ChatGPT

Building a Polymarket Subgraph on The Graph's Hosted Service

1. Smart Contracts, Addresses, and ABIs in Subgraph Development

Smart Contract Address: In Ethereum (and Polygon), each deployed smart contract has a unique address – essentially the location where the contract's code lives on-chain. For example, Polymarket's core contracts each have a fixed address on Polygon that identifies them. When developing a subgraph, you **must specify the contract addresses** for the data sources you want to index. The Graph will then listen to blockchain events from those addresses only [1].

ABI (Application Binary Interface): An ABI is a JSON-formatted description of a smart contract's functions, events, and data structures. It tells The Graph how to decode event logs and call data from that contract. In subgraph development, you provide the ABI so that the Graph Node knows how to interpret the contract's events and function signatures. When initializing a subgraph, you supply the contract address and the ABI file – if the contract is verified, the Graph CLI can fetch the ABI for you, or you can provide it manually [1]. The ABI is essential for writing mapping handlers that transform on-chain event data into your subgraph's entities. In summary, **the address targets the correct contract, and the ABI ensures the subgraph can** understand **that contract's data.**

For example: If you index Polymarket's Conditional Tokens contract (address on Polygon) in your subgraph, you will include that address in your `subgraph.yaml` and attach its ABI. The Graph will then process events like `ConditionPreparation` or `PositionSplit` from that address and decode the event fields using the ABI definitions. Without the correct ABI, those events' data (like condition IDs, outcome indexes, etc.) would not be interpretable. When using The Graph's CLI, you can usually grab the ABI from a block explorer. *E.g.*, on Polygonscan's contract page you can find a **"Contract ABI"** section and click "Export ABI" to download the JSON [2]. This JSON file is then placed in your subgraph's `abis/` directory and referenced in the subgraph manifest.

2. Key Polymarket Contract Addresses on Polygon

Polymarket's on-chain ecosystem involves a few primary contracts. To index **all trades, market events, resolutions, and positions** since 2020, you'll want to include these contracts in your subgraph:

- **Conditional Tokens Core (Polymarket's Main Market Contract):** This is the **foundation of Polymarket's markets**, implementing Gnosis's Conditional Tokens Framework (CTF). It's an ERC-1155 contract that manages outcome tokens and conditions. **Address (Polygon):** `0x4D97DCd97eC945f40cF65F87097ACe5EA0476045` [3]. This contract emits events for market creation, token splits/merges, and resolution outcomes. In Polymarket's case it handles functions like `prepareCondition` (creating a market question), `splitPosition` (minting yes/no outcome tokens by locking USDC collateral), `mergePositions` (burning outcome tokens back into USDC), and `reportPayouts` (resolving a market) [4]. Indexing this address

will capture **market creation events**, **token mint/burn events**, and **resolution events** (detailed below).

- **Polymarket CTF Exchange (Trading Contract – legacy for binary markets):** This contract facilitates **trades of the outcome tokens for USDC** on-chain. Polymarket originally used a single exchange contract for matching trades on binary (Yes/No) markets. **Address (Polygon):** `0x4bFb41d5B3570DeFd03C39a9A4D8dE6Bd8B8982E` [5]. It's often labeled "Polymarket CTF Exchange" on Polygonscan [6]. This contract handles the **order book/AMM logic** — users submit buy/sell orders off-chain which are settled on-chain via this contract (it performs atomic swaps between the ERC-1155 outcome tokens and the USDC collateral) [7] [8]. Important events emitted here include: `TokenRegistered` (when a new market's token pair becomes tradeable), `OrderMatched` (when a buyer and seller are matched – i.e. a trade execution), and `OrderFilled` (details of each filled order, including price and amount) [9] [10]. By indexing this address, you can capture **all trade events** (each trade will usually emit an OrderFilled or OrderMatched event with price, size, and possibly trader info).

