LLM Agents for Cross-Platform Prediction Market Arbitrage

Abhi Wadhwa
University of Southern California
abhiw@usc.edu

Summer 2026

Abstract

Kalshi and Polymarket price the same real-world events but use different naming conventions, fee formulas, and settlement mechanics. Exploiting the spread between them requires solving three problems in sequence: figuring out which contracts on the two platforms refer to the same event, computing fee-adjusted pricing discrepancies from live order books, and placing trades where settlement is non-atomic. I formalize the microstructure of both platforms, define three arbitrage types (cross-platform, implied same-platform, monotonicity violations), and frame the contract matching problem as entity resolution. I build a four-stage hybrid NLP pipeline—regex extraction, Sentence-BERT embeddings, rapidfuzz string matching, optional LLM verification—and evaluate it on 5,000 synthetic market pairs. The pipeline without any LLM hits 93.9% F1. Real-data evaluation on live Kalshi and Polymarket markets, LLM agent experiments with actual API calls, and paper trading simulations are in progress. Results from a May 2026 live pull revealed that 96.2% of Kalshi's current inventory consists of multi-leg parlays with no single-event Polymarket equivalent, which I did not anticipate.

1 Introduction

This project started with a football game. Chiefs–Bills, October 12, 2025. I was up at 2 AM refreshing two browser tabs, which is not a habit I recommend, and I noticed Kalshi had Kansas City at 47 cents while Polymarket had them at 52. Five cents of spread on a contract that pays a dollar. After fees the edge was about three cents per contract, so I bought YES on Kalshi and NO on Polymarket. Made $2.80 on a hundred contracts. Then I sat there wondering how many times a day this sort of thing happens and nobody notices.

Answer: a lot. Kalshi is the first CFTC-regulated designated contract market for event contracts, NYC-based, and they now list thousands of markets across sports, politics, weather, economic indicators. Polymarket runs on Polygon. In 2025 it processed $22.88 billion in volume. That much money flowing through two platforms with two different trader populations, two different fee schedules, two entirely different settlement mechanisms. Of course prices disagree. And disagreement is just arbitrage waiting for someone to notice.

Saguillo et al. (2025) estimated roughly $40 million in realized cross-platform arb profits between April 2024 and April 2025, concentrated around high-attention events like presidential debates and NFL playoffs. That's captured profit. The total spread, counting opportunities that expired because of latency or capital constraints or because nobody was looking, is larger. How much larger? Nobody knows. I wanted to find out, or at least build the tooling to find out.

Three separate skills are involved, and they share almost nothing.

NLP matching. Kalshi encodes events as structured tickers: `KXNFLGAME-25OCT12-KCBUF`. Polymarket uses free text: "Will the Kansas City Chiefs beat the Buffalo Bills on October 12, 2025?"

You need to know that `KC` means Kansas City, that `25OCT12` is October 12, 2025, that `NFLGAME` implies a binary win/loss outcome. Politics and econ are worse. Kalshi says "GDP Q3 2025 above 2.5%" and Polymarket phrases it "Will US GDP growth exceed 2.5% in 2025 Q3?" Same event. Zero surface overlap.

Fee-adjusted pricing. Both platforms charge taker fees. Different formulas. Kalshi uses a nonlinear formula with a ceiling function. Polymarket's is continuous. A spread that looks profitable before fees can turn negative after them. Can an LLM reliably compute $\lceil 0.07 \times 42 \rceil$? I had doubts.

Execution under uncertainty. This part worried me from the start. Prediction market arb is non-atomic. You fill one leg on Kalshi, then go fill the other on Polymarket. Between those two fills the price on the second platform can move. If the NO price jumps by more than your original edge during that gap, your "arbitrage" is a loss. The clock does not care about your spreadsheet.

Matching is a language problem. Fee calculation is a math problem. Execution is decision-making under sequential uncertainty. I chose this decomposition because each subtask is independently measurable and the outcome is in dollars. It's a clean testbed for LLM agents, cleaner than most benchmarks, because real money is the metric.

The question I'm after is narrow. Not "can LLMs do arbitrage." Where in the pipeline do they add value, and where do they break? I expected them to be good at matching, since matching is fundamentally about language understanding, and bad at fee calculation, since that is fundamentally arithmetic. I built the rule-based agent as a sanity check. It became the main result.

Four contributions:

1. A formal treatment of cross-platform prediction market microstructure: fee models for Kalshi and Polymarket, three arbitrage types, and the matching problem framed as entity resolution.

2. A hybrid NLP pipeline for cross-platform contract matching, evaluated on a synthetic benchmark with per-category precision/recall.

3. A synthetic benchmark of 5,000 cross-platform market pairs for development and unit testing.

4. A framework for LLM agent evaluation (task decomposition, prompt design, rule-based baselines, metrics) designed but not yet tested with real LLM API calls on live data.

Roadmap. Section 2 covers market microstructure, fees, and a formal treatment of arbitrage types. Section 3 describes the matching pipeline and the order book model. Section 4 is the LLM agent design and the rule-based baselines. Section 5 reports synthetic benchmark results. Section 6 is a status report: what's verified, what's preliminary, what I haven't done. Section 7 is discussion.

