ChatGPT

Utilizing Polymarket's Subgraph for Strategy Development & Backtesting

Introduction

Polymarket is a decentralized prediction market platform whose **subgraph** indexes comprehensive on-chain data (trades, markets, users, etc.) since mid-2021 [1]. This rich dataset provides a foundation for developing and backtesting automated trading strategies. By leveraging the subgraph's historical market and trading data in combination with external APIs (like Polymarket's Gamma metadata API and CLOB order-book API), one can design a pipeline for **Semantic Logic Oracles (SLO)** and **Market Behavior Oracles (MBO)** to generate trading signals. In this report, we detail:

- Key categories of trading strategies applicable to Polymarket and their data requirements.
- For each strategy type, what data is available via the subgraph versus what must be supplemented (e.g. via Gamma API for metadata, off-chain order book snapshots, news feeds, etc.), and the feasibility of backtesting/automation with current infrastructure.
- Recommendations for engineering an SLO/MBO pipeline using the subgraph as the base layer (with points where additional data enrichment is needed).
- A mapping of subgraph schema fields (e.g. `Market`, `Trade`, `User`, `Position`, `DailyStats`) to specific components of a strategy pipeline: signal generation, backtesting, PnL attribution, and risk management.
- A modular system architecture that integrates data ingestion, oracle logic, signal processing, backtesting, and execution for live trading on Polymarket.

Throughout the analysis, we reference Polymarket's subgraph schema and data availability, and we assume the ability to query this subgraph for historical data. All on-chain data references are from the **Polymarket subgraph** unless otherwise noted. External data sources (Gamma API, CLOB API, news/NLP feeds) are discussed where relevant.

Strategy Types Applicable to Polymarket

Polymarket's unique microstructure (an on-chain AMM/CLOB hybrid for binary outcome tokens) enables several categories of trading strategies. Below we enumerate major strategy types and discuss for each: the strategy's premise, required data for development/backtesting, coverage of that data in the subgraph, additional data needs, and feasibility of automation with the available infrastructure.

1. Microstructure-Based Trading Strategies

Description: Microstructure strategies exploit short-term market mechanics and order flow patterns. Examples include *order book mispricing arbitrage, mean-reversion after large trades, spread capture, and high-frequency range trading*. These are **intra-day, high-frequency** strategies driven by fine-grained price movements (often seconds to minutes). A real-case example from the provided spec is a *"Revert-on-Spike"* strategy: if a single large buy sweeps the price of "Yes" from 0.55 to 0.70, the strategy shorts that spike, betting on mean reversion [2]. Another example is *range scalping*, repeatedly buying and selling within a narrow price range (e.g. 0.48–0.53) for small profits [3]. These approaches treat **order book dynamics, trade timing, and liquidity** as primary signals.

Data Requirements: Microstructure strategies require **high-resolution trade and order book data**: every tick or trade execution with timestamp, price, size, and side, plus **order book depth snapshots** to gauge available liquidity and spreads. The strategy logic often uses features like short-term price jumps, order imbalance, recent volume bursts, and open interest changes. For example, to detect a "spike" one might look at the price change in 10 seconds and z-score relative to recent volatility [4], identify if a **large trade** (e.g. >$X USD) caused it [5], and confirm low follow-up volume (indicating a transient dislocation) [6]. All these signals demand granular timestamped data.

- **Required data points:** per-second price updates or last trade price, trade volume in short windows (e.g. last 60s trades), identity of aggressive buyer/seller (to distinguish taker vs maker behavior), and ideally the top-of-book bid/ask and sizes (to estimate slippage and to place limit orders for range trading). Open interest (total shares outstanding) is also useful to see if a large trade introduced new liquidity or just transferred shares.

Subgraph Coverage: Polymarket's subgraph **fully logs every executed trade** since 2021 [7] [8]. Each `Trade` entry includes the taker `trader`, the maker `counterparty`, `side` (BUY/SELL), `amount` of outcome tokens, `price` per share (normalized 0–1), `timestamp`, and `fee` [9] [10]. This is rich tick-level data for reconstructing **trade-by-trade price time series** and order flow. For example, one can fetch all trades in a market sorted by time to analyze **price discovery over time, volume surges, and buy/sell pressure** [11]. The subgraph data enables detecting large trades (by `amount` or USD `cost`) and their immediate price impact. It also captures whether a trade was via the legacy AMM or the newer "NegRisk" (possibly order-book style) exchange (`Trade.exchange` field) – and even links the taker vs maker addresses [12], which helps identify if liquidity came from a particular provider or market-making bot. Additionally, open interest can be inferred using `Position` data or token events: a `Split` event indicates new outcome tokens minted (increasing open interest) while a `Merge` event indicates tokens destroyed for USDC (decreasing open interest) [13] [14]. Thus, the **historical trade tape** and position changes needed for microstructure analysis are largely available via the subgraph.

However, the subgraph does **not** store order book snapshots or unfilled orders – it only records executed trades. There is no direct on-chain record of the **top 5 bid/ask levels** at each moment (which the strategy spec suggests was accessed by polling an API every second). The subgraph also lacks event-driven granular timestamps beyond block time (trades are timestamped to the second, but there's no millisecond-level order sequencing). Despite these limitations, one can approximate certain microstructure signals by analyzing the sequence of trades. For instance, an absence of trades following a big price move can imply lack of follow-through interest (one of the criteria for a "transient spike"). Similarly, trade `price` and `amount` give clues to order book depth that was consumed by that trade (a huge `amount` with a big price jump implies a thin order book). The subgraph's `fee` field allows incorporating trading fees into the strategy's cost calculations [15], and knowing the `counterparty` might help identify if the liquidity came from a known automated market maker address or a whale user.

Missing Data & External Sources: To fully utilize microstructure strategies, **real-time order book data** and more frequent updates are needed beyond what the subgraph provides. The Polymarket **CLOB API** (central limit order book) or WebSocket feed is necessary to get **current bid/ask quotes and depth** in live trading. For backtesting, if historical order book snapshots were not recorded, one might need to approximate them. The subgraph alone doesn't include unexecuted orders or the state of the automated market maker at each moment. An alternative is to use Polymarket's Data API (Gamma) to fetch historical price points at higher frequency or reconstruct an implied order book using the constant product market maker formula (if applicable in legacy markets). In practice, **Gamma API** provides off-chain metadata and possibly endpoints for recent trades or candles, but historical full-depth snapshots per second would likely require one's own data logging. Additionally, **external timestamps for real-**

world events can be relevant – e.g. knowing when an important news article or tweet came out could explain a sudden spike (though purely microstructure strategies intentionally avoid using news and rely only on *market-internal* data).

In summary, the subgraph gives all the **executed trade data** needed to develop and backtest microstructure models in terms of identifying patterns of price movement and volume. What's missing is the **live order book state** and ultra-low-latency feeds – for backtesting, one may simulate fills at last traded price or assume you trade within the next tick. **Polymarket's current infrastructure (subgraph + CLOB API)** can support automation: the subgraph for research/backtest, and the CLOB API for placing real orders at specific prices. The feasibility of backtesting high-frequency strategies is moderate: you can replay historical trades to test trigger conditions (e.g. detect a 15% price jump and then assume you execute a counter-trade at say the 10-second later price). Without recorded order book, you can't precisely know the best entry/exit price *within* those seconds, but using the sequence of trades as a proxy is a reasonable approach. Many microstructure signals (like **order flow imbalance, short-term mean reversion, and arbitrage** opportunities) are detectable from the subgraph trade history [11], so one can definitely develop and partially backtest these strategies. **Automation feasibility:** High for reactive strategies (since Polymarket now offers a live CLOB API for execution), but success will depend on having a robust real-time data feed and careful testing of assumptions made in backtesting due to missing order book data.

2. Semantic Correlation Arbitrage Strategies

Description: Semantic correlation arbitrage strategies exploit relationships between **different markets or events**. The idea is to leverage *external knowledge or event logic* – for example, if two Polymarket questions are related (say *Market A: "Will Candidate X win the election?"* and *Market B: "Which party will control the senate?"*), their outcomes are correlated. A semantic arb trader might buy or sell across these markets to arbitrage discrepancies: e.g. if Market A's price implies Candidate X is very likely to win, but Market B's price doesn't reflect the same political leaning, there may be profit in aligning them. Another example is using **real-world news or data** as a trading signal: if a breaking news story should affect a market's probability but the price hasn't moved yet, that's an opportunity (a form of *event-driven trading*). This category includes **cross-market arbitrage** (trading *between* Polymarket markets that have a logical relationship) and **external parity trading** (trading Polymarket vs an external prediction or odds). It often involves **NLP or semantic analysis** – e.g. parsing news articles or tweets (hence "semantic logic") – to anticipate price moves, or simply recognizing that one outcome implies another. We can also include **statistical correlation** pairs trading: finding markets that historically move together and betting on convergence if they diverge, even if the relationship is not strictly logical.