- **Polymarket CTF Exchange (current NegRisk version for multi-outcome markets):** In late 2022/2023 Polymarket introduced an upgraded exchange to support multi-outcome markets and "negative risk" features. **Address (Polygon):** `0xC5d563A36AE78145C45a50134d48A1215220f80a` [5]. If you plan to index **all historical data up to the present**, you should include this newer exchange as well. It emits similar events (`TokenRegistered`, `OrderMatched`, `OrderFilled`) for newer markets. Polymarket's docs note that the old exchange covers legacy binary markets, while this **NegRisk CTF Exchange** covers multi-outcome markets [11]. In a subgraph, you can include both exchange addresses as separate data sources to ensure you capture trades from both phases of Polymarket's platform.

- **Resolution/Oracle Contracts:** Polymarket uses the Conditional Tokens contract's built-in resolution mechanism (`reportPayouts`), often called by an **oracle address**. Each market condition has an *oracle* (an address allowed to report the outcome). In early Polymarket, the oracle might have been a team-controlled address or a resolution contract. Later, Polymarket integrated **UMA's Optimistic Oracle** for decentralized resolution on some markets (via an UMA adapter contract). For subgraph purposes, the key event is still the `ConditionResolution` event emitted by the main Conditional Tokens contract when a market is resolved [4] [12]. You can index that event to capture resolution data (which condition ID was resolved and the outcome). If needed, you might also include the UMA adapter's address/events (e.g. `QuestionInitialized` events on Polymarket's UMA Oracle adapter) to get additional context like question text or ancillary data, but this is optional. The **UMA adapter** contract (if indexing) is at another address (not strictly required for basic trade/position data).

- **USDC Token Address (collateral):** Polymarket markets use USDC (Polygon) as collateral for trades. The USDC token contract on Polygon is `0x2791Bca1f2de4661ED88A30C99A7a9449Aa84174`. You generally do *not* need to index the USDC contract directly; instead, track USDC amounts from events on the exchange (which will indicate how much USDC was transferred per trade). However, if you wanted to double-check balances or record every USDC transfer related to Polymarket, you could use the exchange contract's events or even index `Transfer` events from USDC filtered by Polymarket contract addresses (where sender or receiver is the exchange). In most cases, Polymarket's own events give you the needed trade volumes, so adding USDC as a data source isn't necessary.

Each of the above addresses has verified source code on Polygonscan (and thus verified ABIs). For instance, Polymarket's Conditional Tokens contract page on Polygonscan is labeled **"Polymarket: Conditional Tokens"** and the CTF Exchange is labeled **"Polymarket : CTF Exchange"**, confirming you have the correct addresses [13] [14]. These are the "canonical" addresses to use in your subgraph manifest.

3. Obtaining Polymarket's Verified ABIs

To write the subgraph mappings and decode events, you will need the ABI files for each contract above. Here are several ways to get them:

- **Block Explorer (Polygonscan):** For any verified contract, Polygonscan provides an **ABI export**. Navigate to the contract's page (e.g. [Polymarket Conditional Tokens on Polygonscan] [13] or [Polymarket CTF Exchange on Polygonscan] [14]). Under the "Contract" tab, scroll to the **"Contract ABI"** section. There is usually a button to "Download ABI" or "Export ABI" – clicking this will give you the JSON ABI for that contract [2]. Save the JSON as a file (for example, `ConditionalTokens.json` or `PolymarketExchange.json`). In your subgraph project, create an `abis/` folder and put these files there.

- **Polymarket GitHub Repository:** Polymarket has open-sourced their subgraph code, including ABI files. You can find a folder named `abis` in the polymarket-subgraph repository. For example, there should be JSON files for the conditional tokens contract and the exchange contracts. Using those ensures you have the exact ABIs that Polymarket used. (If using this repo, double-check you pick the ABI for the correct network – Polymarket is on Polygon, and the ABIs in the repo correspond to the Polygon deployments.)

