Cross-Venue Arbitrage Architecture: Algorithmic Reconciliation of Prediction Market Events

1. Structural Divergence in Binary Derivatives Markets

The fundamental engineering challenge in developing an arbitrage system between Kalshi and Polymarket lies not in the arithmetic of the trade—identifying when the sum of opposing outcomes is less than unity—but in the reconciliation of fundamentally disjoint data taxonomies. We are presented with two distinct market microstructures: Kalshi, a Designated Contract Market (DCM) regulated by the Commodity Futures Trading Commission (CFTC), and Polymarket, a global platform operating on the Polygon blockchain using the Gnosis Conditional Token Framework (CTF). This regulatory and technological bifurcation results in divergent API architectures, specifically in how recurring events are instantiated, labeled, and indexed.

To implement a robust Python-based arbitrage strategy, one must first deconstruct the "Series" architecture of Kalshi, which serves as the structured anchor for recurring economic events (e.g., CPI releases, Federal Reserve interest rate decisions). The core task—finding the "equivalent" on Polymarket—is non-trivial because Polymarket utilizes a graph-like structure of Events and Markets loosely coupled by Tags, rather than a rigid, regulatory-defined Series template. This report provides an exhaustive technical analysis of the Kalshi get_series endpoint and its Polymarket counterparts, culminating in a detailed algorithmic strategy for programmatic event matching and order book reconciliation.

1.1 The Regulatory Impact on Data Taxonomy

The structure of an API is often a mirror of the legal framework governing the exchange. Kalshi, as a CFTC-regulated entity, must define its contracts with statutory precision. A "Series" on Kalshi is not merely a collection of events; it is a regulatory filing. When Kalshi lists "CPI," it refers to a specific contract template approved by the CFTC, governed by a Rulebook that dictates settlement sources, expiration times, and strike price intervals.[1] This necessitates a hierarchical database model where a Series parent spawns Event children, which in turn contain Market siblings.

Conversely, Polymarket operates with the flexibility of a decentralized protocol. Its "markets" are smart contracts. While it possesses a concept of "Series," typically utilized for sports leagues (e.g., "NBA 2024"), its economic and political markets often emerge organically or via governance proposals. These are categorized via "Tags" or "Categories" rather than a rigid inheritance model.[3] Consequently, an arbitrage bot cannot simply map SeriesID_Kalshi to

SeriesID_Polymarket. Instead, it must employ a "Reconciliation Engine"—a software layer capable of semantic translation, mapping the rigid syntax of Kalshi's Series to the fluid, natural-language metadata of Polymarket's Gamma API.

1.2 The Liquidity Landscape and Arbitrage Opportunity

The arbitrage logic requested—identifying scenarios where Price(Yes_A) + Price(No_B) < $1.00—relies on the isolation of liquidity pools. Kalshi orders are matched via a Central Limit Order Book (CLOB) settled in USD held in clearinghouse accounts. Polymarket orders are matched via a hybrid CLOB/AMM (Automated Market Maker) system settled in USDC on-chain.[5] These pools do not communicate; price discovery is asynchronous. This latency allows for "negative spreads" to persist, provided the automated system can mathematically verify that "Event A" on Kalshi and "Event B" on Polymarket describe the exact same real-world outcome. The failure to rigorously verify this identity is the primary risk vector; a "CPI > 3.0%" market is distinct from a "CPI > 3.1%" market, yet simple string matching might confuse them. The Python implementation must therefore prioritize exact entity extraction over fuzzy similarity.

---

2. Kalshi API Architecture: The Recurring Event Anchor

The Python implementation typically begins by querying Kalshi. Because Kalshi's data model is stricter, it acts as the "Master" data source in the reconciliation relationship. We retrieve the "Truth" from Kalshi's Series and then attempt to locate its reflection in the noisier Polymarket ecosystem.

2.1 The get_series Endpoint: Definition and Utility

The primary entry point for discovering recurring arbitrage opportunities is the get_series endpoint. This endpoint exposes the templates for all recurring markets, allowing the Python application to anticipate future events and understand the structure of current ones.

Endpoint Specifications:

- **Method:** GET
- **Path:** /series (or /series/{series_ticker})
- **Library:** kalshi-python SDK or direct HTTP requests to https://api.elections.kalshi.com/trade-api/v2.
- **Key Response Fields:**
  - ticker: The immutable alphanumeric identifier (e.g., KXHIGHNY for NYC High Temp, CPI for Consumer Price Index).
  - title: The canonical name of the recurring event.
  - frequency: The periodicity (e.g., monthly, daily, weekly).
  - category: The sector (e.g., Economics, Politics, Weather).