Data Requirements: These strategies need **contextual and relational data** about markets, which the on-chain trades alone do not provide. Key requirements: - **Market metadata:** understanding what each market is about (the question text, category, resolution criteria, and timeline). This is crucial to identify related markets or to know which external real-world variables influence it. For example, to correlate an election market with a polling data trend, one needs the mapping from the market's `questionId` to "Election for X on date Y". - **Cross-market price data:** the historical price series of multiple markets, to compute correlations or to detect when one moves and the other lags. This can be obtained from subgraph trades (for each market) but may need alignment by time. - **Event timing and external triggers:** knowledge of when events happen (debates, earnings reports, news releases, etc.) and possibly real-time news feeds. If doing true *semantic arbitrage*, you'd integrate a feed of relevant news (perhaps filtered or processed by an NLP model to detect sentiment or implications for market outcomes). - **Off-chain or external price sources:** sometimes you might arbitrage between Polymarket and another platform's odds. In such a case, you need data from the other source (e.g. odds on Betfair, centralized exchange, or an internal predictive model's implied probability).

Subgraph Coverage: The Polymarket subgraph **provides robust historical price and volume data for each market**, which is useful for analyzing correlations and co-movements. One can query all trades or daily price points for two markets and check if price disparities opened up. However, **crucially, the subgraph does *not* contain any human-readable market descriptions** or grouping of related markets [16]. Fields like `Market.questionId` are just 32-byte hashes on-chain, and `Market.oracle` gives an oracle address but not the event details. This means purely from subgraph data it's impossible to tell which markets are semantically related (e.g. two markets might both be about the same election or sport, but the subgraph has no notion of that). The subgraph also lacks **market end times or resolution date** [17], which are important for timing strategies (a market approaching its resolution behaves differently than one far from it).

What *is* available: - **Unique market IDs and outcomes:** Each `Market` has an ID and tracks whether it's resolved or not, and `winningOutcome` if resolved [18]. This at least allows filtering out resolved markets in live trading and, post-resolution, using outcome results for analysis. - **Historical price series:** by iterating over `Trade.timestamp` and `Trade.price` for a given market, one can reconstruct the probability time series. This is crucial for finding correlated movements. The subgraph's complete trade log enables, for instance, computing the correlation of Market A's daily closing price with Market B's, or detecting if Market A's price moved significantly 10 minutes before Market B's price did (a lead-lag relationship). - **Volume and participation data:** `Market.totalVolume` and `uniqueTraders` can indicate market popularity [19]. A strategy might focus on major markets (with many traders) where information flow is more robust, or conversely on niche markets where mispricings might be larger. The subgraph can identify if *the same users* are active in two markets via `MarketParticipation` or by cross-referencing `User` trades, which could hint that the markets share a common interest group (but this is a subtle signal).

Missing Data & External Sources: To unlock semantic arbitrage, **off-chain data integration is essential**. The **Gamma API** (Polymarket's metadata API) fills many gaps: it provides **human-readable market information** like the question text, category, tags, and possibly the resolution date or event timeline [20]. By querying Gamma, one can group markets by event or theme (e.g. all markets related to "COVID-19" or all markets under an "Election 2024" event). This grouping is critical: it lets an algorithm know that Market X (e.g. "Will Candidate A win?") and Market Y ("Will Candidate B win?") are opposites or related (perhaps complements in a multi-outcome event). Gamma's data structure includes *Events* that can contain multiple markets [21], thereby enabling semantic linking.

Additionally, for true semantic signals, one would integrate **external event data**: for example, if doing news-driven trading, you'd use news APIs or social media feeds. A *Semantic Logic Oracle (SLO)* might process news in real time (even using NLP or a knowledge graph) to output a signal like "News of injury to player – related championship market likely to move." In backtesting, incorporating this is challenging – one would need historical news archives or at least timestamps of major news to simulate the strategy. Without that, one can approximate semantic arb by looking at price moves in one market as a proxy for news and seeing if another related market lagged behind (essentially a lead-lag statistical arb).

Feasibility of Automation & Backtesting: With the current infrastructure, **cross-market statistical arbitrage** can be studied and automated fairly well. For instance, if Market A and B historically move together, one can use subgraph data to quantify their relationship (correlation, cointegration) and backtest a strategy that buys one and sells the other when they diverge. The subgraph provides all necessary price history for that analysis. One can then automate such a strategy by continuously querying the subgraph or a real-time feed for both markets' latest prices and executing trades when a threshold is met. The main challenge is latency and market impact – but since these are slower strategies (changes over minutes or hours), using the subgraph or a periodic API call is sufficient.

However, **semantic/news-driven arb** (reacting to external information faster than the market) is trickier. Backtesting it requires aligning external data with market data. If one has a log of event timestamps (e.g. when a poll result was announced) one could check if Polymarket's price moved immediately or with delay. The subgraph alone won't have that external timeline, so it must be combined with external datasets. Automation in live trading would require real-time monitoring of news (outside the Polymarket system) and then using the Polymarket API to trade – this goes beyond the subgraph, but the subgraph remains useful for *development*: you could simulate "What if I traded whenever a certain external signal happened?" by looking at historical price changes around those times.

In summary, **semantic correlation strategies** need significant enrichment from **Gamma API** (for market semantics) and potentially other data sources (news, off-chain odds). The **subgraph is an excellent source of historical pricing for multiple markets**, enabling rigorous backtests of relationships and the ability to detect past arbitrage opportunities. Automation is feasible especially for cross-market arbitrage *within Polymarket*, as all trading can be done on the same platform (and one can use the Polymarket trading API to simultaneously place orders on multiple markets). The main limitation is that you must build an external knowledge pipeline (SLO) to drive these strategies – the subgraph alone is not enough for the "idea" of the trade, only for the numbers to execute it.

3. Price Parity & Statistical Arbitrage Strategies

Description: These strategies focus on **pricing consistency and statistical fair value**. In prediction markets, **price parity** refers to the fact that related outcome prices should align with logical constraints. The simplest case is a binary market: the "Yes" and "No" outcomes should sum to ~$1 (100%) when ignoring fees. If at any moment Yes is trading at 0.60 (60%) and No at 0.60 (60%), there's a clear arbitrage – the two sum to 120%, implying a free money opportunity by shorting both. A trader can deposit 1 USDC to **split** into 1 Yes and 1 No token [22], sell both for $1.20, and later redeem nothing (because one of them will lose) – netting $0.20 risk-free. Conversely, if Yes+No sum to < $1 (say 0.40 + 0.40 = 0.80), one can buy both outcomes for $0.80 total and later **merge** them to redeem 1 USDC [14], pocketing $0.20. This is a pure **on-chain arbitrage** since Polymarket's smart contracts allow splitting and merging outcome tokens freely. More generally, in a multi-outcome market (say outcomes A, B, C that are mutually exclusive), the sum of probabilities should equal 1 (100%). Statistical arbitrage extends this idea: maybe two different markets should have the *same* price (parity) because they are effectively the same proposition, or a basket of markets has a known linear combination that should sum to a certain value. Another angle is arbitraging **Polymarket vs external odds** – e.g. if Polymarket's yes price is 0.60 but Betfair's odds imply 0.65, one could buy on Polymarket and hedge on Betfair. However, here we'll focus on *on-platform parity and statistical mispricings*. These strategies are usually **market-neutral** and can be automated as **arbitrage bots**.

Data Requirements: Key data needed is **synchronous price information for all related outcomes** to detect parity deviations. For a single market (binary or multi-outcome), one needs the prices of each outcome at the same point in time. For cross-market parity (e.g. two separate markets that are logically linked), one needs both market prices at the same time. Additional requirements: - **Fee and transaction cost info:** Polymarket charges fees on trades (and possibly a fee on outcome redemption). Knowing the fee rate is crucial because a tiny deviation like 0.99 vs 1.00 might not be profitable after fees. The `Trade.fee` field and global fee stats can help incorporate this [23]. - **Liquidity info:** an arb might exist in theory, but the depth might be insufficient to exploit fully. So ideally one knows how much volume is available at the mispriced quotes. In practice, an arbitrage bot might only execute if the order book has enough depth or the profit margin is above some threshold. - **Position management:** since arbitrage often involves holding opposite positions (e.g. long Yes and long No simultaneously from different actions), tracking the inventory and ensuring it can be resolved to cash is needed. For

example, if you short Yes by splitting, you end up holding an equivalent No position as collateral – you must track that to later merge it. The data on total positions and ensuring you don't exceed your capital are relevant.

Subgraph Coverage: The subgraph contains **all trade executions for each outcome token**, which can be used to infer parity. Specifically, by looking at the latest trades on each outcome, one can estimate the current price. If the market is liquid and frequently traded, the last trade price is a good approximation of the midpoint. The subgraph's real strength is that it **records the actual arbitrage actions** when they happen: `Split` events (collateral to outcome tokens) and `Merge` events (outcome tokens to collateral) [13] [14]. These events are direct evidence of parity arbitrage in action. For example, if at time T an arbitrage condition existed, you will often see a `Split` transaction where a user minted tokens, followed by trades that bring prices back in line. By querying these events, one can identify when and how often the market had mispricing. This is invaluable for backtesting: you can confirm that whenever your algorithm would have acted, someone indeed did (or didn't), and calculate the profit. The subgraph also provides `Redemption` events for claiming winnings [24], which matters if your strategy holds positions through resolution – but an arbitrage strategy usually closes (merges) before resolution.