- **Graph CLI auto-fetch:** If you use the Graph CLI `graph init --from-contract` command, it can fetch the ABI from Etherscan/Polygonscan automatically if the contract is verified. You would provide the contract address and network (Polygon) during initialization. Since Polymarket's contracts are verified, the CLI will retrieve the ABI and place it in your subgraph scaffold. For example, initializing with the address `0x4D97DCd97eC945f40cF65F87097ACe5EA0476045` on polygon would auto-populate the ABI in most cases. (If this fails, use one of the manual methods above.)

Why ABIs are essential: The ABI defines each event's structure. For instance, Polymarket's ConditionalTokens ABI includes events like `ConditionPreparation(conditionId, oracle, questionId, outcomeSlotCount)`. In your **mapping.ts** you might have a handler like `handleConditionPreparation(event: ConditionPreparationEvent): void`. The Graph will use the ABI to know that `event.params.conditionId` is a `bytes32`, `event.params.oracle` is an `address`, etc., and it can decode those from the raw log data. Similarly, for the exchange's `OrderFilled(uint256 orderId, address maker, address taker, uint256 outcomeToken, uint256 amount, uint256 price)` (example structure), the ABI delineates those fields so your handler can easily access `event.params.price` and so on. Make sure to attach each ABI to the corresponding data source in `subgraph.yaml`.

Tip: Keep ABIs minimal by stripping out unneeded functions. The Graph only cares about events (and any functions you explicitly call in mappings). You can use a simplified ABI JSON containing just the events definitions to reduce bundle size. However, starting with the full verified ABI from Polygonscan is perfectly fine and ensures you don't miss anything.

4. Designing the Subgraph Schema for Polymarket Data

To capture **complete historical trade data** and related market info, you will define a GraphQL schema with entities that model Polymarket's domain: markets, trades, users, positions, etc. Below is a suggested schema structure (as an example) that you can customize:

- **Market Entity:** Represents a prediction market (a condition). Stores static info like the question or metadata hash, and dynamic info like resolution status. Key fields might include:
- `id` – a unique ID for the market. This could be the conditionId (a 32-byte hash) converted to a string (The Graph's IDs are strings).
- `questionId` – the identifier of the question (from `ConditionPreparation` event, often a bytes32 question ID) [4] . Polymarket might use a question ID that ties to an off-chain question; if using UMA, this could link to ancillary data.
- `oracle` – the Ethereum address that is designated as the oracle for this market (from the event). This might be Polymarket's resolution address or the UMA adapter contract address, etc.
- `outcomeSlotCount` – number of outcomes (2 for binary Yes/No markets, potentially more for multi-outcome markets).
- **Relationships:** You can link markets to trades and positions. For example, a Market can have a field `trades: [Trade!]! @derivedFrom(field: "market")` which allows querying all trades associated with that market. Similarly, you might have `positions: [Position!]! @derivedFrom(field: "market")` to get all user positions in that market.
- `creationBlock` or `timestamp` – when the market was created (from the block of the ConditionPreparation event).
- `resolved` (Boolean) – whether the market has been resolved/finalized.
- `resolutionTime` – a timestamp or block number of when resolution occurred.
- `winningOutcome` – an outcome index or identifier for the winning outcome (set when resolved). For binary markets, you could use `winningOutcome = 1` for "Yes" or 0 for "No" (depending on how you index outcomes). Alternatively, store a boolean like `resolvedYes` or store payout ratios. In Polymarket's case, the ConditionalTokens framework will provide payout numerators for each outcome upon resolution [15] – you can store those if needed to indicate result.

- (Optional) `title` or `question` – Polymarket's question text isn't on-chain in the ConditionalTokens events. It might be obtainable via the UMA adapter (as ancillary data) or from an off-chain source. In a pure on-chain subgraph, you might not have the human-readable title. If you integrate the UMA adapter's `QuestionInitialized` events, you could parse the ancillary data for the question title. Otherwise, you might omit this or allow a separate process to hydrate it.