2 Background and Market Microstructure

2.1 Binary Contracts

Definition 2.1 (Binary Contract)**.** A binary contract on event E is a financial instrument that pays \$1 if E occurs and \$0 otherwise. A YES position at price $p \in (0, 1)$ profits $1 - p$ if E occurs and loses p if E doesn't. A NO position at price q profits $1 - q$ if E doesn't occur and loses q if it does.

Both platforms trade binary contracts. That is roughly where the similarity ends.

Kalshi is CFTC-regulated, based in New York. A designated contract market. Tickers follow a structured convention: `KXNFLGAME-25OCT12-KCBUF` encodes category (NFL game), date (October

12, 2025), and teams (Kansas City vs. Buffalo). REST and WebSocket APIs, authenticated via RSA-PSS signatures. Order books only report YES-side bids and asks. NO-side prices you infer.

Polymarket settles on Polygon. Events get condition IDs through the UMA oracle system. Three API layers: Gamma for market metadata, CLOB for order book data, Data for historical trades. Authentication is EIP-712 typed signatures. Order books report YES and NO tokens as independent tradeable assets.

A structural difference that matters more than it sounds: Kalshi only publishes bid data for the YES side. If the best YES bid sits at p cents, the implied NO ask is $100 - p$ cents. Polymarket reports YES and NO as separate tokens with their own books. Nothing forces those two prices to sum to exactly \$1 at any given instant. Which creates same-platform arb opportunities all by itself. More on that in Section 2.3.

2.2 Fee Models

Fees eat arbitrage. A fraction-of-a-cent error in your fee model turns a profitable trade into a losing one. The two platforms don't even use similar formulas, which I found irritating.

Definition 2.2 (Kalshi Taker Fee)**.** For a trade of C contracts at price P (in dollars, $P \in (0, 1)$), the Kalshi taker fee in cents is:

$$f_K(C, P) = \lceil 0.07 \cdot C \cdot P(1 - P) \rceil \text{ cents.} \tag{1}$$

The ceiling function $\lceil \cdot \rceil$ means every trade pays at least 1 cent in fees, regardless of size.

Definition 2.3 (Polymarket Taker Fee)**.** For a trade of Q tokens at price P, the Polymarket taker fee in USDC is:

$$f_P(Q, P) = \gamma \cdot Q \cdot P(1 - P), \tag{2}$$

where γ is a category-dependent fee rate, approximately 0.02 for sports markets.

Both share the $P(1 - P)$ kernel. But that ceiling function on Kalshi's side introduces a discontinuity that punishes small trades.

Proposition 2.4. *Both fee functions are maximized at $P = 0.5$ and vanish as $P \to 0$ or $P \to 1$.*

Proof. The function $g(P) = P(1 - P)$ has derivative $g'(P) = 1 - 2P$, which equals zero at $P = 1/2$. Since $g''(P) = -2 < 0$, the point $P = 1/2$ is a maximum. At the boundaries, $g(0) = g(1) = 0$. Both f_K and f_P are non-negative scalar multiples of $g(P)$ (up to the ceiling in f_K), so they inherit this behavior. □

Remark 2.5. Fees are fattest right where arbitrage lives. Mid-probability events near $P = 0.5$ attract the most disagreement between platforms. Diverse opinions. Heavy trading. Big price gaps. But those same events charge the most in fees. The market structure protects market makers from people like me. I started calling this the fee moat.

Concrete numbers. Buy 100 contracts at $P = 0.50$ on Kalshi: $\lceil 0.07 \times 100 \times 0.25 \rceil = \lceil 1.75 \rceil = 2$ cents. Same trade on Polymarket with $\gamma = 0.02$: $0.02 \times 100 \times 0.25 = 0.50$ USDC. Kalshi's fee is four times bigger at the midpoint. At $P = 0.10$, Kalshi charges $\lceil 0.07 \times 100 \times 0.09 \rceil = \lceil 0.63 \rceil = 1$ cent, Polymarket charges $0.02 \times 100 \times 0.09 = 0.18$ USDC. The ratio narrows at extreme prices, but Kalshi is always at least as expensive because the ceiling rounds up.

2.3 Arbitrage Types

Three types. Different structure, different risk profile, different detection difficulty.

Definition 2.6 (Direct Cross-Platform Arbitrage)**.** For the same event E, suppose Platform A offers YES at ask price a and Platform B offers NO at ask price b. The gross edge per contract is:

$$e_{\text{gross}} = 1 - a - b. \tag{3}$$

The net edge after fees for n contracts is:

$$e_{\text{net}}(n) = 1 - a - b - \frac{f_A(n, a)}{n} - \frac{f_B(n, b)}{n}. \tag{4}$$

An arbitrage opportunity exists when $e_{\text{net}}(n) > 0$ for some $n \geq 1$.