The utility of get_series for the Python implementation is twofold. First, it allows for **pre-filtering**. An arbitrage bot focused on economic indicators can fetch all series and filter for category == 'Economics', discarding thousands of irrelevant weather or pop-culture markets. Second, it provides the **search terms** for Polymarket. The title field in the Series response (e.g., "Federal Funds Rate") becomes the seed keyword for querying the Polymarket Gamma API.[1]

2.2 Navigating the Hierarchy: Series to Markets

Once a target Series is identified (e.g., CPI), the application must traverse down to the tradable assets. This involves a two-step API cascade:

1. **Fetch Events:** Using GET /events with the query parameter series_ticker=CPI. This returns a list of specific temporal instances (e.g., "CPI Release for March 2026"). The response object contains the event_ticker and crucial temporal metadata like mutual_exclusive (boolean) and strike_date.[10]
2. **Fetch Markets:** Using GET /markets with the query parameter event_ticker={event_ticker}. This returns the actual binary contracts.

Critical Data Points for Matching:

To enable precise matching with Polymarket, the Python code must extract specific attributes from the Market object that are often embedded in text fields:

- **Market Title:** Often contains the variable condition (e.g., "> 3.0%").
- **Subtitle:** Often contains the context (e.g., "Consumer Price Index annual change").
- **Expiration Time:** The ISO 8601 timestamp representing the cessation of trading or the determination of the outcome. This is the single most important validator for cross-exchange matching.[12]

2.3 Handling Recurrence and Naming Conventions

Kalshi's naming convention is formulaic. A series ticker like INTRATE (Interest Rate) will spawn events with tickers like INTRATE-24MAR20. The Python strategy can exploit this predictability. By parsing the suffix, the system can determine the exact date of the event without needing to read the full description. This contrasts with Polymarket, where titles are often human-written and inconsistent (e.g., "Fed March Rate Cut" vs. "March Fed Decision"). The Kalshi Series API thus provides the *structured* data against which the *unstructured* Polymarket data is measured.

Data Point	Kalshi Source	API Field	Example Value
Series Name	GET /series	title	"Fed Funds Rate"

Event Date	GET /events	strike_date	"2026-03-20T18:00:00Z"
Strike Price	GET /markets	title	"Target range > 5.25%"
Contract ID	GET /markets	ticker	INTRATE-24MAR20-T5.25

---

3. Polymarket Gamma API: Locating the Equivalent Interface

The user's request explicitly asks to "search the polymarket api details so that you find the equivalent api" to Kalshi's get_series. A superficial analysis might suggest Polymarket's GET /series endpoint is the direct equivalent. However, deep research into the Gamma API documentation and usage patterns reveals that for *economic and political arbitrage*, this is often a false cognate. The functional equivalent required for the Python implementation is the **Tags** system combined with the **Events** endpoint.

3.1 The "Series" Endpoint: A False Friend?

Polymarket's Gamma API does expose a /series endpoint (https://gamma-api.polymarket.com/series). The documentation indicates this endpoint returns objects containing tickers, titles, and recurrences. However, practical analysis of the API's usage shows it is predominantly populated with **Sports Leagues** (e.g., NBA, NFL, Premier League) where the "Series" structure (Season -> Game) maps cleanly.

For economic events like the CPI or Fed Rates, Polymarket typically does *not* group these under a formal "Series" object accessible via /series. Instead, these are standalone events linked by shared metadata. Relying solely on GET /series would cause the Python bot to miss the vast majority of economic arbitrage opportunities. Therefore, while GET /series is the *nominal* equivalent, the *functional* equivalent for this strategy is GET /tags.[3]

3.2 The Functional Equivalent: Tags and Categorization

To programmatically replicate the functionality of "fetching all CPI events" (as one would on Kalshi), the Python script must interact with the **Tags** endpoints.

- **Endpoint:** GET /tags
- **Utility:** This returns a list of all categories used on the platform. The script should cache this list and search for IDs corresponding to "Economics", "Fed", "Inflation", "CPI",

"Politics", and "Elections".