- **Trade Entity:** Represents an executed trade (match between a buyer and seller or a swap with the market maker). Each `Trade` typically corresponds to an `OrderFilled` or `OrderMatched` event on the exchange contract. Fields:

- `id` – a unique ID for the trade. A common pattern is `txHash-logIndex` to uniquely identify each event. For example, concatenate the transaction hash with the log index of the event (since multiple trades can occur in one transaction).
- `market` (Relation) – reference to the `Market` entity that the trade pertains to. This can be set by finding which condition or outcome token was traded. (The `TokenRegistered` event links a conditionId to specific token indices on the exchange [16] ; trades will reference those.) You might store the conditionId on the Trade directly, then link to Market by conditionId.

- `outcomeIndex` – which outcome was bought/sold in this trade. For binary markets, you might use 0 for "No" and 1 for "Yes" (or similar). If the trade event directly specifies an outcome token address or position ID, you'll need to map that to an outcome index. For example, if `OrderFilled` has an argument for the outcome token, you know whether it was the YES token or NO token being transacted. Storing the index (or an enum like "YES/NO") helps query the side of the trade.
- `amount` – quantity of outcome tokens traded. This would come from the event (perhaps `amount` field in `OrderFilled`). It represents how many outcome tokens were exchanged.
- `price` – the price at which the trade executed. Polymarket prices are effectively the fraction of USDC paid per outcome token. In events, you might have to calculate this: Bitquery notes **Price (YES) = USDC_paid / YES_tokens_received** [17]. Sometimes the event gives these values directly. If not, you can derive price if you have both sides: e.g. if `OrderFilled` gives `amount` of outcome and maybe `cost` in USDC, you can compute price = cost/amount. Consider storing price as a `BigDecimal` for easy human-readable queries (The Graph supports BigDecimal via converting big ints).
- `buyer` and `seller` – addresses of the parties. If the exchange's events include maker/taker addresses, you can fill these in. For example, an `OrderMatched` event might include both maker and taker addresses (or order IDs that you can map to addresses). If only one address is directly in the event (perhaps the trader who took an order), you might treat that address as the `buyer` if they bought outcome tokens, and infer the counterparty via other means. In an automated market making scenario, there isn't a distinct seller user (the "seller" is the pool). But Polymarket's system is order-book based, so generally there is a maker and a taker for each trade. Include both if possible for completeness.
- `isBuy` (Boolean) – you could include a field to denote if the perspective of the recorded trade is a purchase of outcome token (`true` if the initiator bought YES for USDC, `false` if they sold YES for USDC). This can be derived from order type or by comparing which side of the trade the `taker` was on [18] [19]. It might help when calculating positions.
- `timestamp` – the block timestamp when the trade happened. The Graph will give you `event.block.timestamp` for the event; store it so you can do time-series queries or daily volume aggregations.
- `txHash` – (optional, since id already encodes it) the transaction hash for reference.

- **Derived relationships:** You might want to link Trade to the User entities as well. For instance, a Trade could have `buyerUser: User! @relation` and `sellerUser: User!`, and the User entity can have a list of `trades` (or separate lists for buys and sells) via `@derivedFrom`. This way you can query all trades a particular user was involved in.

- **User Entity:** Represents a user (address) who participated in markets. This is mainly to normalize addresses and allow easy querying of "all trades by user" or "all positions of user". Fields:

- `id` – the user's address (as a hex string).
- Perhaps aggregate stats like `totalVolume` or `tradeCount` can be computed if desired (you can update these in mapping every time the user does a trade).
- Relationships: `trades` – you can have `trades: [Trade!]! @derivedFrom(field: "buyer")` and maybe another for seller, or a combined approach. Alternatively, simply have one `trades` list and include both buyer and seller trades in it (you would push to this list from both perspectives). Simpler: treat "trades" as all trades where this user was either buyer or seller. In practice you might create two separate user trade entities for each trade (one for buyer side, one for seller side) to avoid complexity – but that doubles the count of trades. Many subgraphs just record one trade entity and have both roles inside it. In that case, to get a user's trades you

query for trades with `buyer == user` or `seller == user`. You might not even need the User entity to query that (GraphQL can filter by buyer or seller field). However, having a User entity with lists of positions and trades can be handy in GraphQL queries (less filtering needed).