Definition 2.7 (Implied Same-Platform Arbitrage)**.** On a single platform, if the best YES ask p_{yes} and best NO ask p_{no} satisfy

$$p_{\text{yes}} + p_{\text{no}} < 1 - \frac{f(n, p_{\text{yes}}) + f(n, p_{\text{no}})}{n} \tag{5}$$

for some n, then buying both YES and NO guarantees a profit.

Same-platform arb is rare on Kalshi. Their centralized exchange mechanically enforces $p_{\text{yes}}+p_{\text{no}} \geq 1$. On Polymarket it happens. YES and NO tokens trade independently. During volatile news events the two sides decouple. I've watched it happen during earnings releases where the spread blew out for ten or fifteen seconds.

Definition 2.8 (Monotonicity Violation)**.** For a set of threshold markets on the same underlying variable X with thresholds $T_1 < T_2 < \cdots < T_k$, the prices must satisfy:

$$\mathbb{P}(X > T_1) \geq \mathbb{P}(X > T_2) \geq \cdots \geq \mathbb{P}(X > T_k). \tag{6}$$

A violation $\mathbb{P}(X > T_i) < \mathbb{P}(X > T_{i+1})$ for some i signals mispricing.

These show up in economic threshold markets. Kalshi lists five markets on Q3 GDP growth: above 1%, above 1.5%, above 2%, above 2.5%, above 3%. If "above 2.5%" is trading at 45 cents while "above 2%" sits at 42 cents, that violates monotonicity. The less restrictive threshold has to be priced weakly higher. These violations tend to be small, 1 to 3 cents, and they close fast.

Remark 2.9 (Non-Atomic Settlement Risk). This is not futures arb where both legs fill simultaneously on one exchange. The legs execute sequentially. Buy YES on Kalshi. Go buy NO on Polymarket. Between those fills, call it δ seconds, the Polymarket price can move. If the NO price jumps by more than the original edge during that gap, the "arbitrage" becomes a loss. Real profits run smaller than theoretical edge for this reason. Always.

2.4 The Matching Problem

None of the arb detection matters if I can't figure out which Kalshi market corresponds to which Polymarket market. This is entity resolution.

Formally: let $\mathcal{K} = \{k_1, k_2, \ldots, k_m\}$ be the set of active Kalshi tickers and $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ the set of active Polymarket events. I want all pairs $(k_i, p_j) \in \mathcal{K} \times \mathcal{P}$ that refer to the same real-world event.

The mapping is not one-to-one. A single Polymarket event ("Will the Chiefs beat the Bills?") might correspond to multiple Kalshi tickers if Kalshi lists both game outcome and point spread. Going the other direction, a Kalshi ticker for "Will inflation exceed 3% in 2025?" might have no Polymarket counterpart at all.

What makes matching hard:

- **Naming conventions.** Kalshi uses structured abbreviations (`KC` for Kansas City, `BUF` for Buffalo). Polymarket spells things out, sometimes with articles ("the Kansas City Chiefs").
- **Temporal encoding.** Kalshi does `25OCT12` (YYMMMDD). Polymarket does whatever: "October 12, 2025" or "Oct 12" or "10/12/25."
- **Granularity.** Kalshi lists 10 threshold levels for GDP. Polymarket lists 5 different ones. You find the overlap or you don't.
- **Ambiguity.** Political events are the worst. "Will Biden comment on the debate?" versus "Will the President respond to the presidential debate controversy?" Same event? Maybe. Depends on context and timing. I genuinely don't know sometimes.

Example 2.10. The ticker `KXNFLGAME-25OCT12-KCBUF` decomposes as:

- `KX`: Kalshi exchange prefix.
- `NFLGAME`: category (NFL game outcome).
- `25OCT12`: date (October 12, 2025).
- `KC`: Kansas City Chiefs.
- `BUF`: Buffalo Bills.

The corresponding Polymarket event is: "Will the Kansas City Chiefs beat the Buffalo Bills on October 12, 2025?" Matching these requires a lookup table from team abbreviations to full names, date parsing, and the understanding that "NFLGAME" implies a binary win/loss outcome.

Miss a match and you miss money. False-match two unrelated events and you trade them against each other. That is not arbitrage. That is gambling with extra steps.

3 System Architecture

3.1 Hybrid Matching Pipeline

The matcher runs candidates through four stages. Easy matches die early. Hard ones propagate to more expensive layers. I designed it this way because LLM calls cost money and I don't have much of it.

Stage 1: Rule-Based Extraction. Regex patterns rip Kalshi tickers apart into structured fields: category, teams/entities, date, threshold if applicable. Polymarket descriptions get keyword extraction with named entity recognition pulling out teams, dates, numerical thresholds. Both sides produce a canonical tuple (c, e_1, e_2, d, t) where c is category, e_1, e_2 are entities, d is date, t is threshold. Exact field matches yield high-confidence pairs immediately. On synthetic data about 60% of true matches get caught here. Mostly sports, where both platforms name things in structured ways.

Stage 2: Sentence-BERT Embedding. Whatever's left unmatched gets normalized. Kalshi tickers have their abbreviations expanded from a lookup table. Polymarket descriptions get lowercased with articles removed. Both pass through `all-MiniLM-L6-v2` (Reimers and Gurevych, 2019). Cosine similarity above 0.85 counts as a match. This catches cases where field extraction choked: weird abbreviations, non-standard date formats, events that refuse to decompose cleanly into (c, e_1, e_2, d, t).