- **Endpoint:** GET /events (with filters)
  - **Query Parameters:** tag_id={id}, closed=false, active=true, limit=100.
  - **Behavior:** This returns a list of active events associated with the tag. This is the direct operational counterpart to Kalshi's GET /events?series_ticker=... workflow.

This structural difference dictates the Python architecture: On Kalshi, we iterate through *Series*. On Polymarket, we iterate through *Tags*.

3.3 The Gamma Event Object: Nested Complexity

Polymarket's data model is more nested than Kalshi's. A single "Event" object (retrieved via GET /events) acts as a container for multiple "Markets" (tradable contracts).

The "Group" Market Nuance:

In Kalshi, "Fed Rate > 5.0%" and "Fed Rate > 5.25%" are usually separate Market objects (though they may share a parent Event). In Polymarket, these are often consolidated into a single **Group Event**.

- **Event:** "Fed Interest Rates: March"
- **Market 1:** "5.00-5.25%"
- **Market 2:** "5.25-5.50%"
- **Market 3:** "5.50-5.75%"

The Python implementation must handle this nested array. It cannot simply match Event-to-Event; it must match **Kalshi Market** to **Polymarket Market**. Crucially, the detailed description of the outcome in a Polymarket Group Market is often found in the groupItemTitle field or embedded within the description or question fields of the inner market object.[16]

3.4 Retrieving the CLOB Token IDs

For the final step of the user's request—comparing prices—the Gamma API provides the essential "keys" to the order book.

- **Field:** clobTokenIds
- **Description:** An array of strings representing the ERC-1155 token IDs for the "Yes" and "No" outcomes.
- **Structure:** ``. typically, Index 0 is "Yes" (or the primary outcome) and Index 1 is "No".
- **Usage:** These IDs are passed to the CLOB API (clob.polymarket.com) to fetch the live bid/ask spreads. The Gamma API itself only provides outcomePrices (implied probabilities from the AMM or last trade), which are insufficient for high-frequency arbitrage precision. The strategy requires identifying these Token IDs during the discovery phase to enable low-latency polling during the execution phase.[3]

---

4. Algorithmic Reconciliation Strategy: The Matching Engine

We have established that Kalshi provides a structured list of Series, and Polymarket provides a list of Events filtered by Tags. The core engineering task is to cross-reference these two lists. Since there is no shared unique identifier (like a CUSIP or ISIN), the Python implementation must employ a **Semantic Matching Strategy**. This strategy relies on three layers of filtration: Temporal, Semantic, and Numerical.

4.1 Layer 1: Temporal Blocking (Date Synchronization)

The most computationally efficient way to reduce the search space is to filter by date. Comparing every active Kalshi market against every active Polymarket market is an $O(N \times M)$ operation. Blocking by date reduces this significantly.

Python Logic:

1. Extract expiration_time from the Kalshi market object. Convert to a UTC Date object.
2. Extract endDate (or resolutionDate) from the Polymarket event object. Convert to a UTC Date object.
3. **The "Fuzzy Date" Problem:** Kalshi and Polymarket often have slightly different expiration timestamps for the same event. Kalshi might expire at market close (4:00 PM ET), while Polymarket might expire at midnight UTC.
    - *Strategy:* The code should allow a configurable tolerance_window (e.g., ±24 hours). If abs(kalshi_date - poly_date) <= timedelta(hours=24), the pair is considered a candidate for semantic analysis. If not, it is discarded immediately.[4]

4.2 Layer 2: Semantic Similarity (Fuzzy String Matching)

Once candidates are filtered by date, the system must determine if "CPI > 3.0%" (Kalshi) describes the same phenomenon as "March CPI: 3.0% or higher" (Polymarket).

Library Selection:

The standard difflib in Python is insufficient for this task as it is sensitive to word order. The report recommends rapidfuzz (a faster, MIT-licensed C++ alternative to fuzzywuzzy) for its speed and specific "Token Set" algorithms.

Algorithm: token_set_ratio

This algorithm tokenizes the string, sorts the tokens alphabetically, and then compares them. This neutralizes the difference between "Fed Rates March" and "March Fed Rates."

- **Input A:** "Federal Funds Rate target range" (Kalshi Series Title) + "Target range > 5.25%" (Kalshi Market Title).