- `positions: [Position!]! @derivedFrom(field: "user")` – list of all Position entities for this user (see below).

- **Position Entity:** Represents a user's position in a given market – essentially how many outcome tokens they hold (and thus their current stake). Since Polymarket uses tokens for outcomes, a user's position can be tracked by token balances. However, token transfers happen frequently (every trade is a transfer of outcome tokens between users or between user and exchange contract). Instead of trying to update on every `TransferSingle` (which is doable but heavy), Polymarket's own subgraph takes an aggregate approach. The **"positions" subgraph** likely computes each user's net position per market by processing trades and splits/merges. We can do similarly: Fields:

- `id` – a combination of user + market (e.g. `${userAddress}-${marketId}`). This ensures uniqueness per user-market pair.
- `user` – relation to the User entity.
- `market` – relation to the Market entity.
- `yesBalance` / `noBalance` – for binary markets, store how many "Yes outcome tokens" and "No outcome tokens" the user currently holds. At any given time, a user holding X Yes and Y No implies certain things (if unresolved, they could merge min(X,Y) back to USDC). After resolution, one of these balances will become redeemable for USDC. You will update these balances on each trade or split/merge event. For example, if a user buys 100 YES, you'd add 100 to yesBalance (and possibly the seller's yesBalance decreases). If they sell 50 YES, subtract 50, etc. Similarly track No side. *Note:* If the user always holds either Yes or No exclusively (which is common if they're speculating), one of these might be zero. If you want to support multi-outcome markets, you might have to generalize this. Perhaps have an array or a separate entity for balances per outcome. But for simplicity, focusing on binary, two fields works.
- `collateralLocked` – (optional) how much USDC collateral is effectively locked for this position. In Conditional Tokens, holding one YES and one NO is equivalent to having 1 USDC locked (because you could merge them to reclaim it). If a user only holds YES and no NO, they have a one-sided bet – effectively they paid some cost for those YES tokens. The subgraph could compute an implied "investment" or track how much collateral they've used, but that can also be derived from trade history. You might skip this or compute via trades if needed.

- `pnl` or `initialCost` – (optional) Polymarket's PNL subgraph likely tracks profit and loss per user per market. That involves valuing the position over time and at resolution. This gets complex (you'd need price history or final payout). Unless explicitly needed, you can omit and let users compute PnL off-subgraph.

- **Resolution Entity (optional):** Since a market has a one-to-one relationship with its resolution, you might not need a separate entity; you can just mark fields on the Market. However, if you want a standalone record of the resolution event (maybe to list all resolutions as a history), you could define a Resolution entity. Fields:

- `id` – could use the market's ID or the transaction ID of the resolution.
- `market` – relation to Market.
- `winningOutcome` – which outcome won (or a payout vector).
- `oracle` – the address that reported the outcome (from the event, should match Market.oracle).

- `timestamp` – when resolved.
- `payoutNumerators` – if multi-outcome, an array of payout values each outcome got (for binary, you'd have [0,1] for No win or [1,0] for Yes win, in a scaled manner). In binary you could simplify to a bool or the winning outcome index.
  This entity isn't strictly necessary but could be nice for querying all past resolutions.