For multi-outcome markets, the subgraph's data model (Conditional Tokens framework) ties all outcomes of a market together under one `Market` ID, making it easier to fetch all outcome prices. For example, one could query all `Trade` entries for a given `Market` sorted by time and separate by `outcomeIndex` to see if the sum of probabilities deviated from 1. Another approach: use `TokenRegistry` to get all outcome token IDs for a market [25], then query the latest trades for each token. Since Polymarket outcomes trade against USDC, one can compare the combined cost of buying a complete set versus 1 USDC.

Missing Data & External Sources: In theory, the necessary on-chain data for parity checks is mostly there. One challenge is **synchronization of prices** – the subgraph gives discrete trades. If outcome A traded 5 minutes ago at 0.4 and outcome B just traded now at 0.65, is that a mispricing or just stale price for A? It might require assuming that if one outcome hasn't traded recently, its price is roughly the last trade or use an interpolation. A real arbitrageur would be watching the live order book to see current quotes for both outcomes simultaneously. Therefore, using the **CLOB API** or a live price feed is important to do real-time parity arb. The subgraph can tell us historically that such gaps existed, but to act on them, a bot would need to fetch the current order book on both sides (or at least the best bid/ask for Yes and No). Polymarket's off-chain order book (if using one) or the on-chain AMM formula can be used to calculate the *instantaneous price* for a small trade – a backtester might simulate: "if I try to buy 10 Yes right now, what price do I get and what happens to No's price?".

For **external parity** (Polymarket vs other markets), obviously one needs that external data source. For example, if doing statistical arbitrage between Polymarket and another prediction market or betting exchange, the external price history must be obtained separately. That's outside the subgraph's domain but can be integrated in backtesting if available (e.g. downloading odds history from the other platform).

Feasibility of Automation & Backtesting: Parity arbitrage is the most straightforward to backtest and automate with the current infrastructure. The rules are clear (e.g. execute split/merge when sum of prices off by >X%) and the subgraph provides the needed historical evidence to test these rules. One can write a backtest that goes through each market's trade stream and computes the implied sum of outcome probabilities after each trade, checking if it crosses an arbitrage threshold. The backtest would simulate performing a `Split` and subsequent trades to realize profit (accounting for fees).

Because the subgraph even tracks fees and volumes, you can accurately model the profit. Automation is highly feasible: one could run a **bot service** that monitors prices (via a GraphQL subscription or the Polymarket API) and automatically submits on-chain transactions to split/merge and trade. Polymarket's design encourages this because it relies on arbitrageurs to keep prices aligned (in the AMM model, if someone buys a lot of Yes, an arbitrageur should come and buy cheap No to restore balance, etc.). The presence of 17 million trades in 3 years [26] and explicit mention of **arbitrage detection** as a possible analysis [27] suggests that arbitrage activity is detectable and presumably executed by some users. With the subgraph, one can even identify addresses that frequently perform splits/merges and classify them as arbitrageurs [28].

One consideration is that **Polymarket fees and blockchain transaction costs** eat into arbitrage margins. The subgraph's global stats on fees collected [23] can help calibrate how wide a mispricing must be to profit. Also, arbitraging requires capital in USDC (for splits) or in both tokens (for merges), so strategy must manage inventory – but the subgraph's `Position` data can be used to simulate the inventory and ensure that after a series of arbitrages your net holdings and PnL are tracked correctly.

In conclusion, parity/statistical arbitrage strategies are **highly data-supported by the subgraph** and relatively easy to test. For live deployment, integration with the **CLOB API** for real-time quotes and an automated **execution engine** to perform splits/merges and trades is needed. The strategy logic itself is straightforward to implement given the data. Backtesting reliability is high, since arbitrage conditions involve discrete conditions (sums of prices) that can be checked against historical trade data with precision.

4. Long-Horizon Trend & ML-Based Strategies

Description: These are strategies that operate on a **longer time scale** (hours, days, or weeks) and often incorporate predictive modeling or machine learning. Examples include: - **Trend-following or momentum**: e.g. if a market's probability has been steadily rising due to accumulating information, the strategy might continue to go long (buy Yes) expecting the trend to continue (until perhaps just before resolution). Conversely, **mean-reversion** strategies might short overly rapid moves expecting a correction over multi-day periods (distinct from the very short-term mean reversion in microstructure). - **Fundamental or ML-driven prediction models**: using historical data and possibly external features to predict the fair value of a market. For instance, training a model on past similar events to forecast the probability, or using an ensemble of predictors (polling data, sentiment, etc.) to derive a probability and then trading when Polymarket's price deviates from that. - **Portfolio strategies**: constructing a portfolio of positions across many markets to maximize long-term returns (could be informed by ML classification of outcomes or hedge ratios). - **Long-horizon arbitrage**: This could include things like exploiting the time decay of prices as an event approaches (e.g. if a "Yes" should converge toward either 0 or 1 by resolution, perhaps there's a pattern to how mispricings decay).

These strategies generally involve **time-series analysis** and often **supervised learning or statistical modeling** beyond simple rules. They might also consider broader patterns (like seasonality in prediction markets, or how markets react to recurring events such as monthly economic reports).

Data Requirements: For long-horizon strategies, one needs **historical time-series data** at an appropriate granularity (hourly, daily, etc.) and possibly a **large feature set** (news sentiment, macro variables, etc.) for ML models. Key requirements: - **Time-series of prices and volume** for each market over its lifetime. This could be OHLC (Open/High/Low/Close prices) by day or hour, trading volume per interval, volatility measures, etc. - **Outcome resolution data** (the actual outcome of each market). For example, if you build a predictive model, you need to train it on past events' outcomes. The subgraph gives the `winningOutcome` for resolved markets [18], which serves as the ground truth label. -

Features describing markets: e.g. days to resolution, type of event, number of traders (could indicate how information-rich it is), etc. Some of these features come from on-chain data (e.g. we can derive how far from creation we are, or how many users traded = `Market.uniqueTraders` [19]). Others like event type or external data (polling numbers, team statistics, economic indicators) are off-chain. - **Global or sector data**: The overall market sentiment could be relevant. For instance, if the entire platform sees a volume spike (maybe due to a surge of new users or a big news cycle), individual markets might show correlated trends. The subgraph's `DailyStats` provides daily aggregate volume, user count, new markets, etc. [29] which can be a proxy for "market-wide sentiment" or regime shifts over time. - **Risk metrics**: Over long horizons, you need to manage risk (drawdowns, diversification). So data on **covariances** between markets or maximum historical swings is useful. Those can be computed from price history as well.

Subgraph Coverage: The subgraph is well-suited for providing **training and backtesting data for long-term strategies**. Since it has every trade, one can derive any timeframe's candles or statistics from it. For example: - To get daily price and volume for a market, you could aggregate `Trade.price` and `Trade.volume` per day. The subgraph even has a `DailyStats` entity, but note it's global for the whole platform, not per market [30] . There isn't a built-in per-market daily time series, so you'd compute that yourself using trades. Still, the raw data is there to build indicators like 7-day moving average, ATR (average true range), etc. - The `Market` entity gives some static info like creation time and whether it resolved. If resolution happened on-chain, the `Market.resolutionTimestamp` (once synced) and `winningOutcome` can be used to mark the end of the series and the final payout [18] . - **User and position data** from the subgraph can provide interesting features for ML or long-term strategies. For instance, `Market.uniqueTraders` might correlate with market efficiency: a market with thousands of traders might incorporate information faster than one with few traders. `User` data can identify if a few "whales" hold a large position (though you'd have to deduce that by scanning `Position` balances). The subgraph can let you measure concentration (e.g. top 1-2 traders volume share in that market) which could predict future volatility or mean-reversion. - **GlobalStats and DailyStats** allow you to place a single market's behavior in context. For example, if your model needs to know overall market activity, `dailyStats.activeUsers` or `dailyStats.volume` could be an input [29] .

In short, the subgraph provides a *complete historical dataset* to train models and backtest hypotheses. Every outcome's historical price trajectory and final result are available, which is ideal for supervised learning (e.g. train a classifier that given features at time T predicts if outcome will resolve Yes or No, akin to a forecasting model).

Missing Data & External Sources: While historical on-chain data is rich, many long-horizon strategies benefit from **off-chain data**: - **Market metadata** (via Gamma API) to incorporate features like category or event description. An ML model might want to know if a market is about sports, politics, crypto, etc., since different domains have different dynamics. Gamma can supply a market's category/tag and resolution date (if known). For example, knowing a market's end date allows adding a feature like "days until resolution" or a time-decay factor. - **External fundamental data:** If predicting an election market, one might use polling numbers; for a crypto price market, one might use actual price of the underlying asset; for sports, team stats or elo ratings; for economy, actual economic indicators. None of that is in the subgraph, but they can be merged in during development of ML models. - **Alternate prediction sources:** Some strategies might compare Polymarket odds to a trusted predictor (like a prediction model or another market's odds) and trade on the gap. That requires pulling that alternate data stream. For example, one could use Metaculus predictions or an ensemble from PredictIt and see if Polymarket lags. - **News/NLP sentiment:** On longer horizons, the cumulative effect of news can drive trends. If one has a sentiment index or news count for an event, feeding that into a model could improve it. That data must come from outside (news API or web scraping), possibly processed via NLP into numeric features.

