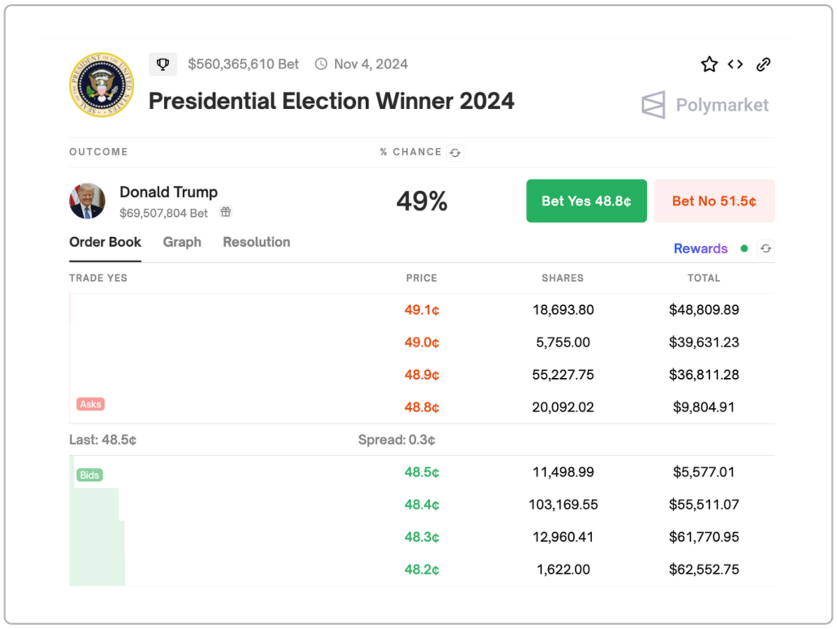
ChatGPT

Building a Polymarket-Style Order Book Prediction Market (Back-End Design)

Background: AMM vs Order Book Models in Polymarket

Polymarket's Original AMM (LMSR): In its early version, Polymarket used an Automated Market Maker based on the **Logarithmic Market Scoring Rule (LMSR)** to price trades. This meant the smart contract itself would always quote a price for yes/no shares using a formula, so a user could trade at any time even without a counterparty. For example, **the first trader in a new market could immediately buy shares** – the LMSR algorithm would determine the price and sell to them, providing what's called "perpetual liquidity" [1]. In other words, **you were never stuck waiting for another user** to take the opposite side of your bet; the AMM would always fill the order (with price slippage determined by the cost function) [1]. This design ensured the market always had liquidity, similar to Uniswap-style constant product AMMs in DeFi.

Shift to Order Book Exchange: Today, Polymarket has **transitioned to a central limit order book model**, matching buyers and sellers of shares directly like a traditional exchange. In the current system, there is **no "house" taking the other side** of trades – every trade is purely *peer-to-peer*, with Polymarket acting only as the neutral matching engine [2]. Users post **limit orders** to buy or sell "YES" or "NO" outcome shares at specified prices, and trades occur when orders from opposite sides are compatible. The platform's interface now shows a familiar order book with buy orders (bids) and sell orders (asks) at various price levels [3], and users can also place **market orders** that execute against the best available prices. The screenshot below illustrates Polymarket's order book interface, with green bids and red asks for a market's YES shares:



$560,365,610 Bet · Nov 4, 2024

Presidential Election Winner 2024 — Polymarket

OUTCOME	% CHANCE		
Donald Trump $69,507,804 Bet	49%	Bet Yes 48.8¢	Bet No 51.5¢

Order Book · Graph · Resolution — Rewards

TRADE YES	PRICE	SHARES	TOTAL
	49.1¢	18,693.80	$48,809.89
	49.0¢	5,755.00	$39,631.23
	48.9¢	55,227.75	$36,811.28
Asks	48.8¢	20,092.02	$9,804.91
Last: 48.5¢	Spread: 0.3¢		
Bids	48.5¢	11,498.99	$5,577.01
	48.4¢	103,169.55	$55,511.07
	48.3¢	12,960.41	$61,770.95
	48.2¢	1,622.00	$62,552.75

Polymarket's web interface showing the order book for a YES outcome share (bids in green, asks in red). The order book lists every open buy or sell order and the prices and sizes available [3].