Stage 3: Fuzzy String Fallback. Residual candidates, similarity between 0.60 and 0.85, get processed with rapidfuzz (Bachmann, 2021) token-set-ratio. Handles typos, minor rewording, cases where the embedding model gave moderate similarity to pairs that genuinely match. Threshold: 80% token-set-ratio.

Stage 4: LLM Verification. The ambiguous zone. Candidates with token-set-ratio between 80% and 95%, or SBERT similarity between 0.70 and 0.90. These get shipped to an LLM. The prompt includes both descriptions and asks for binary match/no-match plus a one-sentence explanation. This stage costs roughly 50× the latency of Stage 2, so it only sees the tail. On synthetic data about 8% of candidate pairs reach it.

Each stage rejects obvious non-matches before the next one fires. The LLM only touches the stuff that cheaper methods couldn't resolve.

3.2 Order Book Model

Each side of each market is an L2 order book: sorted arrays of (price, volume) tuples for bids and asks.

$$\text{Book}_{\text{ask}} = \{(p_1, v_1), (p_2, v_2), \ldots, (p_L, v_L)\} \quad \text{with } p_1 < p_2 < \cdots < p_L, \tag{7}$$

where L is the number of price levels. Each level is a standing limit order. To fill n contracts you walk up the book: eat v_1 contracts at p_1, then v_2 at p_2, and so on until filled or the book is empty.

Effective fill price for n contracts is volume-weighted:

$$\bar{p}(n) = \frac{\sum_{i=1}^{k} p_i \cdot \min(v_i, r_i)}{\sum_{i=1}^{k} \min(v_i, r_i)}, \tag{8}$$

where $r_1 = n$ and $r_{i+1} = r_i - \min(v_i, r_i)$, with k being the first level where $r_k \leq v_k$.

Cross-platform synchronization is a headache I haven't solved. Kalshi's WebSocket updates arrive every 100–500ms. Polymarket's CLOB can lag 1–3 seconds during heavy volume. A book older than 30 seconds is garbage for arb purposes. Prices move 2–3 cents in that window.

3.3 Fee-Adjusted Edge Calculation

Net edge for a cross-platform trade of n contracts, buying YES on Platform A at effective price $\bar{a}(n)$ and NO on Platform B at effective price $\bar{b}(n)$:

$$e_{\text{net}}(n) = 1 - \bar{a}(n) - \bar{b}(n) - \frac{f_A(n, \bar{a}(n))}{n} - \frac{f_B(n, \bar{b}(n))}{n}. \tag{9}$$

This is concave in n for two separate reasons. First, $\bar{a}(n)$ and $\bar{b}(n)$ are weakly increasing: walking up the book means worse prices. Second, Kalshi's ceiling function makes the per-contract fee $f_K(n, P)/n$ weakly decreasing, because the 1-cent overhead from the ceiling spreads over more contracts. Tension: bigger size improves fee efficiency but worsens fill prices.

Optimal position size n^* satisfies:

$$n^* = \arg\max_{n \in \{1,\ldots,N_{\max}\}} n \cdot e_{\text{net}}(n), \tag{10}$$

where $N_{\max}$ is the minimum depth available across both books. I don't optimize over every possible size. I compute $e_{\text{net}}(n)$ for $n \in \{1, 10, 25, 50, 100\}$ and pick the winner. The surface is smooth enough that this five-point grid works. I spent an embarrassing amount of time writing a continuous optimizer before realizing five grid points gave the same answer every single time.

Minimum edge threshold: 2%. Below that, skip. That 2% covers execution risk (the non-atomic gap from Remark 2.9), slippage from stale books, and the chance that the matcher handed me a false positive.

4 LLM Agent Design

4.1 Task Decomposition

I split the pipeline into three tasks. Each one testable on its own.

Definition 4.1 (Task T_1: Match Verification)**.** Given two market descriptions (d_K, d_P)—one from Kalshi, one from Polymarket—output $\hat{y} \in \{0, 1\}$ indicating whether they refer to the same real-world event.

Definition 4.2 (Task T_2: Trade Decision)**.** Given an arbitrage opportunity $o = (\text{match pair}, \text{order books}, \text{edge}, \text{fees})$ and current portfolio state $s = (\text{positions}, \text{capital}, \text{daily PnL})$, output a decision $\hat{d} \in \{\text{trade}, \text{skip}\}$ and, if trading, a position size $\hat{n}$.

Definition 4.3 (Task T_3: Risk Assessment)**.** Given portfolio state s and a set of pending opportunities $\{o_1, \ldots, o_k\}$, output risk metrics: estimated portfolio variance, maximum loss scenario, and a recommendation to continue trading or pause.

T_1 is language. T_2 is quantitative. T_3 is monitoring that gates the other two. In a fully autonomous agent all three run continuously; my evaluation framework tests them in isolation first, then together.