Feasibility of Automation & Backtesting: Developing and backtesting long-horizon and ML strategies is **highly feasible** with the subgraph data. You can easily slice out, say, the past 3 years of all markets in a certain category, train a model, and simulate its performance. The main limitation is that many ML features are external, so one must assemble a dataset that combines subgraph data with those external sources – which is a one-time data engineering effort. The subgraph's data can be exported to CSV or a database for ML work.

When moving to **live trading**, these strategies are typically low-frequency (maybe daily trades or a few per event). That means using the subgraph or Gamma API in real-time is not a latency issue – polling every few minutes or hours is fine. One could use the Gamma API's `/markets` endpoint to get current prices for all markets periodically, or maintain an index via TheGraph. The execution can often be done with limit orders if you want to minimize slippage (since you're not in a hurry like high-frequency arb). Polymarket's CLOB allows placing limit orders via API [31]. So a live long-horizon strategy might look like: model says "Market X is underpriced, true odds should be 70% vs market 60%" – you then place a limit buy order slightly below 0.60 to try to get in, and perhaps plan an exit before resolution or at resolution.

Risk management is a big part for long-horizon strategies. They might hold positions for weeks, so you need to monitor not only on-chain data but any changes in the underlying narrative. This is where a **Semantic Logic Oracle** could again come into play even for longer-term positions: e.g. if a sudden news breaks that invalidates your model's prediction, you might want to exit early. So an integration of SLO (monitoring off-chain info) with your ML strategy can provide alerts.

Overall, the **current subgraph infrastructure strongly supports** the research and development phase for long-term and ML strategies (data completeness, outcomes for ground truth, etc.). With additional data (Gamma metadata, external features) and a pipeline to continuously update signals, these strategies can be automated using Polymarket's APIs. The feasibility is high, given that execution speed is not critical and the data is largely historical – the main challenge is building a robust predictive model, which is outside the scope of data availability.

Summary of Data Needs vs Availability

To synthesize the above analysis, the table below summarizes each strategy type, the data it needs, what the Polymarket subgraph provides, what is missing (requiring external sources), and how feasible it is to backtest/automate with the current infrastructure:

Strategy Type	Key Data & Signals Required	Subgraph Data Availability	External Data Needed	Automation Feasibility
Microstructure-Based (e.g. order book arbitrage, short-term mean reversion, HFT range trading)	- Tick-by-tick trades (price, size, time) - Order book depth (bid/ask levels) - Short-term volume & volatility (seconds/ minutes) - Identification of large trades & order flow imbalances	Full history of executed trades with `price`, `amount`, `timestamp`, taker/maker (`trader/ counterparty`) [9] [10]. Can derive per-second price changes and volume from trade timestamps. Open interest changes via `Split` / `Merge` events [13] [14]. No direct order book snapshots or quotes between trades.	– **CLOB API** for real-time order book (top-of-book prices, depth). – Possibly on-chain AMM formula to compute price impact. – Ultra-low-latency feeds (if doing HFT) beyond subgraph's indexing speed. – (Optional) External clock synchronization or event timestamps to align trades.	**High** for backtesting patterns (trade data covers needed signals); **Moderate** for full realism (order book not in history). Automation is **feasible** (Polymarket API allows placing orders) but requires combining subgraph insights with live order book data.

Strategy Type	Key Data & Signals Required	Subgraph Data Availability	External Data Needed	Automation Feasibility
Semantic Correlation Arbitrage (e.g. cross-market arbitrage, news-driven trades, related outcomes)	- Market definitions (event details, categories) - Groups of related markets & their prices - External events or news triggers (timestamps, sentiment) - Lead-lag price relationships across markets	Historical price series for any market (from trade data) to compute correlations or detect lags. `MarketParticipation` and `User` data to see overlapping traders (optional insight into related markets). No on-chain metadata: question text, category, or event grouping not available [16]. No direct link between markets that are logically related.	– **Gamma API** for market metadata: human-readable questions, categories, tags, resolution time [20]. This allows linking markets by event or topic. – **External info feeds**: news APIs, social media, domain-specific data (polls, sports stats) for signals. – **Real-time price feed** for multiple markets (Gamma Data API or subgraph queries) to catch divergences promptly.	**Moderate** for backtesting (can identify related markets if metadata is merged; cross-market price data is available from subgraph). **Feasible** for automation with enriched data (SLO pipeline needed for news/ semantic signals; use Polymarket API to execute on multiple markets).

Strategy Type	Key Data & Signals Required	Subgraph Data Availability	External Data Needed	Automation Feasibility
Price Parity & Stat Arb (e.g. intra-market Yes vs No parity, multi-outcome sum, cross-platform arb)	- Simultaneous outcome prices (Yes vs No, or all outcomes) - Trade prices around mispricing moments - Fee rates and transaction costs - Ability to short (sell outcome you don't hold via splitting)	Every trade for each outcome token, enabling calculation of price sums. `Split` / `Merge` events signal arbitrage opportunities and how they were resolved [22] [14]. Known fee from trade data (fee per trade) [15] and possibly global fee rate settings. `Position` data to simulate holding both outcomes and final redemption.	– **CLOB/API for live quotes** to continuously monitor outcome prices in sync (subgraph has slight delay). – (If cross-platform) External market odds data feed. – (Optional) Bot logic to execute split/merge transactions on-chain.	**High** for both backtesting and automation. Historical mispricings can be identified precisely with subgraph data, and strategy logic is straightforward. Live trading bots can directly use on-chain actions; current infrastructure (The Graph + Polymarket API) supports this with minimal latency constraints (since arbitrage is relatively slow on-chain).

Strategy Type	Key Data & Signals Required	Subgraph Data Availability	External Data Needed	Automation Feasibility
Long-Horizon & ML-Based (e.g. trend-following, predictive models, portfolio optimization)	- Long-term price history per market (days/ weeks/ months) - Outcome resolution results for model training - Macro-level trends (market-wide volume, new users) - Features: time to event, event type, external predictors (polls, etc.)	3+ years of historical trade data to derive daily or hourly price series [1] . Outcome labels via `Market.winningOutcome` for resolved markets [18] . Platform-wide daily stats for context (active users, volume) [29] . User/Position data for features like number of traders or whale presence. Not directly providing event end dates or categories (needed for some features).	– **Gamma API** for event metadata: resolution date, category/tags (for features like “days until resolution”, event category dummy variables). – **External datasets** (polling data, financial indicators, etc.) for model input. – **News sentiment** aggregators if using NLP as feature. – (Optional) Alternative forecasts (e.g. expert predictions) to compare with market odds.	**High** feasibility. Subgraph provides the core dataset for backtesting strategies and training ML models (complete and labeled data). Combining with external data is straightforward offline. Live execution is **feasible** with low-frequency API polling and order placement (latency is not critical). SLO can enhance risk management by flagging relevant news.

(= available in subgraph; = not available in subgraph)

The table above highlights that **Polymarket’s subgraph covers nearly all on-chain trading data needed for strategy development**, especially for price-centric and arbitrage strategies. Missing pieces are mostly **off-chain context or real-time order book data**, which can be supplied by Polymarket’s **Gamma API** (for metadata and some data queries) and **CLOB interface** (for live trading/quotes), as well as external information sources for semantic signals.

SLO & MBO Pipeline Engineering

To effectively harness the data for trading signals, we propose a pipeline centered on two types of oracles: a **Semantic Logic Oracle (SLO)** and a **Market Behavior Oracle (MBO)**. These oracles serve as specialized analytics modules within the strategy system:

- **Semantic Logic Oracle (SLO):** This component ingests *external knowledge and contextual data* to interpret what the market ought to believe. It focuses on **meaning and fundamental logic**

around events. The SLO uses inputs like market metadata (from Gamma API), news feeds, social media (tweets, Reddit), and any structured data about the event (e.g. polling data for elections, team lineups for sports, economic indicators for finance markets). Its outputs are signals or adjustments related to *intrinsic value or logical relationships*. For example, the SLO might output: *"Market X (Election) is about to close in 2 days; polls show candidate leading by 10%, implying fair price ~0.90"* or *"News sentiment for Company Y has turned sharply negative, implying the probability of outcome 'Y stock above $Z' should drop."* Essentially, the SLO tries to be a **fundamental or event-driven oracle**, telling the trading system what the price *should* be or flagging when real-world developments occur.