Schema Example (GraphQL SDL): To illustrate, here's a simplified schema for key entities (binary market assumption):

```
type Market @entity {
  id: ID!                 # Use conditionId as ID (e.g. "0x...")
  questionId: Bytes!      # The question ID (bytes32)
  oracle: Bytes!          # Oracle address responsible for resolution
  outcomeSlotCount: Int! # Number of outcomes (2 for binary)
  creationTimestamp: BigInt!
  resolved: Boolean!
  resolutionTimestamp: BigInt
  winningOutcome: Int        # Index of winning outcome (0 or 1 for binary;
null if not resolved)
  # Relationships
  trades: [Trade!]! @derivedFrom(field: "market")
  positions: [Position!]! @derivedFrom(field: "market")
}

type Trade @entity {
  id: ID!                     # e.g. "0xabc123...-1" (tx hash + log index)
  market: Market!             # reference to the Market (condition)
  outcomeIndex: Int!          # which outcome was traded (0 or 1 for binary)
  amount: BigInt!             # number of outcome tokens traded
  price: BigDecimal!          # price in USDC per outcome token
  buyer: User!                # user address who bought the outcome
  seller: User!               # user who sold (could be another user or the
LP; if LP, maybe use a placeholder or same as buyer in opposite role)
  timestamp: BigInt!          # trade timestamp (block time)
  transactionHash: Bytes!     # tx hash for reference
}

type User @entity {
  id: ID!                     # user address (checksum hex string)
  tradesAsBuyer: [Trade!]! @derivedFrom(field: "buyer")
  tradesAsSeller: [Trade!]! @derivedFrom(field: "seller")
  positions: [Position!]! @derivedFrom(field: "user")
}

type Position @entity {
  id: ID!                     # e.g. concat of user address and market ID
  user: User!
  market: Market!
  yesBalance: BigInt!         # current YES tokens held by user for this
market
  noBalance: BigInt!          # current NO tokens held
```

```
  # (If multi-outcome, you could replace yesBalance/noBalance with an array
or separate entity OutcomeBalance)
}
```

This schema would allow you to query things like: **all trades for a given market**, a user's current position, etc. For example, a GraphQL query to get all trades in market X with details might look like:

```
{
  market(id: "0x...conditionId") {
    trades(orderBy: timestamp, orderDirection: asc) {
      amount
      price
      outcomeIndex
      buyer { id }
      seller { id }
      timestamp
    }
  }
}
```

And a query for user's positions:

```
{
  user(id: "0x...address") {
    positions {
      market { id oracle resolved winningOutcome }
      yesBalance
      noBalance
    }
  }
}
```

Note: Ensure to update these entities in your mappings. For instance, when a `ConditionPreparation` event is handled, you'd create a new `Market` entity with the data (conditionId, oracle, outcomes, etc.). When an `OrderFilled` / `OrderMatched` comes in, create a `Trade` entity and update the corresponding Users and their Position balances. Also handle `PositionsSplit` and `PositionsMerge` events – these events indicate a user minted or redeemed outcome tokens by interacting with the Conditional Tokens contract directly (which can also affect their position even without a trade). Polymarket's main contract events include `PositionSplit` (user splits USDC into outcome tokens) and `PositionsMerge` (user merges outcomes back) [20] [21]. Your subgraph should handle those to update Position balances (in a split, if user splits 100 USDC, they gain 100 YES and 100 NO in their balance; in a merge, they lose tokens and essentially get back collateral off-chain). Additionally, the `TransferSingle` / `TransferBatch` (ERC1155 standard events) from the ConditionalTokens contract can be used for fine-grained tracking of token movements between addresses (the exchange contract address and user addresses). For simplicity, many subgraphs avoid handling every transfer and instead use the higher-level events, but transfers can help identify, for example, when tokens move from user to exchange (when placing an order) or exchange to user (when

order filled). If you go that route, filter on `from` and `to` in those events to distinguish trade-related movements.