- **Input B:** "Fed Rates: March 2026" (Polymarket Event Title) + "5.25-5.50" (Polymarket Market Group Item Title).
- **Processing:** Concatenate the Series/Event context with the Market specifics for both sides before comparing.
- **Threshold:** A score > 85 (out of 100) is typically required to trigger the next layer of validation.[21]

4.3 Layer 3: Entity Extraction (The "Hard" Filter)

In financial markets, a "similar" title is dangerous. "CPI > 3.0%" and "CPI > 3.1%" are string-similar but mutually exclusive. Arbitraging them against each other would result in a guaranteed loss if the actual CPI comes in at 3.05%. Therefore, the Python code must implement **Exact Numerical Matching** using Regular Expressions (Regex).

Strategy:

1. **Extract:** Use Python's re module to find all numerical sequences (integers and floats) in both titles.
   - re.findall(r"[-+]?\d*\.\d+|\d+", text)
2. **Compare:** The "Strike Price" (e.g., 3.0) extracted from the Kalshi title *must* exist in the set of numbers extracted from the Polymarket title.
3. **Unit Awareness:** The code should also normalize units. It should recognize that "3%" and "0.03" are equivalent, though in prediction markets, "3.0" usually appears as "3.0" in the text strings.

Handling Ranges vs. Thresholds:

A common edge case is Kalshi using a threshold (" > 5.25") and Polymarket using a range ("5.25 - 5.50").

- *Logic:* If Kalshi is " > X", and Polymarket is "X - Y", this is a match *if and only if* the arbitrage strategy accounts for the upper bound risk. Ideally, the code should look for Polymarket markets that explicitly state " > X" or "X or higher". If matching a Range against a Threshold, the "No" side of the Polymarket Range ("Not 5.25-5.50") does *not* perfectly hedge the "Yes" side of the Kalshi Threshold ("> 5.25"), because the event could land at 5.60. The Python strategy must strictly pair " > X " with " > X " or construct a synthetic position (buying multiple ranges), though simpler 1:1 matching is recommended for the initial implementation.

---

5. Python Implementation Blueprint

This section details the architectural strategy for the Python code, focusing on modularity, rate limiting, and the specific library interactions required.

5.1 Architecture Overview

The application should be structured as an asynchronous event loop to handle the I/O-bound nature of querying two different APIs.

- **MarketReferenceFetcher Class (Kalshi):**
  - *Responsibility:* Queries get_series and get_markets.
  - *Output:* A normalized list of ArbitrageCandidate objects containing {source: 'Kalshi', ticker, normalized_title, strike, expiration, bid_ask_tuple}.
- **PolymarketDiscovery Class (Polymarket):**
  - *Responsibility:* Queries get_tags and get_events.
  - *Output:* A normalized list of ArbitrageCandidate objects. It handles the extraction of clobTokenIds.
- **ReconciliationEngine Class:**
  - *Responsibility:* Runs the matching logic (Date Block -> Fuzzy Match -> Regex Validator).
  - *Output:* A list of MatchedPair objects linking a Kalshi Ticker to a Polymarket CLOB Token ID.
- **ArbitrageExecutor Class:**
  - *Responsibility:* Polls prices for the MatchedPair list and checks the " < 50 cents" (or < $1.00 total cost) logic.

5.2 Handling Rate Limits and Pagination

Both APIs enforce rate limits. The Python code must implement **Exponential Backoff**.

- **Kalshi:** strict limits (e.g., 10 req/sec for public/basic tiers).
- **Polymarket:** Gamma API is public but rate-limited; CLOB API requires authentication for higher limits.
- **Strategy:** Use the tenacity Python library to decorate API fetching functions.
  - @retry(wait=wait_exponential(multiplier=1, min=4, max=10))
  - Pagination: Both get_markets (Kalshi) and get_events (Polymarket) are paginated. The code must loop through cursor (Kalshi) or offset (Polymarket) until all candidates are retrieved.[10]

5.3 Asynchronous Execution

Given the need to check multiple markets simultaneously, the asyncio and aiohttp libraries are preferred over synchronous requests.

- *Logic:* Fetching 100 Polymarket order books sequentially is too slow. The prices will move. The code should spawn async tasks to fetch the Kalshi Orderbook and Polymarket CLOB Book for a matched pair in parallel, minimizing the "leg risk" (the risk that one price moves while you are checking the other).

---

6. Execution Data: Comparing Bids and Asks

The user's prompt emphasizes ensuring the code compares "yes and no bid ask prices". This requires interacting with the order book APIs, not just the metadata APIs.