- **Data integration:** The SLO pipeline will utilize the **subgraph's market structure** as a base: it knows the `Market.id` and on-chain `questionId` for each market, and via Gamma API it can fetch the actual question text and resolution date. It might maintain a mapping of `Market.id -> Event details` in an internal database. For semantic correlation strategies, the SLO can identify groups of markets that belong to the same event or theme. For example, if two markets share a Gamma `eventId`, the SLO knows they are related (like multiple outcomes of one event). It can then enforce logic like "if one outcome is trading at X, the sum with the other outcome should be 1" (though this particular logic might already be enforced via parity arb in MBO). For more complex correlations, SLO could know that *Market A resolves to Yes implies Market B likely Yes*, etc., forming a graph of implications.

- **External enrichment:** The SLO would incorporate **Gamma's off-chain data** (market descriptions, categories) and might call external APIs. For news, one could use webhooks or RSS feeds for specific keywords (e.g., the name of the event or key entities in the question). An advanced SLO might even use an **LLM or NLP** to parse news articles and extract structured info (this is suggested by Polymarket Agents framework, which integrates news vector databases and LLMs [32] [33]). For example, if a question is "Will XYZ win the Best Actor Oscar?", the SLO could track entertainment news, identify a breaking story about awards, and produce a bullish or bearish signal.

- **Output signals:** The SLO's output can be thought of as *recommendations or flags* that feed into strategy logic. It might be a target probability for a market, an alert that "market X and Y should converge" or a simple binary flag "avoid trading Market Z now because a critical announcement is imminent (high uncertainty)". In the microstructure spec's terms, SLO would address the risk of *"mispricing actually being real price" due to new info* [34] – i.e., if a large price jump is justified by news, SLO would catch that and prevent the MBO from fading the move blindly. In summary, SLO ensures the strategy is **context-aware**.

- **Market Behavior Oracle (MBO):** This oracle processes the **raw market data streams and patterns** to identify technical trading signals. It is grounded in the subgraph/on-chain data and focuses on **order flow, price patterns, and anomalies in trading behavior**. The MBO monitors things like price trends, volatility, volume surges, order book imbalances, and arbitrage conditions purely from the market's behavior. Essentially, it's the quantitative engine that detects setups like *"price just spiked by +15% on high volume"* (which might be a mean-reversion short signal), *"volume has been steadily increasing with price – momentum building"*, *"Yes+No price sum != 1 – arbitrage opportunity"*, *"whale address X is aggressively buying – momentum signal or potential informed trader"*, etc.

- **Data integration:** The MBO heavily uses **subgraph data in real-time** or near-real-time. It can subscribe to new `Trade` events (if using a GraphQL subscription or polling the subgraph's latest trades) to get a stream of trades. If very low latency is required, the MBO might instead

use the **Polymarket CLOB WebSocket** to get immediate order/trade updates. The subgraph acts as the historical memory for the MBO: it can query recent trades (last N minutes) from the subgraph or maintain its own sliding window of trades to compute metrics like short-term moving averages, RSI, order flow imbalance (e.g. count of BUY vs SELL in last X trades) [11]. It may also query the `Position` or `User` entities periodically to glean insights (for example, if a sudden move happens, MBO could check if it was a known whale increasing their position by looking at `User.totalVolume` or recent `trades` of that user – though this might be too slow for real-time and would require caching user profiles).

- If Polymarket's markets operate via an AMM, the MBO can also incorporate the known pricing formula to estimate the impact of trades. If it's order-book based, the MBO will incorporate **order book data** (likely from the CLOB API). For instance, MBO might continuously fetch top-of-book prices and sizes for each market. This allows it to calculate things like bid-ask spread (could signal a liquidity issue if spread widens), order book imbalance (more depth on bid vs ask side could signal direction), and identify when arbitrage conditions arise (e.g. if best bid on Yes and best bid on No sum to >1 USDC, that's an arbitrage to sell both to those bids).
- **Output signals:** The MBO produces *technical trade signals* such as:
  - **Mean-reversion signal:** e.g. "Yes price is 3 standard deviations above its 5-minute moving average after a single trade" – potential sell (short) signal.
  - **Breakout/momentum signal:** e.g. "Price broke a long-term range with high volume" – maybe buy signal if strategy is trend-following.
  - **Arbitrage signal:** "Outcome prices out of line (or Polymarket vs external odds out of line)" – execute arbitrage (split/merge and trade).
  - **Order flow signal:** "80% of last 100 trades are buys and price is climbing" – indicates strong upward pressure (could be momentum signal or, if SLO disagrees, a contrarian signal).
  - **Liquidity signal:** "Market depth significantly reduced (order book thin)" – maybe adjust strategy or set wider stops.
  - **Risk alerts:** e.g. "Volatility in this market is the highest in 6 months (from subgraph dailyStats or internal calc)" – could trigger reducing position size.

The MBO essentially is the **algorithmic brain that reads the tape**. It relies on subgraph for historical context and on live data for instant decisions.

Integration of SLO & MBO: In a complete system, these two oracles work **in tandem**. Think of SLO as providing **contextual filters and fundamental biases**, and MBO providing **market-timing and technical triggers**. Some examples of their interaction: - **Confirmation and filtering:** An MBO signal to short a spike should be cross-checked with SLO: *Did SLO detect any new information that justifies the spike?* If yes (e.g. a news event), the trade might be filtered out or handled with caution (maybe smaller size or skip entirely) [34]. If SLO finds no news ("this looks like a pure order-book anomaly"), then the confidence in the mean-reversion trade is higher [35]. - **Opportunity identification:** SLO might highlight that two markets should be priced similarly (say they are duplicates or one is logically implied by another). MBO can then monitor the price spread between them and signal when it widens beyond a threshold to execute the arbitrage. Here SLO sets the *theoretical model* ("these should converge"), and MBO looks for the *execution timing* ("they have diverged now, execute pair trade"). - **Event risk management:** SLO knows event dates; if a market is about to resolve (say hours away from an election result), SLO can flag elevated risk of volatility or information influx. MBO, on receiving that, could switch strategies (e.g. stop doing range trading and maybe only do very short-term trades or pull out entirely to avoid being caught by last-minute news). Essentially, SLO can dynamically adjust MBO's parameters or enable/disable certain strategies based on context (a form of regime switching). - **Combining signals:** For a long-horizon ML model (which might be part of SLO's output), if it predicts a certain

probability, the MBO might use that as an input to its strategy. For example, if the model thinks Yes should be 80% but currently it's 60%, the MBO could gradually buy (momentum strategy in line with fundamental value). Meanwhile, if the model and recent market behavior disagree (model says 80% but price is dropping on heavy volume), the system might hold off or investigate why (maybe new info model doesn't have).

Pipeline Implementation: Implementing this SLO/MBO pipeline involves creating modular components that share data: 1. **Data Input Layer:** The base layer collects data from the subgraph and external sources. It might consist of: - A subgraph GraphQL client that can fetch recent trades, query market info, etc. at regular intervals or via subscriptions. - A Gamma API client for metadata (market names, events, resolution dates, etc.). - Connections to external feeds (WebSocket for order book, news API clients, Twitter stream, etc.). This layer feeds raw data into the oracles. 2. **Semantic Logic Oracle Module:** Code that subscribes to relevant external feeds and metadata. It could update an internal state like: "Market 0x123: ends on 2025-12-31, category=Elections, related markets = [0xabc, 0xdef] (same event). Latest news sentiment = positive." It might use rules or ML models to output signals such as recommended price or a directional bias for each market. 3. **Market Behavior Oracle Module:** Code that processes market data streams (trades, order book updates). It maintains indicators per market (e.g. moving averages, recent volatility, order flow stats). It detects conditions (spike, breakout, divergence) and emits trading signals or alerts. The MBO can also use on-chain data to detect **whale activity** (for example, if a single address accounted for a large share of volume today, MBO can note that). 4. **Signal Coordinator/Strategy Logic:** This component receives inputs from SLO and MBO and makes decisions on actual trades. It implements the strategy rules, effectively **merging the oracles' insights**. For instance, a strategy might be encoded as: "If MBO signals mean-reversion SHORT and SLO context is clear, then execute SHORT position with size determined by risk limits; else if SLO indicates information risk, ignore signal." This layer can also manage multiple strategy types simultaneously (one for arbitrage, one for trend, etc.), each with conditions that might involve both oracle's data. 5. **Backtesting Engine:** It's prudent to have a module that can take historical data (from subgraph, plus recorded news if available) and simulate the above logic. This engine would use the same SLO and MBO logic, but feed them historical data in a time-stepped manner, and record the trades and PnL. Because our data sources are modular, one can replay say July 2023 by feeding the trades of that period from the subgraph and historical news headlines to the system to see how it would have performed. The backtester needs to simulate order execution (e.g. assume if we send a trade, it executes at the next trade's price or within the order book's depth). 6. **Execution Module:** When live, the system needs to place and manage orders on Polymarket. Polymarket offers APIs/SDKs for this (as noted in the Polymarket Agents repo, there's a `Polymarket` class to execute orders on the DEX [31], and also Python/TypeScript CLOB clients for signing and sending orders [36]). The Execution module will take trade decisions from the strategy logic and interface with these APIs. It must handle order placement (market or limit), transaction signing, and monitoring for fills. Since Polymarket uses Polygon, transactions are fast but costs and confirmation times must be considered – the execution module might choose between a direct on-chain trade (taker) or placing an order (maker) depending on strategy. 7. **Risk Management & Monitoring:** A separate thread or module should continuously monitor risk parameters: total exposure per market, aggregate USDC at risk, worst-case scenarios, etc. It uses data like our current positions (which could be tracked internally or even cross-checked with subgraph's `Position` entity for our trading address to ensure accuracy). It also uses subgraph data like volatility or volume to adjust risk models. For example, if a market's volatility triples (maybe detected via DailyStats or recent trade variance), the risk module could cut our allowed position in half. It can also enforce constraints like "no more than X USDC exposure in any single market" (maybe proportional to market volume). In case of breaches, it instructs the strategy to scale down or sends immediate orders to hedge.