5. Reference Implementations and Resources

Polymarket has already built and open-sourced subgraphs which you can study or fork. Notably, **Polymarket's official subgraph repository** (polymarket-subgraph on GitHub) contains multiple subgraph definitions for their platform [22]. This repository is a goldmine for understanding how to index Polymarket data. It is organized into subfolders (e.g. *fpmm-subgraph*, *orderbook-subgraph*, *activity-subgraph*, *positions-subgraph*, *oi-subgraph*, *pnl-subgraph*, etc.) corresponding to different aspects of the data. For example, the **"activity subgraph"** likely tracks trades and volume, the **"positions subgraph"** tracks user holdings, **"oi"** is open interest, **"pnl"** is profit-and-loss, etc. You can refer to their `schema.graphql` files in each for insight on entity design, and the mapping logic in `mapping.ts` files for how they calculate derived metrics. Since Polymarket's team designed these subgraphs for their production system, they serve as an accurate reference. *(For instance, the Polymarket Positions subgraph computes positions in a similar way to what we described, and the Trades/Activity subgraph will show how they record each trade event.)*

Beyond the code, here are some public endpoints and resources you can use:

- **The Graph Hosted Service API (legacy):** Polymarket's subgraph was originally deployed on The Graph's hosted service under the name `polymarket/matic-markets`. You can find it here: [23]. Using The Graph's GraphiQL interface, you can query this subgraph to see the available entity types and fields. For example, you might find entities like `markets`, `outcomeToken`, `tradeLogs`, etc., depending on how they structured it. This can confirm what information is indexed. *(Keep in mind that Polymarket might have shifted to other indexing services, so the hosted service subgraph might be outdated, but it's still a good reference for the schema.)*

- **The Graph Explorer (Decentralized Network):** Polymarket has subgraphs on the decentralized Graph network as well. The official **Polymarket Subgraph** is accessible in the Graph Explorer (you can find it by searching "Polymarket" on thegraph.com/explorer). The Graph's documentation specifically highlights a Polymarket subgraph page [24]. There, you can view the schema and try out queries in the playground. According to The Graph's guide, Polymarket's schema includes entities for positions (redeemable outcome holdings) and possibly events like redemptions [25] [26]. The Graph Explorer also shows **additional Polymarket subgraphs** for activity, PnL, open interest, etc., which correspond to the multiple subgraphs Polymarket uses [27]. You can click those and inspect their schemas too. This is an excellent way to verify your own schema design against an official one.

- **Polymarket Documentation:** The developer docs at docs.polymarket.com have a **Subgraph** section. In particular, the "Subgraph Overview" page confirms that *"Polymarket has written and open sourced a subgraph that provides … event indexing for volume, user positions, market and liquidity data"* [22]. They mention it's hosted by a third-party (Goldsky) and give GraphQL playground links for each subgraph (Orders, Positions, Activity, Open Interest, PnL) [28]. While those are read-only API endpoints, they indicate how Polymarket splits the data. You can mimic a similar approach or combine everything into one subgraph as per your needs. The Polymarket docs also provide some details on the on-chain framework (Conditional Tokens, etc.) that can deepen your understanding – for example, see *"Conditional Token Framework – Deployment and Additional Information"* where the contract addresses are listed and a link to an audit is provided [29].

- **Community and Tutorials:** The Ethereum developer community has discussed Polymarket data indexing. For instance, the Bitquery documentation we cited has dedicated sections on Polymarket, including example GraphQL (Bitquery's GraphQL, not to be confused with subgraph GraphQL) to fetch Polymarket data [4] [30]. While Bitquery's approach differs (it queries their data warehouse), the breakdown of events and data (condition preparation, order filled events, etc.) in their docs can guide your mapping functions. Additionally, The Graph's forum or discord might have threads on Polymarket subgraphs if you search for them, and the *ethfinance* Reddit discussion [23] shows how others have tried querying Polymarket's subgraph for historical prices, which might give hints on which entities/fields exist.

