration): Deploy and manage multiple trading bots
* [**Strategy Configuration**](https://hummingbot.org/hummingbot-api/#strategy-management): Configure and deploy trading strategies
* [**Market Data**](https://hummingbot.org/hummingbot-api/#market-data): Access real-time and historical market data
* [**Backtesting**](https://hummingbot.org/hummingbot-api/#backtesting): Test your strategies with historical data

For the complete API reference, visit the [API documentation](http://localhost:8000/docs) when your API is running.

**📈 Strategy Management**[**¶**](https://hummingbot.org/hummingbot-api/#strategy-management)

Configure and deploy trading strategies with real-time updates

* GET /controllers - List available strategy controllers
* POST /controllers/{name}/deploy - Deploy strategy controller
* PUT /controllers/{id}/config - Update strategy parameters
* GET /scripts - List available trading scripts
* POST /scripts/run - Execute trading script

**📉 Market Data**[**¶**](https://hummingbot.org/hummingbot-api/#market-data)

Access real-time and historical market data

* GET /market-data/ticker/{pair} - Get current ticker data
* GET /market-data/orderbook/{pair} - Get order book snapshot
* GET /market-data/candles/{pair} - Get historical candles
* GET /market-data/trades/{pair} - Get recent trades
* WS /market-data/stream - Real-time market data stream

**🤖 Bot Orchestration**[**¶**](https://hummingbot.org/hummingbot-api/#bot-orchestration)

Deploy, configure, and manage multiple bot instances

* GET /bot-orchestration/bots - List all bot instances
* POST /bot-orchestration/deploy - Deploy new bot
* PUT /bot-orchestration/bots/{id}/config - Update bot configuration
* POST /bot-orchestration/bots/{id}/start - Start bot
* POST /bot-orchestration/bots/{id}/stop - Stop bot
* GET /bot-orchestration/bots/{id}/status - Get bot status

**🧪 Backtesting[¶](https://hummingbot.org/hummingbot-api/" \l "backtesting" \o "Permanent link)**

Run strategy backtests with historical data

* POST /backtesting/run - Start new backtest
* GET /backtesting/results/{id} - Get backtest results
* GET /backtesting/metrics/{id} - Get performance metrics
* POST /backtesting/optimize - Run parameter optimization

<https://hummingbot.org/developers/strategies/>

<https://hummingbot.org/v2-strategies/executors/positionexecutor/#execution-flow>

**Execution Flow**[**¶**](https://hummingbot.org/v2-strategies/executors/positionexecutor/#execution-flow)

Here's a simplified flow of how the PositionExecutor operates in conjunction with the triple barrier method:

1. The PositionExecutor initiates a position based on signals from the strategy, which interprets market data
2. It continuously monitors market prices and compares them against the defined barriers.
3. If the price hits the take profit or stop loss levels, the PositionExecutor executes a trade to close the position accordingly.
4. The trailing stop is adjusted as the price moves favorably, providing a dynamic risk management tool.
5. The time limit barrier ensures that positions do not remain open indefinitely, addressing the risk of market conditions changing over time.

<https://github.com/hummingbot/hummingbot/blob/development/controllers/directional_trading/ai_livestream.py>

class AILivestreamController(DirectionalTradingControllerBase):

def \_\_init\_\_(self, config: AILivestreamControllerConfig, \*args, \*\*kwargs):

self.config = config

super().\_\_init\_\_(config, \*args, \*\*kwargs)

# Start ML signal listener

self.\_init\_ml\_signal\_listener()

def \_init\_ml\_signal\_listener(self):

"""Initialize a listener for ML signals from the MQTT broker"""

try:

normalized\_pair = self.config.trading\_pair.replace("-", "\_").lower()

topic = f"{self.config.topic}/{normalized\_pair}/ML\_SIGNALS"

self.\_ml\_signal\_listener = ExternalTopicFactory.create\_async(

topic=topic,

callback=self.\_handle\_ml\_signal,

use\_bot\_prefix=False,

)

self.logger().info("ML signal listener initialized successfully")

except Exception as e:

self.logger().error(f"Failed to initialize ML signal listener: {str(e)}")

self.\_ml\_signal\_listener = None

def \_handle\_ml\_signal(self, signal: dict, topic: str):

"""Handle incoming ML signal"""

# self.logger().info(f"Received ML signal: {signal}")

short, neutral, long = signal["probabilities"]

if short > self.config.short\_threshold:

self.processed\_data["signal"] = -1

elif long > self.config.long\_threshold:

self.processed\_data["signal"] = 1

else:

self.processed\_data["signal"] = 0

self.processed\_data["features"] = signal

short, neutral, long = signal["probabilities"]

AILivestreamControllerConfig(DirectionalTradingControllerConfigBase)

* Set LongThreshold
* Set ShortThreshold

Receive:

# self.logger().info(f"Received ML signal: {signal}")

short, neutral, long = signal["probabilities"]

if short > self.config.short\_threshold:

self.processed\_data["signal"] = -1

elif long > self.config.long\_threshold:

self.processed\_data["signal"] = 1

else:

self.processed\_data["signal"] = 0

* self.processed\_data["features"] = signal

Routes to PositionExecutor…

**Position Executor**

**PositionExecutor:** Manages opening and closing positions of equal amounts, ensuring the portfolio remains balanced ± the position's profit or loss. It's applicable in both perpetual and spot markets, requiring pre-ownership of the asset for spot markets.

The **PositionExecutor** uses a configuration object, **PositionExecutorConfig**, to manage an order after it is placed, following the [Triple Barrier Method](https://www.mlfinlab.com/en/latest/labeling/tb_meta_labeling.html). This configuration sets pre-defined stop loss, take profit, time limit, and trailing stop parameters.

class TripleBarrierConf(BaseModel):

# Configure the parameters for the position

stop\_loss: Optional[Decimal]

take\_profit: Optional[Decimal]

time\_limit: Optional[int]

trailing\_stop\_activation\_price\_delta: Optional[Decimal]

trailing\_stop\_trailing\_delta: Optional[Decimal]

# Configure the parameters for the order

open\_order\_type: OrderType = OrderType.LIMIT

take\_profit\_order\_type: OrderType = OrderType.MARKET

stop\_loss\_order\_type: OrderType = OrderType.MARKET

time\_limit\_order\_type: OrderType = OrderType.MARKET

Key Configs:

* stop\_loss: Determines the stop-loss percentage
* take\_profit: Sets the take-profit percentage.
* time\_limit: Establishes a time limit for the trade.
* trailing\_stop\_activation\_price\_delta: Specifies the delta for activating a trailing stop.
* trailing\_stop\_trailing\_delta: Sets the trailing delta for the trailing stop.

**📈 Strategy Management**[**¶**](https://hummingbot.org/hummingbot-api/#strategy-management)

Configure and deploy trading strategies with real-time updates

* GET /controllers - List available strategy controllers
* POST /controllers/{name}/deploy - Deploy strategy controller
* PUT /controllers/{id}/config - Update strategy parameters
* GET /scripts - List available trading scripts
* POST /scripts/run - Execute trading script

Can I access this controller through API?

TODO: refine underlying bitcoin quantile model

<https://github.com/sydney-machine-learning/quantiledeeplearning/blob/main/all_notebooks/bitcoin/uni/bitcoin_uni_clstm_quantile.ipynb>

<https://arxiv.org/pdf/2405.11431>

TODO: try running hummingbird

TODO: integrate RD-Agent to incrementally improve model