In engineering terms, this pipeline could be implemented as a set of **microservices or modules** communicating through a message bus. For instance: - A service for data ingestion (subgraph & Gamma) populating an internal state or database. - SLO service that updates context and pushes events (like "Market 123 news_alert"). - MBO service that listens to trade events and pushes technical signals. - A central strategy engine (could be one process or multiple per strategy) that listens to both and decides trades. - A backtester which can run the above in a simulated mode using historical data. - Execution service that receives trade orders and interacts with the Polymarket API/CLOB (including handling keys, nonce, etc.).

This modular approach aligns with the Polymarket Agents architecture which also separates data sources (Gamma, etc.) and uses AI agents [37] [31]. By keeping SLO and MBO modular, one can iteratively improve each (e.g. plug in a better NLP model for SLO without changing the rest, or refine an indicator in MBO).

Mapping Subgraph Data to Strategy Pipeline Components

Polymarket's subgraph schema defines several entities (`Market`, `Trade`, `User`, `Position`, `DailyStats`, etc.). We map these data fields to how they support various components of our strategy pipeline: **signal generation (via SLO/MBO)**, **backtesting and simulation**, **PnL calculation**, and **risk management**.

- **Market Data (`Market` entity):** The `Market` fields provide context and lifecycle info.
- *Signal Generation:* `Market.id` and `questionId` are identifiers to link with Gamma API for SLO to fetch semantic info. Knowing `outcomeSlotCount` tells us if it's binary or multi-outcome (important for parity strategies and how we compute probabilities). `Market.oracle` might group markets that share an oracle (some oracles resolve multiple related markets, which could hint at correlation). In MBO, `tradeCount`, `totalVolume`, and `uniqueTraders` inform us of liquidity and activity – a strategy might require minimum volume to engage a market. Low `uniqueTraders` could mean one player dominates (risky for manipulation).
- *Backtesting:* `creationTimestamp` and `resolutionTimestamp` demarcate the period we care about for that market's data – our backtester would simulate trades only between creation and resolution. `resolved` and `winningOutcome` are crucial for backtesting outcome-dependent strategies: they allow the simulator to settle bets correctly at the end (if our strategy holds a position through resolution, we can determine PnL by whether we held the winning outcome). For example, if we predict "Yes" and hold until resolution, `winningOutcome` tells if we get payout or not.
- *PnL Attribution:* Markets act as segments for PnL. We may compute PnL per market to see which events the strategy did well on. `Market.totalVolume` can be used to normalize PnL (was profit proportional to volume?). Also, if `Market.fee` information were present (not explicitly, but we can sum trade fees in that market), we could attribute how much cost was paid in each market.

- *Risk Control:* `Market` fields like `totalVolume` and `tradeCount` serve as risk indicators – the strategy might avoid markets below a volume threshold (to reduce slippage risk). Also, if a market is nearing resolution (`resolutionTimestamp` approaching), risk control might reduce exposure due to potential unpredictability or closing auction volatility. We know Polymarket doesn't have an explicit "end time" on-chain [17], but we might approximate it from Gamma or by noticing no trades after a certain point.

- **Trade Data (`Trade` entity):** Each trade is a fine-grained data point for price and volume.

- *Signal Generation:* This is the primary input for the MBO. `Trade.price` provides the current price point (probability) at that timestamp. By streaming these, we get the price time series for technical signals (moving averages, breakouts). `Trade.side` and the presence of `trader` vs `counterparty` let us infer order flow (e.g. if many trades have side="BUY" by different traders, that's buy pressure). The trade `amount` and `cost` (in USDC) allow calculation of volume and detection of large trades (e.g. if `cost` >> average, a whale trade happened). We can flag trades above a USD threshold as potential **large order events** (like the spikes in strategy A1) [5]. `timestamp` is used to sequence events and calculate time-based indicators (like number of trades per minute, or time gap between trades which can signal liquidity gaps). The subgraph's trade data even enables reconstructing order book changes in an inferred way: e.g., if a single trade moves price substantially, it implies multiple order book levels were taken out.
- *Backtesting:* In simulation, we typically step through historical trades as the "time ticks" where the world changes. Our backtest engine can simulate reacting right after each trade or at fixed intervals. When simulating an order execution, we might assume we fill at the next trade's price (if we acted just after a trade). If our strategy places limit orders, we would need a more complex simulation (perhaps injecting our own orders into the historical order flow). Trade data also provides the ground truth to compare our simulated trade decisions: e.g. if strategy predicted a price drop, did the subsequent trades actually go down? We measure model precision by what the trades did.
- *PnL Calculation:* Each executed trade (both historical and our simulated strategy's trades) contributes to PnL. The subgraph's notion of cost and fee can be reused: e.g. realized PnL from a round-trip trade = selling price - buying price, adjusted for `fee`. We will incorporate the `fee` field from each trade our strategy would have done to get net P/L (since Polymarket charges fees on trades [15]). If our strategy is market-making (placing both buy and sell), the trade record tells which side we were on and how much fee we earned or paid.

- *Risk Control:* Trade data helps monitor execution quality and slippage. If our last few trades executed at worse prices than expected (e.g. strategy wanted to sell at 0.60 but by the time our order executed the price in `Trade` log was 0.55), the system can flag that slippage is high and perhaps reduce order sizes. Also, analyzing sequences of trades can warn of regime change: e.g. if volatility (price jumps per trade) suddenly doubles, risk module could tighten stops or cut positions. In real time, if we see a series of unusually large trades (maybe a sign of an informed trader entering), risk control might hedge or pause (unless our strategy specifically follows large traders). In essence, the live monitoring of `Trade` stream is part of risk management to detect abnormal market behavior.

- **Position Data (`Position` entity):** Positions track user holdings of outcome tokens.

- *Signal Generation:* Generally, positions are less directly used for generating signals compared to trades and market data. However, one could derive interesting signals from position data: for example, **open interest** (total tokens in circulation) can be computed by summing all `Position.balance` for an outcome. A rising open interest might indicate new money coming in (bullish signal if one side is being minted heavily). If we identify certain **whales** (via `User` data) and monitor their `Position` in a market, noticing a whale accumulating a large position over time could be a bullish signal (they might know something) – albeit this is complex to do in real time unless you track specific addresses. In practice, these are advanced uses; many strategies may not directly query `Position` for signals.
- *Backtesting:* We will use the concept of positions internally to track the strategy's simulated holdings. The subgraph's `Position` schema gives a template for what to record: `balance`, `avgBuyPrice`, `avgSellPrice`, and `realizedPnL` [38]. We might not literally use the subgraph to track our test strategy (since subgraph is read-only for historical data), but we

mimic it. For example, as our strategy buys and sells in simulation, we calculate the VWAP of buys and realized PnL on sells the same way the subgraph does. This ensures our backtest PnL attribution is accurate. We can cross-verify with the subgraph's data when applicable: e.g. if we simulate acting exactly as a historical user did, we should get the same `Position.realizedPnL` as recorded on-chain [39]. That's a good consistency check.

- *PnL Attribution:* In live trading, we could query our own trading wallet's `positions` via the subgraph to verify PnL and holdings. The subgraph would show our average entry price (`avgBuyPrice`) and realized profits. It's a convenient way to reconcile our internal accounting. For strategy analysis, we can aggregate PnL across positions: for example, sum `realizedPnL` for all positions closed in a month to report profit, or identify which markets gave the highest PnL (maybe find strengths/weaknesses of the model).

- *Risk Control:* Positions are central to risk. The risk module will use `balance` (the size of our current position in each market) to enforce limits. For instance, if our rules say maximum 5000 USDC exposure per market, and one position's `balance * current_price` exceeds that, it should prevent adding more. The subgraph position can be an independent source of truth for how much exposure we have (in case our system lost track, we can always query the blockchain via subgraph to see actual balances). Also, position data like `avgBuyPrice` can help set stop-loss levels (e.g. if current price deviates X% from avg cost, consider cutting loss or taking profit). If we were running multiple strategies, position data ensures we consider total exposure (maybe two strategies both buy the same market – the position sum tells overall risk).

- **User Data (`User` entity):** Each user (address) accumulates stats like total volume, trades, markets participated, etc.