4.2 Agent Architecture

For each task the LLM gets a structured prompt with everything it needs. Full prompts are in the code repo. Here I describe the design choices that turned out to matter.

T_1: the prompt shows the Kalshi ticker expanded into natural language (via the abbreviation lookup table) next to the Polymarket description. "Do these two descriptions refer to the same event? Answer YES or NO. Then explain your reasoning in one sentence." I log the explanation but don't use it algorithmically. Maybe I should. I tried parsing the explanations once and gave up.

T_2: the prompt includes matched pair descriptions, both order books (top 5 levels), computed gross and net edge, fee breakdown, current positions, available capital, running daily PnL. The LLM outputs JSON: {``action'': ``trade''/``skip'', ``size'': int, ``reason'': string}. The reason field exists for interpretability, mostly so I can debug at midnight.

Here is the test embedded in T_2. Can the LLM correctly verify the fee computation? The prompt gives raw inputs and asks the agent to confirm or reject the precomputed fee:

> *Kalshi fee for 100 contracts at $P = 0.47$: computed as $\lceil 0.07 \times 100 \times 0.47 \times 0.53 \rceil = \lceil 1.7402 \rceil = 2$ cents. Is this correct?*

Yes, it is correct. I expected this to be where LLMs fall apart. Multi-step floating-point multiplication into a discrete ceiling function. Approximate reasoning breaks down on exactly this kind of thing. But I don't have real error rates yet. Section 6.3 will produce those.

T_3: the prompt presents everything. All open positions, entry prices, current market prices, total exposure. The LLM estimates worst-case loss (every position goes to zero on the losing side) and recommends whether to keep trading or stop. Probably the easiest of the three tasks for an LLM since it is mostly qualitative reasoning about concentration.

4.3 Rule-Based Baseline

No LLM at all. Pure code. Fast.

Matching. Stages 1–3 of the hybrid pipeline (Section 3.1) with tuned thresholds: SBERT similarity ≥ 0.85, rapidfuzz token-set-ratio $\geq 80\%$. Stage 4 doesn't run.

Trading. If $e_{\text{net}}(n^*) > 0.02$ and depth is at least 50 contracts at best price on both sides, execute. Position sizing is fractional Kelly:

$$n = \min\left(\left\lfloor 0.25 \cdot \frac{e_{\text{net}}}{\sigma^2} \right\rfloor,\ N_{\max}\right), \tag{11}$$

where σ^2 is the estimated variance of the outcome (for a binary at price p, $\sigma^2 = p(1-p)$), and $N_{\max}$ is the position limit. The 0.25 multiplier is quarter-Kelly, standard practice when you're trading variance against expected growth. Limits: \$100 per market, \$10,000 total exposure.

Risk. Hard stop: if daily PnL drops below -5% of capital, stop trading. Max 30% of capital in any single category. Dumb rules. Microsecond execution. That is the point.

4.4 Evaluation Metrics

Matching: precision (P), recall (R), F1 ($F1 = 2PR/(P+R)$). Globally and per category.

Trading: portfolio metrics.

- Cumulative PnL: total dollars earned net of all fees.
- Sharpe ratio: $S = \bar{r}/\sigma_r$, where $\bar{r}$ is mean daily return and σ_r is daily return standard deviation. Annualized by multiplying by $\sqrt{252}$.
- Maximum drawdown: largest peak-to-trough decline in cumulative PnL.
- Trade count and error count.

Decision quality is the fraction of decisions that agree with the ex-post optimum:

$$\text{DQ} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\left[\text{sign}(\hat{d}_i) = \text{sign}(d_i^*)\right], \tag{12}$$

where $\hat{d}_i$ is the agent's decision and d_i^* is the decision that would have been optimal given realized outcomes. Noisy metric. Some right decisions are just unlucky. I report it alongside PnL, which integrates both decision quality and sizing accuracy.

Status note. The evaluation framework is implemented. Prompts written, rule-based baseline runs, metrics compute correctly on synthetic data. What's missing: real LLM API calls (GPT-4, Claude, etc.) on live market data. Section 6.3 has the plan.

5 Synthetic Benchmark

I built a synthetic benchmark for development and unit testing. This section describes the data generation and reports matching results. These numbers tell me whether the pipeline's internals work, whether the stages compose properly. They do not tell me how the pipeline handles real Kalshi and Polymarket markets, where naming is messier and the difficulty distribution is different. Real-data evaluation is in Section 6.2 and Section 6.3.

5.1 Synthetic Data Generation

Two reasons to start with synthetic data. Reproducibility: live markets change second by second, so experiments aren't repeatable. Control: I can independently vary matching difficulty, arb edge size, and order book liquidity.

5,000 market pairs. Category breakdown: 50% sports (2,500 pairs), 20% crypto (1,000), 15% politics (750), 15% economics (750). This roughly tracks the volume distribution on both platforms. I considered making it 70% sports to match actual listing counts, but wanted enough political and econ pairs to get meaningful recall numbers in those categories.