Advantages of the Order Book Model: Moving to an order book allows more efficient pricing and tighter spreads in liquid markets, since users (including professional market makers) can provide liquidity on both sides at their chosen odds. Unlike the AMM where prices move along a curve with each trade, an order book exchange lets large trades execute at stable prices if sufficient orders are in the book (no automatic slippage beyond the spread). It also means prices are set entirely by supply and demand of users' opinions. Polymarket's order book has attracted dedicated liquidity providers who place orders on both YES and NO sides to earn the bid-ask spread and rewards [4] [5], improving market quality. The downside is that if no one is willing to take the opposite side of a trade, an order might not fill immediately – a sharp contrast to the guaranteed fills under the old AMM. In practice, Polymarket mitigates this by having enough users and incentivizing liquidity, so most popular markets have a healthy order book.

Off-Chain vs On-Chain Implementation

Polymarket calls its new architecture "hybrid-decentralized" – the **order matching happens off-chain on a backend server**, while final settlement of trades and custody of funds occur on-chain via smart contracts [6]. In our case, you've indicated **"no chain,"** meaning you plan to run the entire system off-chain. This simplifies development (no need to write or deploy Solidity contracts) and allows faster, fee-free transactions in the simulation/prototype stage. The trade-off is that the backend must be **trusted** to custody user funds and enforce rules correctly (whereas an on-chain contract is trustless).

For an **off-chain backend**, you will maintain a ledger of user balances (e.g. in a database) and simply update balances and positions as trades occur. Users would deposit/withdraw in a centralized manner (or you can simulate deposits). Polymarket's logic can be reproduced in code: you'll essentially create a centralized exchange engine specialized for binary outcome shares. This engine can run as part of your server application (for example, a Django app with Celery or in-memory matching process). When a user places an order, it's added to an internal order book data structure or database table. The engine then attempts to match it with existing orders according to the rules below. Because everything is off-chain, **the backend itself will perform the functions that Polymarket's smart contracts do**, such as minting/ burning shares and transferring funds between accounts.

On-chain vs Off-chain considerations: One key difference is how shares are represented. On Polymarket, YES and NO shares are tokens (previously using Ethereum's Conditional Tokens framework), and each pair of YES+NO is backed by 1 USDC on-chain [7]. In your off-chain model, you don't need actual tokens – you can represent a user's holdings as entries in a **Position** record (e.g. user X holds Y YES shares and Z NO shares in market M) and enforce that one YES + one NO corresponds to $1 collateral internally. The "collateral" in this case will be tracked in your database (for example, you might have a field for how much cash is set aside for each market's open interest). When trades happen, you will debit and credit user balances and update their share holdings accordingly, ensuring that at all times the system could pay out the winning shares from the collected funds. Essentially, your backend acts like the Polymarket smart contract – it holds the collateral for all outstanding shares and releases it upon resolution to the winners.

To summarize, an off-chain backend design will involve: user account balances (in USD or stablecoin units), order management (storing open orders and matching them), position tracking (how many YES/NO shares each user owns per market), and a transaction ledger to record debits/credits (for transparency and debugging). You'll also need to implement settlement logic when the event outcome is determined (discussed later). Now let's delve into the **order matching logic** which is the core of the back-end.