- *Signal Generation:* While our own strategy might not rely on user stats directly, user data can be mined for insights. For example, one could identify a list of **addresses of known arbitrageurs or market makers** by filtering `User` with high volume and many markets traded [40]. If the MBO sees a trade where `counterparty` is one of those addresses, it might interpret the trade differently (e.g. if a known market-making bot just sold a lot of Yes, maybe it's just rebalancing inventory and not informed; but if a rarely-seen user sold a lot, maybe it's informed selling). Such nuance might be beyond initial scope, but it's possible with user data. Also, if SLO wanted to incorporate "crowd sentiment", it might look at `newUsers` vs `activeUsers` in `DailyStats` to see if a flood of new users (possibly uninformed money) are coming in [29], which could precede a momentum overshoot that then mean-reverts.
- *Backtesting:* User data isn't needed for backtesting our strategy's performance (since we care about our trades), but it can be used to simulate different market conditions. For instance, if we want to test how the strategy performs in markets with different user profiles, we could use `marketsTraded` or `tradeCount` distributions. Additionally, by analyzing historical user behavior (like clustering users into whale vs small trader), we could parameterize some agent-based backtest. This is quite advanced; simpler backtests might not use user data at all.
- *PnL Attribution:* Not directly relevant except that we might label some profits as coming from "informed trader following" if we design a strategy around copying specific users. The subgraph could confirm whether trades we interacted with came from certain users by cross-referencing transaction hashes.

- *Risk Control:* Indirectly, user data can inform risk. If a market is dominated by one or two users (e.g. one user's volume is 60% of that market's volume, which we can find via `MarketParticipation` or top traders list), that's a risk factor – the market could be manipulated or prone to sudden moves if that user exits. A risk rule might be: avoid markets where a single user contributed > X% of volume. We can retrieve such info by querying

`marketParticipations(where: {market: ID})` sorted by volume. The subgraph's ability to find *whale traders* (example query in docs looks for users > $1M volume) [41] shows that identifying big players is straightforward. We could maintain a watchlist of whale addresses and, say, reduce our position if we're on the opposite side of a known whale's trades.

- **Daily and Global Stats (`DailyStats`, `GlobalStats`):** Aggregated statistics over time.

- *Signal Generation:* These high-level stats are more useful for *macro-level signals*. A strategy might not use them day-to-day, but they can help identify **regime changes**. For example, a sudden spike in `dailyStats.volume` or `activeUsers` might coincide with some event (e.g. a major news event brings lots of traders in) [42]. If our strategy knows that historically such spikes lead to increased volatility or momentum, it might adapt (this borders on ML feature usage). Also, long-horizon strategies might use a moving average of daily volume to gauge market momentum. GlobalStats (total markets, total users) is more for growth tracking; not likely used for trading signals, but perhaps to adjust expectations (if Polymarket activity is in a lull vs booming).
- *Backtesting:* DailyStats can be used to recreate broad market conditions for testing robustness. For instance, test the strategy during a period of low activity vs a period of high activity (the dailyStats will tell you those periods). If performance differs, you might refine the strategy.
- *PnL Attribution:* Not directly, but one could compare strategy PnL to overall market volume or volatility on those days to contextualize performance. For example, "we made a lot on days when platform volume was high, and lost on quiet days" might emerge from analysis.

- *Risk Control:* A risk framework might incorporate global context: if overall market volatility (maybe measured as sum of daily volume or large swings) is high, you might reduce leverage. Conversely, in very quiet times, maybe reduce activity because opportunities are fewer (or take smaller profits). Essentially, macro stats might toggle risk appetite. Additionally, if daily new users is surging abnormally, it could indicate an influx of potentially uninformed traders — the strategy might cautiously capitalize but also be wary of unusual behavior.

- **Token Registry / Lifecycle Events:** (`TokenRegistry`, `Split`, `Merge`, `Redemption` entities)

- *Signal Generation:* `Split` and `Merge` events directly signal arbitrage actions as discussed. An MBO-based arb bot might itself trigger these, but interestingly one could also use them as signals: if we observe *someone else* frequently performing splits/merges in a market, that implies the market had repeated mispricings or liquidity provisioning. If our strategy isn't pure arb, we might still glean that a certain market often goes out of balance (opportunity) or that someone is actively arbitraging it (competition). A high frequency of splits could indicate a market with volatile swings (each swing creating arb).
- *Backtesting:* To backtest parity strategies, we would simulate splits/merges. We can verify our simulation against actual split events in history to see if we would have acted similarly. If not implementing parity strategy, splits/merges can be mostly ignored in backtest except as a sanity check that the market self-corrected when mispriced.
- *PnL Attribution:* If our strategy does arbitrage, each `Split` or `Merge` we execute would essentially finalize some PnL (especially merge yields USDC profit). We'd track that. The subgraph's recording of `Redemption` is more for final payout after resolution. If our strategy sometimes holds through outcome resolution, then a `Redemption` event (for our address) would be the final PnL realization (like winning a bet). We can incorporate that in backtesting by using `winningOutcome` to assign PnL rather than explicitly simulating redemption events.

- *Risk Control:* Knowing that splitting requires USDC on hand and merging yields USDC, the strategy must manage capital. If we see too many split events queued (maybe our bot tries to do multiple markets), risk might ensure we don't run out of USDC for redemptions. Also, after performing a split, we hold two opposing positions; risk module might set aside that capital until the merge is done (to ensure we can cover payout if needed). These are more strategy mechanics than subgraph data, but subgraph records can help in debugging if something went wrong (e.g. a split without a corresponding merge means we left a position open accidentally).

To illustrate mapping, consider an **example workflow:** The MBO detects via `Trade` data that in Market X, Yes price jumped from 0.50 to 0.70 on a single trade of 10,000 tokens at time T. It references `User` data and finds the trader is a new user with low trade count (not a known whale), which might suggest an overreaction (signal to fade). SLO checks Gamma metadata: Market X is about an upcoming election two weeks away, and SLO finds no news in mainstream sources about that election at time T. Based on these, the strategy engine decides to short Yes in Market X (fade the spike). It uses the subgraph's last known prices (0.70 Yes, implying ~0.30 No) to decide to either sell Yes or buy No. Since it has no Yes to sell, it does a `Split` of collateral (minting Yes and No) and immediately sells Yes outcome (perhaps via the execution module placing a limit sell around 0.68 to ensure quick fill). In backtest, we simulate this: we deduct one `Split` worth of USDC, record that we have a short Yes position (long No essentially), and then use subsequent `Trade` data to see at what price the next trades occur to assume our order fills. The price indeed mean reverts to 0.60 within minutes; our order filled at 0.65 average. We then decide to close by merging: we buy back Yes at 0.60 (or equivalently, use the No we held to redeem when prices normalized). The subgraph's data on trades confirms Yes traded down to 0.60, and we record profit ~0.05 ($0.05 * number of tokens minus fees). When actually running live, these actions would be done via Polymarket's CLOB (placing an order, etc.), and the subgraph would later show our `Trade` and `Split/Merge` on-chain for verification.

Through this mapping, we see that **each component of the strategy pipeline is supported by specific subgraph data fields**: from identifying signals (trades, volume, outcomes, user behavior) to executing and then verifying outcomes (positions, PnL, resolution events). The Polymarket subgraph acts as the **data backbone** for both development and monitoring of an automated trading strategy on the platform.

Modular System Architecture for Automated Trading

Finally, we outline a proposed **modular architecture** that ties everything together – from data ingestion to live trading execution – leveraging the Polymarket subgraph and complementary services. This design ensures clarity, maintainability, and the ability to backtest and deploy strategies seamlessly.

1. Data Ingestion Layer: This layer is responsible for collecting and updating all required data: - **Subgraph Data Fetcher:** A module (or service) that queries the Polymarket subgraph for historical data and keeps an updated state of recent on-chain events. For example, it could periodically fetch the latest trades (or subscribe if theGraph supports subscriptions) for markets of interest. It also can retrieve static info like the list of all markets and their on-chain fields (trade counts, etc.) as baseline. This forms the historical database used for backtesting and analysis. - **Gamma API Client:** This component queries Polymarket's off-chain API for market metadata and event info. It can fetch all markets with their titles, descriptions, categories, and resolution times [17]. It should also handle refreshing this data (in case new markets are added, or event details updated). The Gamma client essentially enriches the raw subgraph market list with semantic info. - **Order Book Feed (CLOB Interface):** For real-time operations, integrate the Polymarket CLOB WebSocket or client. This provides live updates on order placements, cancellations, and trades on the order book. It's crucial for obtaining the current best bid/ask and depth in each market. Polymarket's provided clients (Python or TypeScript) can be used [36]. If an order book is

not available for a particular market (e.g. legacy AMM markets), the strategy can fall back on querying subgraph for the last trade and maybe calculating an approximate AMM price curve. - **External Data Connectors:** Depending on strategy, include connectors for external sources: e.g. a news API poller (or WebSocket using services like Twitter's API or RSS feeds), a database of historical news for backtesting, any specific data like a polling numbers feed or sports scores feed. If using an ML model hosted externally, a connector to that service (e.g. an API that returns predicted probabilities).