For each true match I generate:

- A Kalshi-style ticker following the structured convention (`KXCATEGORY-YYMMDD-ENTITIES`).
- A Polymarket-style description in natural language.
- Two order books, 10 price levels each. Spreads drawn uniformly from 1–5 cents. Depth at each level from a log-normal distribution with mean 50 contracts.

Negative pairs come from corrupting one field of a true match: swap teams ("Chiefs vs Bills" → "Chiefs vs Ravens"), shift the date by a day, change the threshold ("above 2.5%" → "above 3.0%"), or replace the category entirely. Hard negatives, on purpose. Most fields match, but one critical field differs. About 40% of the 5,000 pairs are true matches; 60% are negatives, half hard and half easy (completely different events).

5.2 Matching Results on Synthetic Data

Five matching configurations:

1. **Regex only**: Stage 1 alone.
2. **Regex + SBERT**: Stages 1–2.
3. **Regex + SBERT + Fuzzy**: Stages 1–3 (full pipeline, no LLM).
4. **Full pipeline**: All four stages (mock LLM verifier applying string-similarity heuristics to simulate Stage 4 decisions).
5. **Mock LLM only**: Skip all rule-based stages; the mock verifier processes every pair.

Caveat. Stage 4 here is a mock LLM. Not real GPT-4. Not real Claude. It is a string-similarity heuristic pretending to be an LLM. The numbers below test pipeline architecture and Stages 1–3. They say nothing about real LLM performance on Stage 4. Real LLM evaluation is in Section 6.3.

The first three rows matter. The hybrid pipeline without any LLM hits 93.9% F1 on synthetic data. Precision and recall balanced. Each stage earns its keep. Regex alone: 79.8% F1, catching

Table 1: Cross-platform matching results on the **synthetic** benchmark (5,000 market pairs). Stage 4 uses a mock LLM verifier, not real LLM API calls. These numbers test pipeline logic, not real-world matching performance.

Method	Precision	Recall	F1
Regex only	0.891	0.723	0.798
Regex + SBERT	0.931	0.912	0.921
Regex + SBERT + Fuzzy	0.931	0.947	0.939
Full pipeline (+mock LLM)	0.978	0.943	0.960
Mock LLM only	0.854	0.912	0.882

the structured easy stuff. SBERT jumps it to 92.1%, a 12.3-point gain, handling cases where field extraction choked. Fuzzy matching adds 1.8 points on the remaining near-misses.

The last two rows are about the mock verifier. As Stage 4 it boosts precision from 93.1% to 97.8%. Used alone it is mediocre, 88.2% F1, because it lacks the structured extraction the other stages provide. Good for validating the cascade architecture. Not informative about real LLM performance.

Category-level results on synthetic data: sports easiest (structured abbreviations map cleanly), politics hardest (paraphrase, ambiguity), crypto and econ in between. But I suspect synthetic data understates the difficulty of real political events. Real Polymarket political descriptions are wilder and more context-dependent than anything my generator produces.

5.3 Ablation: Pipeline Stage Contribution

I measured each stage's contribution along three axes: F1 improvement, latency cost, and how many candidate pairs it processes.

The cascade filters hard. Stage 1 resolves about 60% of true matches. Stage 2 catches another 30%. Stage 3 handles about 8%. Only the remaining 2% would reach Stage 4. Latency is correspondingly skewed: regex runs in microseconds, SBERT encoding is 2–5ms per pair, fuzzy matching is comparable, and LLM verification with a real model would average around 800ms per pair. The cascade makes the expensive stage practical. You can not afford 800ms per pair on everything. You can afford it on 2%.

6 Results and Status

I want to be blunt about what is done and what is not. If you skim one section of this paper, skim this one.

6.1 Verified Components

These components are implemented, tested, and correct:

Fee models. The Kalshi formula (Definition 2.2) and Polymarket formula (Definition 2.3) are implemented and verified against platform API docs and actual fee schedules. The ceiling-function behavior produces correct outputs for all tested inputs, including edge cases at extreme prices (P near 0 or 1) and tiny trade sizes ($C = 1$) where the ceiling dominates.

Arbitrage definitions. All three types (cross-platform, implied same-platform, monotonicity violations) are correctly formalized and implemented as detection functions. Given correct prices,

they find the right opportunities. Net edge calculation (Eq. 9) handles both fee models and book depth.

Matching pipeline (Stages 1–3). Rule-based extraction, SBERT embedding, fuzzy string matching. All run correctly. 93.9% F1 on synthetic data (Table 1). The seven Python files all run without errors, which I am probably more proud of than I should be.

Synthetic benchmark. 5,000 pairs, correct category distribution, correct difficulty profile. Does its job: development, debugging, unit testing.

Mathematical framework. Fee model analysis (Proposition 2.4), the fee moat observation (Remark 2.5), order book model, position sizing. Correct descriptions of real systems, derived from platform specs. No empirical validation needed.

6.2 Preliminary Results: Live Market Data

I built a data fetcher that hits the Kalshi API (`api.elections.kalshi.com/trade-api/v2`) and the Polymarket Gamma API (`gamma-api.polymarket.com`), normalizes the responses, and runs the three-stage matching pipeline. Both endpoints are public, no auth required for read access. I ran this on May 25, 2026. Here's what came back.