Order Book Matching Engine Design

At its heart, the matching engine will operate similarly to any exchange, with price-time priority matching of orders, but with special logic to handle the two-outcome nature of markets. Each **order** should be stored with: the market it belongs to, the side (buy or sell), the outcome (YES or NO share), the price (in cents or decimal up to $1), and the quantity (number of shares). Users can place **limit orders** (specifying a price) or **market orders** (execute immediately at best price). We maintain an order book for each market – essentially two books internally (one for YES, one for NO) or a unified structure – and match compatible orders. Here are the key matching scenarios your back-end must handle:

- **Direct User Trade (Order Matching on the Same Outcome):** This is the standard case: a buy and a sell on the **same outcome** meet at a common price. For example, Alice places a buy order for 100 YES shares at $0.48, and Bob has a sell order for 100 YES at $0.45. Alice's bid is higher than Bob's ask, so a trade can execute at $0.45 (or $0.48, depending on how you choose to fill – typically it would match at the price of the older order or the ask price). In this scenario, **shares are exchanged directly between users**: Bob gives up 100 YES shares and Alice pays $45 for them (100 * $0.45). The engine will transfer $45 from Alice's balance to Bob's balance, and transfer 100 YES shares from Bob's position to Alice's position [8]. No new shares are created or destroyed in a direct match – it's a simple swap of assets between two users at an agreed price. This is analogous to a stock trade on an exchange order book. After the trade, update the **last traded price** (market price) for YES (and correspondingly for NO as 1 – price). Both orders (Alice's and Bob's) would be marked filled (or partially filled if quantities differ).

- **Minting (Creating New Shares via Opposite Buy Orders):** If a buy order for YES meets a buy order for NO *at complementary prices* (prices that sum to $1), the system can match them by **minting new shares** backed by collateral. This happens when two users are effectively willing to take opposite sides of a bet, but *neither* is currently holding the shares – they both want to buy shares (one wants YES, the other wants NO). **Polymarket's rule is that a YES buy and a NO buy can trade if their prices add up to $1.00** [9] [7]. For example, say Alice places a limit order to **buy 100 YES at $0.35**, and Bob places a limit order to **buy 100 NO at $0.65**. These orders are on opposite outcomes, and $0.35 + $0.65 = $1.00 exactly, meaning their odds are complementary. The engine will detect this and execute a **mint trade** for the overlapping quantity. In this case, the overlap is 100 shares (both wanted 100). The system will **mint 100 new YES shares and 100 new NO shares** as a pair. Alice receives the 100 YES shares she wanted, and Bob receives 100 NO shares [10]. Each pays the price they bid: Alice pays $35 (100 * $0.35) and Bob pays $65 (100 * $0.65). Instead of one user's money going to the other, their funds go into the **market's collateral pool** (held by the platform) and become the backing for these 100 newly created share pairs [11]. No existing user was on the opposite side – **the platform's contract (or backend) essentially acted as the intermediate maker**, taking $100 total from the two traders and issuing them the shares. This is why it's called *minting*: new YES/NO shares are created out of collateral whenever two opposing buy orders agree on odds [7]. If the two orders were for different quantities, you only mint up to the smaller quantity – e.g. if Alice wanted 100 YES but Bob only wanted 50 NO at $0.65, you'd mint 50 shares for each (using $17.50 from Alice and $32.50 from Bob as in the example below), then Alice's order would remain partially unfilled for the remaining 50 YES shares [10] [11]. The **price** at which this trade occurs is inherently defined by the two orders' prices (here effectively $0.35 for YES and $0.65 for NO). After minting, the market now has 100 YES and 100 NO in circulation backed by $100 collateral held by the platform. Both Alice and Bob have open positions (Alice long YES shares, Bob long NO shares), and the **market's last traded price** could be recorded as 0.35/0.65 (often the midpoint or one side's price is shown as the "market price").