- **Example Repositories:** Apart from Polymarket's own repo, you might not find many third-party subgraphs for Polymarket since the official one is available. However, you could look at similar prediction market subgraphs (like Augur or Omen by Gnosis) for inspiration. Gnosis's Omen used the same Conditional Tokens framework, and their subgraph (if available) would have analogous entities (markets, outcome token balances, liquidity, etc.). TheGraph's official docs also have a **Polymarket subgraph guide** [31] which we referenced – it doesn't show code, but it confirms what's possible with queries.

In summary, leverage Polymarket's open-source subgraph code **as a baseline** – you can even fork it and modify the schema to unify all data in one subgraph if that's your goal. Ensure you index the **Conditional Tokens contract** (for market events and positions) and the **Exchange contracts** (for trades). With a well-designed schema (Market, Trade, User, Position, Resolution, etc.) and the correct ABIs and handlers, you will be able to query historical trades per market, user holdings, and market outcomes easily via GraphQL. Good luck with your custom Polymarket subgraph!

Sources:

- The Graph Documentation – *Subgraph Quick Start (requirements for contract address & ABI)* [1]
- Medium Tutorial – *Exporting ABI from Etherscan/Polygonscan* [2]
- Polymarket Developer Docs – *Subgraph Overview* (open-source subgraph and data types) [22]
- Polymarket Developer Docs – *Conditional Tokens Deployment* (Polygon addresses for core contracts) [29]
- Polygonscan – Verified Contracts labeled "Polymarket" (address confirmation) [13] [14]
- Bitquery Documentation – *Polymarket Main Contract (CTF) API* [4] and *CTF Exchange API* [11] [9] (event descriptions and addresses)
- The Graph Guides – *Querying Polymarket with Subgraphs* (Graph Explorer, schema and queries) [24] [26]
- Reddit r/ethfinance – Polymarket subgraph reference (hosted service link and discussion) [23]

---

[1] Quick Start | Docs | The Graph
https://thegraph.com/docs/en/subgraphs/quick-start/

[2] Creating your own Subgraph: A Step-by-Step Hands-On Guide With Mocaverse Example | by EvanW | Medium
https://medium.com/@evvvv/creating-your-own-subgraph-a-step-by-step-hands-on-guide-with-example-33d8787cfd96

[3] [4] [12] [15] [20] [21] Main Polymarket Contract API | Bitquery
https://docs.bitquery.io/docs/examples/polymarket-api/main-polymarket-contract/

[5] [9] [10] [11] [16] [17] [30] Polymarket CTF Exchange API | Bitquery
https://docs.bitquery.io/docs/examples/polymarket-api/polymarket-ctf-exchange/

6 14 Polymarket : CTF Exchange | Address: 0x4bFb41d5...Bd8B8982E | PolygonScan
https://polygonscan.com/address/0x4bFb41d5B3570DeFd03C39a9A4D8dE6Bd8B8982E

7 8 GitHub - Polymarket/ctf-exchange: Polymarket CTF Exchange
https://github.com/Polymarket/ctf-exchange

13 Address: 0x4D97DCd9...EA0476045 | PolygonScan
https://polygonscan.com/address/0x4D97DCd97eC945f40cF65F87097ACe5EA0476045

18 19 Onchain Order Info - Polymarket Documentation
https://docs.polymarket.com/developers/CLOB/orders/onchain-order-info

22 28 Overview - Polymarket Documentation
https://docs.polymarket.com/developers/subgraph/overview

23 Help: How to query polymarket data? : r/ethfinance
https://www.reddit.com/r/ethfinance/comments/x9xsmq/help_how_to_query_polymarket_data/

24 25 26 27 31 Querying Blockchain Data from Polymarket with Subgraphs on The Graph | Docs | The Graph
https://thegraph.com/docs/en/subgraphs/guides/polymarket/

29 Deployment and Additional Information - Polymarket Documentation
https://docs.polymarket.com/developers/CTF/deployment-resources