Market inventory. Kalshi: 813 active markets. Polymarket: 46 active markets (filtered to sports and crypto with active order books). The asymmetry is itself a finding. Polymarket lists far fewer markets at any given time.

Kalshi's structural shift. This is the thing I didn't expect. Of 813 Kalshi markets, 782—that's 96.2%—were multi-leg parlay contracts under the `KXMVESPORTSMULTIGAMEEXTENDED` ticker family. These are compound bets: "OKC wins by over 3.5 points AND over 219.5 points scored." Only 31 markets (3.8%) were simple binary outcomes. My entire system architecture assumes simple binary matching—one Kalshi event maps to one Polymarket event—but Kalshi's current inventory is almost entirely parlays with no single-event Polymarket equivalent. I stared at my terminal for a while after discovering this.

Matching results. 14 candidate matches, all sports. The highest confidence was 0.564: an OKC Thunder parlay matched to "Will the Oklahoma City Thunder win the NBA Western Conference Finals?" All 14 were structurally wrong—multi-leg Kalshi parlays paired with single-outcome Polymarket markets. The pipeline did assign low confidence scores (none above 0.57), but didn't reject them outright because the team name overlap creates real semantic similarity. "Oklahoma City" is "Oklahoma City" whether it's in a parlay or not.

Order book analysis. Most of the 14 candidates had empty or illiquid Kalshi books—best bid = 0, best ask = 1.00. One showed a Kalshi ask of \$0.062 against a Polymarket bid of \$0.62. Nominal gross edge: 55.8 cents. Incredible, right? No. The Kalshi contract is a multi-leg parlay (OKC wins by 2.5+ AND total over 219.5 AND total over 230.5). The Polymarket contract is a simple series-winner bet. They're not the same event. The price difference reflects the lower probability of the compound outcome, not any kind of market inefficiency. I almost got excited. I shouldn't have.

Key finding. Live data reveals a structural mismatch between current platform inventories that my synthetic benchmark completely missed. Cross-platform arbitrage needs matching pairs of simple binary contracts. Kalshi's shift to multi-leg parlays means most of its volume is in contracts with no Polymarket equivalent. The opportunity set might be smaller than Saguillo et al. (2025)'s \$40 million figure suggests—that covered a period when both platforms listed more single-outcome events.

This doesn't kill the pipeline or the LLM evaluation framework. But it changes what deployment looks like. A production system would need a parlay decomposition stage: parse multi-leg Kalshi contracts into individual components, match each one separately. Haven't built that yet.

6.3 Planned Experiments

Five experiments left, roughly in priority order:

1. **Hand-labeled matching benchmark on live markets ($N \geq 200$ pairs).** Fetch active markets from both platforms, hand-label matches and non-matches across categories, evaluate all five pipeline configurations on real data. This is the single most important missing piece. Synthetic performance does not transfer to real naming patterns.

2. **LLM agent evaluation with real API calls.** Run GPT-4 and Claude on all three tasks (T_1: match verification, T_2: trade decision, T_3: risk assessment) using real market data. Measure the actual fee calculation error rate. I think LLMs will botch the Kalshi ceiling function but I don't have a number yet.

3. **Paper trading simulation with live order book data.** Run both agents, rule-based and LLM, on a multi-week simulation using real order book snapshots. Requires solving cross-platform synchronization (matching timestamps between Kalshi WebSocket data and Polymarket CLOB snapshots). Not trivial.

4. **Fee calculation error rate measurement.** Present real fee problems (actual prices and quantities from live markets) to multiple LLMs. Measure arithmetic error rate, ceiling omission rate, unit confusion rate. A focused slice of item 2 that can run independently.

5. **Full backtest with historical data.** If I can get historical order book data (Kalshi has some through their data API; Polymarket requires scraping), run a retrospective simulation over 3–6 months. Estimate PnL, Sharpe, drawdown for both agent types.

Until these experiments run, what I can claim is limited. Pipeline architecture is sound. Fee math is correct. Matching works on synthetic data. Whether the system finds real arb, and whether LLMs help or hurt at each stage, I don't know yet.

7 Discussion

7.1 What the Synthetic Results Tell Us

The synthetic benchmark validates architecture. Each stage earns its place: regex handles structured easy cases, SBERT catches semantic equivalences regex can't, fuzzy matching picks up near-misses. The cascade filters aggressively enough that the LLM stage, once real, will only see a small fraction of pairs. You don't want to pay 800ms per pair when regex could have answered in microseconds.

93.9% F1 without an LLM. Sounds nice. I am skeptical of it. There is some circularity. Synthetic data is generated by rules, and the pipeline is rule-based. I designed the hard negatives by corrupting single fields, which is exactly the kind of difficulty that field-matching is built to detect. Real hard negatives might be hard in ways I didn't think of. Different categories sharing entity names. Two different games between the same teams on different dates. A regular season matchup vs. a playoff rematch. That kind of thing.

I expect performance to drop on live data, especially on politics and economics markets where Polymarket descriptions are more varied than my synthetic templates. Sports should hold up. NFL teams don't rebrand between game weeks.