- **Merging (Burning Shares via Opposite Sell Orders):** This is the inverse of minting – it occurs when two users want to **sell opposite outcome shares** at complementary prices, allowing the system to **remove those shares from circulation** (burn them) and release collateral. For example, suppose Alice holds YES shares and places a **sell order for 100 YES at $0.35**, and Bob holds NO shares and places a **sell order for 100 NO at $0.65**. If these orders meet (their prices $0.35 + $0.65 = $1.00), the engine can match them by effectively undoing a share pair. It will take 100 YES from Alice and 100 NO from Bob and **burn** those 200 shares (removing them from the market supply) [12]. In exchange, the platform pays out the collateral to both: Alice gets $35 (for her 100 YES at 35c each) and Bob gets $65 (for his 100 NO at 65c each) from the pool [12]. Again, **no money is exchanged directly between Alice and Bob** – the payout comes from the collateral that was backing those shares, via the platform's treasury [13]. After this **redemption trade**, 100 fewer YES and NO are in circulation, and $100 less is held as liability, because those shares have been settled early between the two users. If one had more shares to sell than the other, it would only merge the smaller quantity (similar to partial fills). Essentially, merging allows traders to exit positions when someone else also wants to exit on the opposite side, by **cashing out a complete set (YES+NO) for $1.00**. This is the same concept as a user individually redeeming a YES–NO pair they hold, except here two different users each held one side of the pair and the system facilitated them cashing out together.

These three scenarios cover all possible ways trades occur in the order book model [14] [12]. The engine should always check for matches in this order of priority whenever a new order comes in or an existing order is updated: first match *directly opposite orders on the same outcome*, and if those are not available or not fully filling the order, then check for *cross-outcome matches (mint or merge)* at complementary prices. Any portion of an order that cannot be matched immediately will remain in the order book as an open order (available for future matching).

Price-Time Priority: As with any order book, if multiple orders are eligible to match, you fill against the ones with the best price first, and if prices are equal, the oldest order first. For example, if a user submits a buy YES order at $0.50 and there are several sell YES orders at $0.50, you'd match the one that was posted earliest. Similarly, if multiple opposite-outcome orders exist at complementary prices, match in order of who posted first. Maintaining sorted lists (e.g. a min-heap for asks and max-heap for bids, or sorted DB queries) can help retrieve the best orders quickly.

Market Orders: A market order (say, "buy 200 YES at market") can be handled by iteratively matching it against the best available opposite orders. For a market buy YES, you would fill existing sell YES orders from the lowest ask upward (direct trades). If the market order still isn't filled after consuming all sells (perhaps the order book was empty on that side), you could then look for any *buy NO* orders to trigger minting (because effectively, if someone is willing to buy NO at some price, the market order buy YES can mint shares with them). This means a market order might **sweep the book**: first taking any resting orders on the same side, then, if still unfilled, pairing with opposite-side orders to mint shares until the order size is filled or no more counterpart orders exist. Similarly, a market sell might match against opposite buys, then if needed against opposite sells for merging. In practice, market orders in a system like this will rarely need to invoke mint/merge logic unless the book is completely one-sided, but your backend should handle it for completeness. Always ensure a market order doesn't execute beyond the prices that make sense (you might impose a limit that it won't cross $1 boundary or cause obvious arbitrage losses – e.g., a market order should **never fill against an opposite-outcome order unless it makes a full $1**, to avoid giving away free arbitrage profit).

Unified Order Book View: Internally, you can choose to keep two separate order lists for YES and NO, but it's critical to maintain the **mirror relationship** between them. In Polymarket's design, every order on one outcome can be viewed as an equivalent order on the other outcome at the complementary price [15]. For example, a limit **bid of $0.40 for YES** is effectively also an offer to **sell NO at $0.60** (because if someone else wanted to take the opposite side, $0.40 + $0.60 = $1). Conversely, an **ask of $0.25 for YES** implicitly represents a **bid of $0.75 for NO** (the seller is saying they'll give up a YES share for $0.25, which is the same as wanting $0.75 to not have the event happen) [15]. Polymarket's interface reflects this by showing the "other side" of every order, which keeps the market consistent and helps users see all opportunities. In your backend, you don't necessarily need to create two physical orders for each user order, but you **do need to ensure the matching logic accounts for these equivalences**. One approach is: store all orders in one structure keyed by outcome and side, but when searching for matches also consider the complementary outcome. For instance, when a new buy YES arrives, after checking sell YES orders, look for *buy NO* orders (which are candidates for minting). In effect, you treat a buy NO as if it could match with a buy YES by minting.