6.1 Kalshi Orderbook Structure

Kalshi order books are typically "Yes" centric.

- **Endpoint:** GET /markets/{ticker}/orderbook
- **Response:** yes_bid, yes_ask.
- **Deriving "No":** Kalshi often implicitly structures the "No" price as the reciprocal of the "Yes" price in a fully efficient market, but for arbitrage, we look for the explicit liquidity. The API often provides no_bid and no_ask directly in the detailed order book view. If not, No_Ask = 100 - Yes_Bid (conceptually), but for *execution*, we need the explicit limit orders.

6.2 Polymarket CLOB Data

The Gamma API gives outcomePrices (e.g., "0.65"), but this is a midpoint or last-trade price. For the arbitrage logic ("Yes + No < 1.00"), we need the **Asks**.

- **Endpoint:** GET https://clob.polymarket.com/book?token_id={id}
- **Response:** A JSON object with bids and asks arrays, sorted by price.
- **Token Mapping:**
  - To buy "Yes" on Polymarket: Query the book for clobTokenIds. Look at the asks array. Take the lowest price.
  - To buy "No" on Polymarket: Query the book for clobTokenIds. Look at the asks array.

6.3 The Price Comparison Logic (The Strategy)

The Python code must normalize prices to a common format (Decimal USD).

- **Kalshi:** Prices are integers (Cents). 50 = $0.50.
- **Polymarket:** Prices are strings/floats. "0.50" = $0.50.

The Comparison Loop:

Python

```python
# Conceptual Logic
cost_path_A = kalshi_yes_ask_decimal + polymarket_no_ask_decimal
cost_path_B = kalshi_no_ask_decimal + polymarket_yes_ask_decimal
```

```
if cost_path_A < 1.00:
    print(f"Arbitrage Opportunity: Buy Yes on Kalshi ({kalshi_yes_ask}), Buy No on Poly
({polymarket_no_ask})")
if cost_path_B < 1.00:
    print(f"Arbitrage Opportunity: Buy No on Kalshi ({kalshi_no_ask}), Buy Yes on Poly
({polymarket_yes_ask})")
```

This logic satisfies the user's requirement to identify when the "yes and no side... are both below 50 cents" (or sufficiently low to sum to < $1.00).

---

7. Implications and Edge Cases

7.1 Negative Risk Markets on Polymarket

A crucial "third-order" insight is the existence of "Negative Risk" markets on Polymarket. In markets with three or more mutually exclusive outcomes (e.g., "Fed Rate: A, B, or C"), the "No" token for outcome A is structurally linked to the "Yes" tokens of B and C.

- *Implication:* The "No" liquidity might be synthetic or fragmented.
- *Strategy:* The Python implementation should prioritize **Binary Markets** (Two outcomes) for the initial version to avoid the complexity of converting Negative Risk liquidity into a "No" price. Check the negRisk boolean field in the Gamma API response and filter out true values if seeking simplicity.[4]

7.2 Fee structures

While the prompt asks to ignore the math of the arbitrage, the Python strategy must acknowledge that "Price < $1.00" implies "Price < 1.00 - Fees".

- **Kalshi:** Charges fees on the *trade* (or withdrawal).
- **Polymarket:** No trading fees on the CLOB (currently), but gas fees or withdrawal fees apply.
- **Coding Recommendation:** Define a constant FEE_BUFFER = 0.02 (2 cents) and check Sum < (1.00 - FEE_BUFFER) rather than strict 1.00 to ensure the code identifies *profitable* trades, not just break-even ones.

7.3 Data Latency

Kalshi's API is centralized; Polymarket's Gamma API is an indexer over a blockchain. Gamma might be 1-2 seconds behind the real-time CLOB.

- *Strategy:* Always use Gamma for *Discovery* (finding the market ID) and the CLOB API for *Pricing* (checking the arb). Relying on Gamma outcomePrices for the arbitrage check will result in "Ghost Arbs" that disappear when you try to execute.

8. Conclusion

The construction of an arbitrage system between Kalshi and Polymarket is an exercise in translation. The Python implementation must act as a bridge between the regulated, structured world of Kalshi Series and the decentralized, tagged world of Polymarket Events.