7.2 Expected LLM Behavior

I have a hypothesis and no data. Let me state it so the experiments can kill it.

Matching (T_1): LLMs should add value. Especially on politics and economics where paraphrase and implicit references are common. "Will the former president attend the debate?" matching "Trump debate attendance" requires world knowledge the rule-based pipeline does not have. The gain should be largest on the hard tail.

Fee calculation (T_2): LLMs should fail. The Kalshi formula is multi-step floating-point multiplication followed by a ceiling function. The literature on LLM arithmetic (Dziri et al., 2023; Qian et al., 2022) says errors compound in exactly this setting. The ceiling adds a discrete cliff to a smooth computation. I predict 5–15% error rates on Kalshi fee computation, below 3% on Polymarket's simpler formula. These are guesses. I want to be wrong about the upper end. I don't think I will be.

State tracking (T_3): LLMs should handle qualitative risk assessment okay. Spotting concentration, flagging overexposure. But I expect them to lose track of precise capital across sequential decisions because each prompt call is stateless. Portfolio state is just text in a prompt. The rule-based agent tracks capital with a running counter. No contest.

My headline prediction: LLMs help at the NLP layer, hurt at the quantitative layer. Net contribution to PnL depends on the ratio of language to math in the pipeline. This pipeline has more math than language. That probably means net negative.

7.3 Design Implications

Even without real results the design points somewhere specific. LLMs for matching, extracting information from news, interpreting ambiguous settlement rules. Deterministic code for everything with numbers: fee computation, edge calculation, position sizing, risk limits, order book parsing.

This lines up with the tool-use paradigm from Schick et al. (2023): LLMs reason and plan, tools compute. Here the "tools" are the fee calculator, the order book parser, the Kelly sizing formula. The LLM's job is deciding whether two events match and interpreting qualitative information like news, settlement rules, and regulatory changes. Everything that fits in a formula should be a formula.

A concrete architecture:

1. **Ingestion layer**: rule-based. Parse API responses, build order books, compute derived quantities.
2. **Matching layer**: hybrid. Regex + SBERT + fuzzy for the easy 92% of pairs, LLM for the 8% ambiguous tail.
3. **Detection layer**: rule-based. Fee computation, edge calculation, opportunity scoring.
4. **Execution layer**: rule-based. Kelly sizing, position limits, stop-losses.
5. **Monitoring layer**: LLM. Summarize daily performance, flag anomalies, interpret news affecting open positions.

LLM at layers 2 and 5. Layers 1, 3, 4 are pure code. More complex than "just use an LLM for everything," but should be more reliable and cheaper. Should be. Whether it actually is, well. That is what the experiments are for.

7.4 Limitations

I will start with the obvious one. Right now this is a design paper with synthetic validation. The theory works. The code runs. The pipeline handles fake data. But the numbers that matter, real matching accuracy, real LLM error rates, real PnL, are missing.

My synthetic data understates difficulty in specific ways. I generate order books from parametric distributions: log-normal depth, uniform spreads. Real order books have fat tails, clustering, autocorrelation. Real markets are adversarial. Market makers adapt to arb flow. They widen spreads when they detect it. They use information from one platform's book to update the other. My simulations don't capture any of that. Polite simulations. Mean markets.

I have tested against one mock LLM and zero real ones. GPT-4, Claude, Gemini all have different training distributions. Maybe one handles the ceiling function fine. I doubt it, but I don't know. Multi-model evaluation is planned.

No execution risk modeling with real data. In production you would hit rejected orders, partial fills, platform outages, API rate limits. Polymarket's CLOB has minimum order sizes and tick sizes I have not tested against. I am aware this matters. Have not gotten there yet.

And then there is the regulatory question. Kalshi is CFTC-regulated. Polymarket's primary interface operates from outside the United States, regulatory status ambiguous. Cross-platform arb between a regulated and an unregulated venue raises legal questions for anyone in the U.S. that I do not address here. The technical pipeline works regardless of jurisdiction. Deploying it is a different conversation, and not one I am qualified to have.

Acknowledgments

Thanks to the prediction markets crowd on Twitter who put fee structures, API docs, and arb strategies out in public. That open discussion of market microstructure on Kalshi and Polymarket is the reason this project exists. Also thanks to the USC Viterbi math department for not asking too many questions about why I was refreshing order books during office hours.

References

Max Bachmann. Rapidfuzz: Rapid fuzzy string matching. https://github.com/maxbachmann/RapidFuzz, 2021.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D Hwang, and Yejin Choi. Faith and fate: Limits of transformers on compositionality. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Jing Qian, Hong Wang, Zekun Li, Shiyang Li, and Xifeng Yan. Limitations of language models in certain reasoning tasks. *arXiv preprint arXiv:2210.06726*, 2022.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 3982–3992, 2019.

Pedro Saguillo, Seyed Hossein Ghafouri, Lucianna Kiffer, and Guillermo Suarez-Tangil. Arbitrage in prediction markets. *arXiv preprint arXiv:2508.03474*, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2023.