The unified nature also means you should enforce a **consistency rule**: *no combination of resting orders should create an obvious arbitrage*. If you ever had, say, a bid for YES at $0.52 and a bid for NO at $0.52 simultaneously, that sums to $1.04 – an arbitrageur could mint shares for $1 and sell both for $1.04, risk-free profit [16]. In a competitive market, someone would immediately do this (placing the needed orders to complete the $1 trade) and capture the profit, which effectively pushes the prices back into balance. Your matching engine can proactively handle this by always checking: if **best YES bid + best NO bid ≥ $1**, you can automatically execute a mint-sell operation to remove the arbitrage (or equivalently, match those bids via the protocol). Similarly if **best YES ask + best NO ask ≤ $1**, that's an arbitrage to buy both and redeem for $1 (though this scenario is less common) [16]. You may choose to let traders find these arbitrage opportunities, or implement an automatic arbitrager in your system to keep prices aligned. Polymarket's design relies on the fact that **rational traders will keep the sum of odds ≈ $1** by taking advantage of any deviation [17]. As long as your engine allows minting and merging when the $1 rule is exactly met, the market forces should handle slight deviations by quickly filling those gaps.

Minting and Redeeming Complete YES/NO Sets

You specifically asked about **minting/redeeming complete sets of shares**, and indeed this is a crucial feature. In Polymarket, *any pair of one YES + one NO share is always worth $1.00* (ignoring fees) because one of those shares will eventually pay $1 and the other $0 [18]. This means if a user holds an equal number of YES and NO shares in a market, they have a neutral position equivalent to cash. Your system should support the creation and destruction of these pairs seamlessly:

- **Minting Complete Sets:** As described in the matching logic, whenever two opposite buy orders meet, the system mints a complete set (one YES and one NO) for each unit. You can also allow **users to mint shares proactively** without waiting for a counterparty, if they want to provide liquidity or take a balanced position. For example, you might let a user click "Mint shares" on a market, specify an amount (say $100), and then deduct $100 from their balance to create 100 YES and 100 NO shares in their portfolio. This is essentially the user acting as both sides of the above scenario (paying $1 for each YES+NO pair). Polymarket's documentation notes that *shares are created when opposing sides come to an agreement on odds such that the sum of what each side pays equals $1.00* [7]. In our off-chain context, if the user alone wants to mint, they *are* both sides – they deposit $1.00 of collateral per share pair. Typically, a user wouldn't mint just to hold both outcomes (since that's the same as holding cash), but they might mint in order to **sell one side into the market**. This is a strategy for liquidity providers: e.g., mint 100 sets at $1, then post a sell order for 100 YES at the current market price and effectively you'll end up holding 100 NO and some cash, which is a way to short YES or just provide liquidity. From a backend perspective, allowing direct minting is straightforward – it's equivalent to a user depositing collateral and receiving one YES and one NO share (update their position and move funds into the collateral pool).

- **Redeeming (Burning) Complete Sets:** Likewise, if a user holds equal quantities of YES and NO shares, they should be able to **redeem those for $1 each pair** at will. Polymarket's system implicitly does this via the merging of two users' orders or via arbitrage, but you can provide a direct "Redeem" function for convenience. Upon redemption, the user gives up one YES and one NO share (reduce their positions), and your backend releases $1.00 of collateral back to their balance (credit their account). This is essentially the same transaction that happens when two different users do a merge trade, except with one user providing both sides. It's risk-free and doesn't affect market prices. In fact, if the market prices ever diverge (say YES at 0.6, NO at 0.4, which is perfectly aligned) a user holding both could just as well hold $1 – redemption doesn't profit or loss, it's just converting their "share bundle" back to cash. By supporting this, you ensure that **no user is forced to keep a fully hedged position** – they can always get their money out if they have no net exposure. (At market resolution, any such combined position would pay out $1 anyway, but before resolution some users might prefer to free up their capital.)