These ingestion modules might populate a central in-memory store or a set of topic channels. For instance, a pub-sub could be used: "market_X_trade_updates" topic gets new trade events, "news_events" topic gets semantic events, etc. This way, downstream modules can subscribe to the data they need.

2. Semantic Logic Oracle (SLO) Module: Implement the SLO as a service that subscribes to relevant data: - It listens to **market metadata** updates (so it knows how to interpret each market) and to **external events/news**. - It maintains an internal knowledge base: e.g. a dictionary of upcoming event dates (so it knows how far away each market's resolution is), groups of markets by event or category, and possibly a knowledge graph linking related events (if market A is about "Candidate X wins" and market B is "Candidate Y wins", SLO might mark them mutually exclusive if same election). - For real-time, when a news event is detected (say a news API flags breaking news on "Candidate X"), the SLO might score its relevance to each market (using keywords or an NLP model). If relevant, it generates a semantic signal: e.g. "Bullish news for Candidate X's market" or "Event date changed". - The SLO can also output **expected value signals**: using any predictive models, it can produce a probability estimate for each outcome given all information. For example, an ML model input might be current price + various features, and output a "fair" price. SLO could publish "Market X fair_price_estimate = 0.65" which strategy logic can compare to actual price. - Technically, SLO might use libraries or AI models (Polymarket Agents mention vector DB and LLMs for news [33]). In our architecture, this could be a sub-module within SLO where news articles are embedded and matched with markets.

3. Market Behavior Oracle (MBO) Module: Implement MBO as a service focusing on quantitative signals: - It subscribes to **trade feeds** (from subgraph or direct WebSocket) and **order book updates**. It computes indicators in real-time. This could be done using a sliding window per market that updates with each new trade. - The MBO can use a small internal state for each market: e.g. last N trade prices, moving average values, current order book snapshot, last 10-second volume, etc. It updates these on the fly. - On certain conditions, it emits events/signals. E.g., if a trade arrives that is 10% above the previous price and volume > threshold, it emits a "spike_detected(market_id, price_jump=+10%, volume=X)" signal. Or continuously it can emit "parity_gap(market_id) = 5%" if it calculates Yes+No = 1.05. - The MBO could also label incoming trades by the type of counterparty (if it has a cache of user types). It might attach metadata like "trade by likely market maker" vs "trade by new user". - The output of MBO is fed to strategy logic. Some signals might directly trigger actions (like an arbitrage signal might directly tell the strategy to execute, as time is of the essence). Other signals are more informational (like momentum up, etc.) to be used in combination with SLO or other conditions.

4. Strategy Logic & Signal Processor: This is the core decision engine that merges SLO and MBO outputs to decide when and what to trade. One can structure it as multiple *strategy modules*: - **Arbitrage Strategy Module:** Listens for parity signals from MBO (and any SLO context about those markets) and if conditions are met, decides to execute split/merge and trades. For example: if MBO says Yes+No = 1.10 (10% arbitrage) and SLO doesn't indicate any special situation (like imminent resolution which could make liquidity low), then execute arbitrage. The strategy module will formulate the specific orders needed (e.g. how much to trade given capital). - **Reversion Strategy Module (Microstructure):** Listens for spike_detected or range-bound signals. If a spike up is detected and SLO confirms no news, it issues a short order (could be a market sell or a passive sell at slightly below current price). If a range signal

(price at high end of recent range without news) comes, similarly short; if at low end, buy. This module also sets stop-loss/take-profit logic (maybe place an opposite order to exit at a certain favorable price or stop out if price goes further against). - **Momentum/Trend Strategy Module:** Uses both SLO and MBO. For instance, if SLO's fair price is higher than current and MBO sees upward momentum, it buys to ride the trend until maybe equilibrium. Or if many new users are entering (SLO sees high newUsers metric) and MBO sees price moving up steadily, it could join the rally (anticipating a momentum overshoot and then plan to exit later). - **Portfolio Module:** If multiple strategies are running, this part oversees the combined position to avoid conflicts and optimize capital usage. It ensures, for example, that the arbitrage strategy and trend strategy aren't unknowingly taking opposite sides in the same market (if they are, it could just let them net out or prefer one signal over another).

The strategy logic should be configurable with parameters (which can be optimized in backtesting). It produces *trade intents* – basically instructions like "place order to sell 500 Yes at >=0.68 on market X" or "buy 1000 No at market price now". Importantly, this layer also implements **sizing** and basic risk rules per trade (no trade bigger than X, etc. as configured by risk module).

5. Risk Management Module: This runs in parallel and intersects with strategy logic: - It continuously monitors current positions (from our internal record or by querying subgraph for our address's positions). - It calculates metrics like current leverage, exposure per category, unrealized PnL, etc. - It receives signals from SLO/MBO that imply risk (e.g. "imminent event" from SLO or "volatility spike" from MBO). - Based on predefined risk rules, it can override or modify strategy outputs. For instance, it can veto a new trade that would exceed limits ("don't short more Yes, we already have a large short here"), or instruct the strategy to reduce exposure ("event in 1 day, cut positions by 50% now"). - It also handles safe shutdown logic: if overall PnL drops beyond a threshold (max drawdown), it might liquidate all positions and stop trading – this could be triggered by tracking realized PnL and current market prices of held positions. - The risk module likely has some persistent storage of parameters (like max allowed exposure, etc.) and logs of breaches.

6. Backtesting & Simulation Environment: All the above components should be usable in a simulated mode. The backtest engine can: - Load historical data for a period (using the Data Ingestion layer in a historical mode, e.g. reading from database or CSV exported from subgraph). - Replay the data through the SLO and MBO: e.g., feed trades in chronological order to MBO, feed historical news events to SLO at appropriate times. - The strategy logic will generate simulated trade decisions, which the backtester will execute against historical prices. If order book depth is known or assumed, the backtester can simulate partial fills, slippage, etc. Otherwise, a simple approach is to assume our trades execute at the next available trade price. - Record the PnL and metrics. After simulation, output performance stats, and possibly allow parameter tuning. This environment helps validate strategies and calibrate thresholds (e.g. find optimal z-score threshold for spikes, or optimal arbitrage trigger level considering fees). - The modular design means we could plug different strategy modules on/off in backtest to see which adds value.

7. Execution & Interface Module: In live trading, this is critical: - It takes trade intents from strategy logic and converts them to actual API calls or on-chain transactions. For Polymarket's CLOB, that means constructing orders (the Polymarket order utils can help sign orders [36]). For immediate execution (taker trades), it might call a "swap" or trade endpoint (if Polymarket offers a direct trade API). - It then listens for confirmation (order filled, etc.). If partially filled or not filled, it may inform strategy logic to adjust (e.g. if a limit order sits unfilled for too long, maybe cancel or adjust price). - It must handle errors (transaction fails, network issues) gracefully, possibly by retrying or alerting the system. - This module should also update the internal position tracking upon execution (and ideally cross-verify with subgraph after a short delay to ensure everything matches on-chain). - Security: it manages private keys securely since it will sign transactions. Ideally isolated or using a secure signing service.

8. Monitoring & Reporting: (Optional but important) Another component can be a dashboard or logs that monitor the system's state: - It can use subgraph queries to produce real-time reports of PnL, or at least subscribe to our address's trades to confirm what's happening. - It can alert if, say, a strategy hasn't traded in a long time (maybe a feed died), or if an error occurred in execution. - Periodic summary of performance per market, etc., using subgraph data for accuracy.

The architecture described above ensures that **each concern is separated**: data collection vs signal generation vs decision making vs execution. This modularity not only helps in clarity and maintenance but also means each part can be improved independently (for instance, swapping in a better MBO indicator set or an updated ML model in SLO).

Critically, using the **Polymarket subgraph as the base layer** means our historical data is reliable and can be used to validate the entire pipeline offline. Once validated, the **live system** leverages the same logic but with real-time data sources (Gamma, CLOB) and executes on the actual markets. By mapping subgraph fields to our components and filling in gaps with Gamma API and other enrichments, we fully utilize Polymarket's infrastructure to develop sophisticated automated trading strategies.

Sources: The analysis above references Polymarket's subgraph schema and data capabilities [43] [8] [11], as well as insights from a microstructure arbitrage specification [44] [6]. We also leveraged Polymarket's developer resources on Gamma API and CLOB integration [31] to outline the practical implementation of the trading pipeline. This comprehensive approach allows us to harness on-chain historical data and real-time feeds to their fullest extent in building and testing automated trading strategies on Polymarket.

---

[1] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [22] [23] [24] [25] [26] [27] [28] [29] [30] [38] [39] [40] [41] [42] [43] AVAILABLE_DATA.md
file://file_00000000afcc72069faa1a1855ab7f4e

[2] [3] [4] [5] [6] [34] [35] [44] micro-arbitrage.pdf
file://file_00000000d930720681a5bb966ab8c287

[21] Gamma Structure - Polymarket Documentation
https://docs.polymarket.com/developers/gamma-markets-api/gamma-structure

[31] [32] [33] [36] [37] GitHub - Polymarket/agents: Trade autonomously on Polymarket using AI Agents
https://github.com/Polymarket/agents