By anchoring the discovery process on Kalshi’s get_series endpoint to define the universe of recurring economic events, and leveraging Polymarket’s get_tags and get_events as the functional equivalents, the system can systematically identify matching assets. The reconciliation engine, powered by rapidfuzz string matching and Regex-based entity extraction, provides the necessary safety checks to ensure that distinct contracts are not conflated. Finally, by mapping the discovered Market Tickers (Kalshi) and CLOB Token IDs (Polymarket) to their respective order book APIs, the system can execute the precise price comparison requested, identifying risk-free opportunities where the aggregate cost of coverage across venues falls below the settlement value.

API Mapping Summary for Python Implementation

Feature	**Kalshi API**	**Polymarket Equivalent Strategy**
Recurring Templates	GET /series	GET /tags (Filter by Category)
Instance Discovery	GET /events?series_ticker=X	GET /events?tag_id=Y&active=true
Market Definition	GET /markets	GET /events (Nested markets array)
Price (Metadata)	Market last_price	Gamma outcomePrices
Price (Execution)	GET /markets/{ticker}/orderbook	GET /book?token_id={id} (CLOB)
Outcome Keys	yes_ask, no_ask	clobTokenIds (Yes), clobTokenIds (No)

Works cited

1. Series - API Documentation - Kalshi's API, accessed January 15, 2026, https://docs.kalshi.com/python-sdk/api/SeriesApi
2. Kalshi vs Polymarket: Which Is Superior? Markets, Fees & More - RotoGrinders, accessed January 15, 2026, https://rotogrinders.com/best-prediction-market-apps/kalshi-vs-polymarket
3. Fetching Market Data - Polymarket Documentation, accessed January 15, 2026, https://docs.polymarket.com/quickstart/fetching-data
4. How to Fetch Markets - Polymarket Documentation, accessed January 15, 2026, https://docs.polymarket.com/developers/gamma-markets-api/fetch-markets-guide
5. CLOB Introduction - Polymarket Documentation, accessed January 15, 2026, https://docs.polymarket.com/developers/CLOB/introduction
6. Kalshi vs. Polymarket: Which Prediction Market Is Best for You? - Action Network, accessed January 15, 2026, https://www.actionnetwork.com/online-sports-betting/reviews/kalshi-vs-polymarket
7. Quick Start: Market Data - API Documentation - Kalshi's API, accessed January 15, 2026, https://docs.kalshi.com/getting_started/quick_start_market_data
8. Get Series List - API Documentation - Kalshi's API, accessed January 15, 2026, https://docs.kalshi.com/api-reference/market/get-series-list
9. Get Series - API Documentation - Kalshi's API, accessed January 15, 2026, https://docs.kalshi.com/api-reference/market/get-series
10. Get Events - API Documentation, accessed January 15, 2026, https://docs.kalshi.com/api-reference/events/get-events
11. Events - API Documentation, accessed January 15, 2026, https://docs.kalshi.com/python-sdk/api/EventsApi
12. Get Event - API Documentation, accessed January 15, 2026, https://docs.kalshi.com/api-reference/events/get-event
13. Get Market - API Documentation, accessed January 15, 2026, https://docs.kalshi.com/api-reference/market/get-market
14. List series - Polymarket Documentation, accessed January 15, 2026, https://docs.polymarket.com/api-reference/series/list-series
15. polymarketgamma package - github.com/ivanzzeth/polymarket-go-gamma-client - Go Packages, accessed January 15, 2026, https://pkg.go.dev/github.com/ivanzzeth/polymarket-go-gamma-client
16. Gamma Structure - Polymarket Documentation, accessed January 15, 2026, https://docs.polymarket.com/developers/gamma-markets-api/gamma-structure
17. clintmckenna/polymarketR: R Interface to Polymarket Prediction Markets - GitHub, accessed January 15, 2026, https://github.com/clintmckenna/polymarketR
18. Polymarket Markets Scraper - Apify, accessed January 15, 2026, https://apify.com/louisdeconinck/polymarket-events-scraper

19. polymarket-apis · PyPI, accessed January 15, 2026, https://pypi.org/project/polymarket-apis/
20. Get Markets - API Documentation, accessed January 15, 2026, https://docs.kalshi.com/api-reference/market/get-markets
21. Fuzzy string matching in Python (with examples) - Typesense, accessed January 15, 2026, https://typesense.org/learn/fuzzy-string-matching-python/
22. rapidfuzz/RapidFuzz: Rapid fuzzy string matching in Python using various string metrics - GitHub, accessed January 15, 2026, https://github.com/rapidfuzz/RapidFuzz