Implementing these features on the back-end means tracking the **collateral pool** for each market. Every time you mint X shares (whether via matching two orders or a user-initiated mint), you should increase the market's collateral by $X (and correspondingly decrease users' cash balances by the amounts they paid). Every time shares are burned (via merge or redemption), decrease the collateral by the appropriate amount and credit it to the users. In a centralized database, you might not have an actual "pool account" per market, but conceptually you could maintain a field like `Market.collateral_balance` or derive it from the total shares in circulation. Polymarket's own contract ensures at creation that **each pair of "YES" + "NO" shares is fully backed by $1.00 USDC** [7], and you should do the same – never create shares without taking in equal money, and never release money without canceling shares.

A helpful invariant to keep in mind is: *Total collateral held = total YES shares in circulation = total NO shares in circulation (for binary markets).* Initially both 0; when you mint, you increase both YES and NO supply together with collateral; when you burn, you decrease both supplies together and collateral. This ensures that if the event resolves to YES, there's exactly enough money to pay $1 for each YES share (since that equals the collateral and was equal to the number of YES shares). In code, you'll update user balances and positions accordingly and possibly log a **transaction** for transparency. For example, your backend might log a "TRADE_BUY" of –$17.50 for Alice and a "TRADE_BUY" of –$32.50 for Bob for the mint scenario, along with a +$17.50 "POOL" or similar record (or just ensure the sum of user balances decreased by $50, which is now held for that market). On merge, you'd log "TRADE_SELL" +$17.50 for Alice and +$32.50 for Bob, and reduce the pool by $50. Maintaining this audit trail (as you have with a Transaction model) is wise for debugging and integrity.

Settlement and Payout Logic

Designing the back-end also requires handling **market resolution** – when the real-world event outcome is known (yes or no). At that point, trading would be halted and all shares of the winning outcome can be redeemed for $1.00, while shares of the losing outcome expire worthless [7]. Since every share was backed by collateral, you can pay out winners seamlessly:

- If the event resolves **YES (happened)**: Every YES share is now worth $1. The backend should iterate through all user positions in that market and credit $1 for each YES share they hold (and set their YES share holding to zero). The NO shares become worthless – you would set the quantity of NO shares to zero with no payout. The total funds paid out will exactly equal the collateral collected for those shares. In practice, you might simply let users manually redeem, but it's easier to just automatically credit them since outcome is known. For example, if Alice had 100 YES shares, credit her $100 (and record a `SETTLEMENT_WIN` transaction) [19]. If she had 50 NO shares, those get removed with no payout (`SETTLEMENT_LOSS` transaction for record) [19].

- If the event resolves **NO (did not happen)**: Similarly, every NO share pays $1. Credit users who hold NO shares, and remove YES shares as worthless.

Because your system already maintained that one YES + one NO = $1 collateral, you won't have any imbalance – the money to pay the winners is exactly the money that was paid in by traders. The platform doesn't take a profit or loss on the outcomes (unless you charge a fee separately). **Polymarket's documentation confirms** that the shares of the correct outcome "are paid out $1.00 USDC each upon market resolution" [20], which is the principle you'll follow. After settlement, you might reset the market or mark it as resolved/closed. Users can then withdraw their winnings or use them in other markets.

One more edge-case: **Canceling markets.** If a market is invalid or canceled (no outcome), typically both YES and NO shares are refunded at $0.50 each (or whatever the rules dictate). That means if you had collateral, you'd return it evenly to holders of both sides. This is something to plan for, though hopefully rare.

Conclusion and Additional Considerations

Designing your back-end to **"work exactly the same"** as Polymarket's current system involves replicating the above mechanisms. You'll focus on: maintaining the central order book, enforcing the $1 complement rule, and handling the minting/burning of shares with proper collateral accounting. It seems you are using Django – your models (for User balances, Orders, Trades, Positions, Transactions, etc.) already set a solid foundation. The key is implementing the matching engine logic (perhaps in a Django view or consumer that handles order placement). This matching engine should:

- **Reserve funds** when orders are placed (e.g. move a user's money into a "reserved" state so they can't spend it twice) [21], and release or settle those funds when trades execute or orders cancel.
- **Match orders** as per scenarios above, possibly in a loop until an incoming order is fully matched or no more matches. This can be done synchronously when an order is placed, or by a background

process that continuously processes an order queue. Given that you're focusing on back-end, ensure this process is atomic enough to avoid race conditions (you might use DB transactions or locks such that two trades don't mint the same shares twice, etc.).

- **Update positions and balances** transactionally. When a trade happens, update both the buyer's and seller's position counts for YES/NO shares and adjust their balances. Each trade will have two sides to log. For mint/merge, you might create two Trade records (one for each side) or a single record that denotes the contract type traded; but as long as you correctly adjust positions, it's fine.
- **Maintain derived fields** like last traded price, best bid/ask for YES and NO (you already have fields like `Market.last_yes_price`, `best_yes_bid`, etc. which should be updated as orders change) [22] [23]. This will help your front-end display the current market odds easily.
- **Fees and rewards:** Polymarket charges a small fee on trades and has liquidity/holding reward programs. Initially, you can ignore these or implement a simple fee (e.g. subtract 1% of the trade value to a fee account). Since you didn't specifically ask, it's optional, but keep in mind fees must be handled by reducing payouts slightly and aren't part of the core matching logic. If implemented, it means the sum of YES+NO might yield slightly less than $1 to traders because the platform takes the remainder. Polymarket currently has a 2% fee on winnings (implied in prices) and uses reward programs to incentivize liquidity [5] [24] – those are higher-level features you can consider later.

By focusing on the **back-end order matching and share accounting**, you create the foundation on which a front-end can show the live order book and allow users to trade. The **central insight** is that a binary prediction market's order book is really two intertwined order books kept consistent by the invariant that "YES price + NO price = $1". Your backend will uphold that by allowing the minting and burning of shares whenever needed. As a result, users on your platform will experience the same behavior as Polymarket: they can always either find a counterparty or, if none exists, create one by effectively betting against the platform's pool (which is just the aggregate of other users' collateral).

Finally, make sure to test these scenarios thoroughly. Simulate a new market: user A submits a buy YES, no immediate fill, order sits. User B submits a buy NO at complementary price, see that they both get filled and positions updated (mint). Then maybe user A tries to sell some YES, while user C buys YES – that should match directly. Or user A and B both decide to sell their holdings – that should merge and return their money. Verify the collateral accounting at each step (total money in system should remain constant except for fees). By matching Polymarket's model step-by-step, you'll ensure your project works exactly the same way. Good luck with building the back-end – it's a challenging but very rewarding project!

Sources: The above design is informed by Polymarket's documentation and third-party analyses of their system, including ChainCatcher's review of Polymarket's shift from LMSR to order books [1], Polymarket's own help docs [25] [2], and a RockNBlock technical explainer on how Polymarket's binary options order book operates [15] [9]. These sources provide further details on the reasoning and mechanics behind each aspect of the implementation.

---

[1] [16] From AMM to Order Book: Interpreting the Shift in Polymarket's Pricing Mechanism and the Possibility of Integration with DEX - ChainCatcher
https://www.chaincatcher.com/en/article/2196337

[2] [7] [20] [25] What is Polymarket? - Polymarket Documentation
https://docs.polymarket.com/polymarket-learn/get-started/what-is-polymarket

3 Using the Order Book - Polymarket Documentation
https://docs.polymarket.com/polymarket-learn/trading/using-the-orderbook

4 5 24 Automated Market Making on Polymarket
https://news.polymarket.com/p/automated-market-making-on-polymarket

6 8 9 10 11 12 13 14 15 18 How Polymarket Works | The Tech Behind Prediction Markets
https://rocknblock.io/blog/how-polymarket-works-the-tech-behind-prediction-markets

17 Exploring the Transformation of Polymarket's Pricing Mechanism ...
https://www.panewslab.com/en/articles/fz20kk02b04n

19 21 22 23 models.py
https://github.com/kamekazz/voy_a_mi/blob/2d01757ca1f29132383915384cddbcc0ee3db35a/predictions/models.py