Insider Trading Detection on Polymarket

A Technical Course for Engineers

by Laurenz Bougan

Table of Contents

Chapter 1 — The Problem

Every trade on Polymarket leaves a trace. A wallet address, a timestamp, a position size, a resolution outcome. On its own, a single winning bet on a geopolitical event means nothing — someone has to win. But when the same cluster of addresses shows up repeatedly, buying early and heavy on events that hinge on information most of the world didn't have, the noise starts to look like a signal.

This chapter is about understanding exactly what that signal looks like before we build anything to find it.

Polymarket is a decentralized prediction market where participants buy and sell shares in binary outcomes. A share in the YES position of a market that resolves YES pays out one dollar. A share in the NO position pays nothing. The price of a share at any given moment reflects the crowd's aggregate probability estimate for that outcome. If the market prices YES at 0.30, participants collectively believe there is roughly a 30% chance the event resolves YES. Traders who believe the true probability is higher buy YES shares. Traders who believe it is lower sell them or buy NO.

The mechanics matter because they define what anomalous behavior looks like. A trader with genuine information advantage does not behave like a lucky guesser and does not behave like a skilled analyst. They behave like someone who already knows the answer. They enter positions early, before the market has had a chance to reprice around emerging information. They size those positions heavily, because the risk they perceive is lower than the risk the market is pricing. And they win at a rate that has no comfortable explanation in terms of skill or chance.

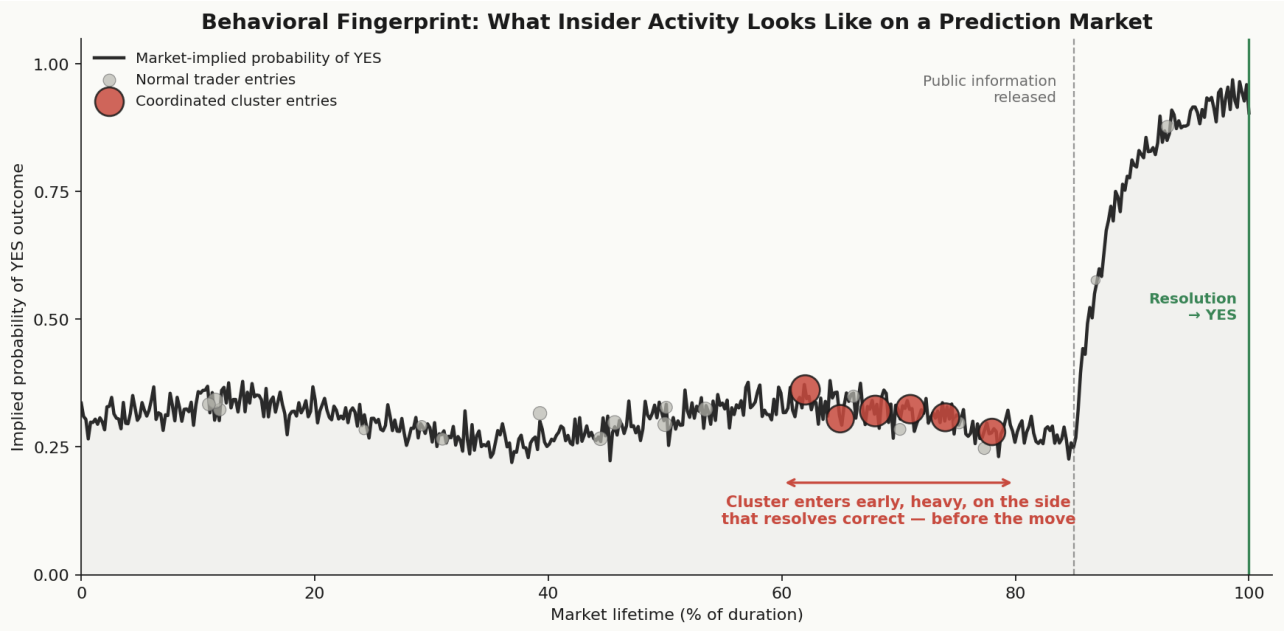
The pattern we are targeting throughout this course is specific. We are looking for accounts that have accumulated a win rate above roughly 90% across five or more geopolitical prediction markets, entered positions at prices significantly below the final resolution probability, and in many cases coordinated that activity with a small number of other wallets operating on the same events at similar times. None of those features alone is a conviction. Together, they form a profile that warrants scrutiny.

Before we go further it is worth being precise about what we mean by insider trading in this context. In traditional financial markets, insider trading has a legal definition tied to material non-public information and a fiduciary duty. Polymarket operates in a different legal environment and the term carries no formal regulatory weight here. What we mean by it is narrower and more behavioral: an actor using information that was not publicly available at the time of trade entry to take positions whose profitability depends on that informational advantage. The practical question is not legal guilt but statistical anomaly. Does this account's trading record have a plausible innocent explanation?

Geopolitical markets are the high-signal domain for this kind of abuse for several reasons. First, they resolve on discrete real-world events: an election outcome, a military action, a diplomatic decision, a leadership change. These are not markets where skill

means synthesizing public data faster than your competitors. They are markets where the informational edge is binary. You either have access to the right conversation, the right document, or the right source, or you do not. Second, geopolitical events tend to have concentrated information environments. The number of people who know the outcome of a covert military operation before it becomes public is small. The number of people who know the result of a closed-door diplomatic meeting before the press release is small. That concentration means that when an informational edge exists, it is usually traceable to a small number of actors. Third, these markets tend to have relatively thin liquidity and long resolution timelines, which means a well-timed early position can be placed at highly favorable odds before the market corrects. The asymmetry between entry price and resolution payout can be enormous.

None of this is hypothetical. There have been publicly discussed instances of Polymarket activity on geopolitical markets where position entry timing relative to news events raised obvious questions. A position entered hours before a publicly announced military action. A cluster of wallets buying heavily on an electoral outcome at long odds the day before a result that surprised most polling models. These are not proof of anything on their own. But they are exactly the kind of pattern a systematic detection pipeline should surface.

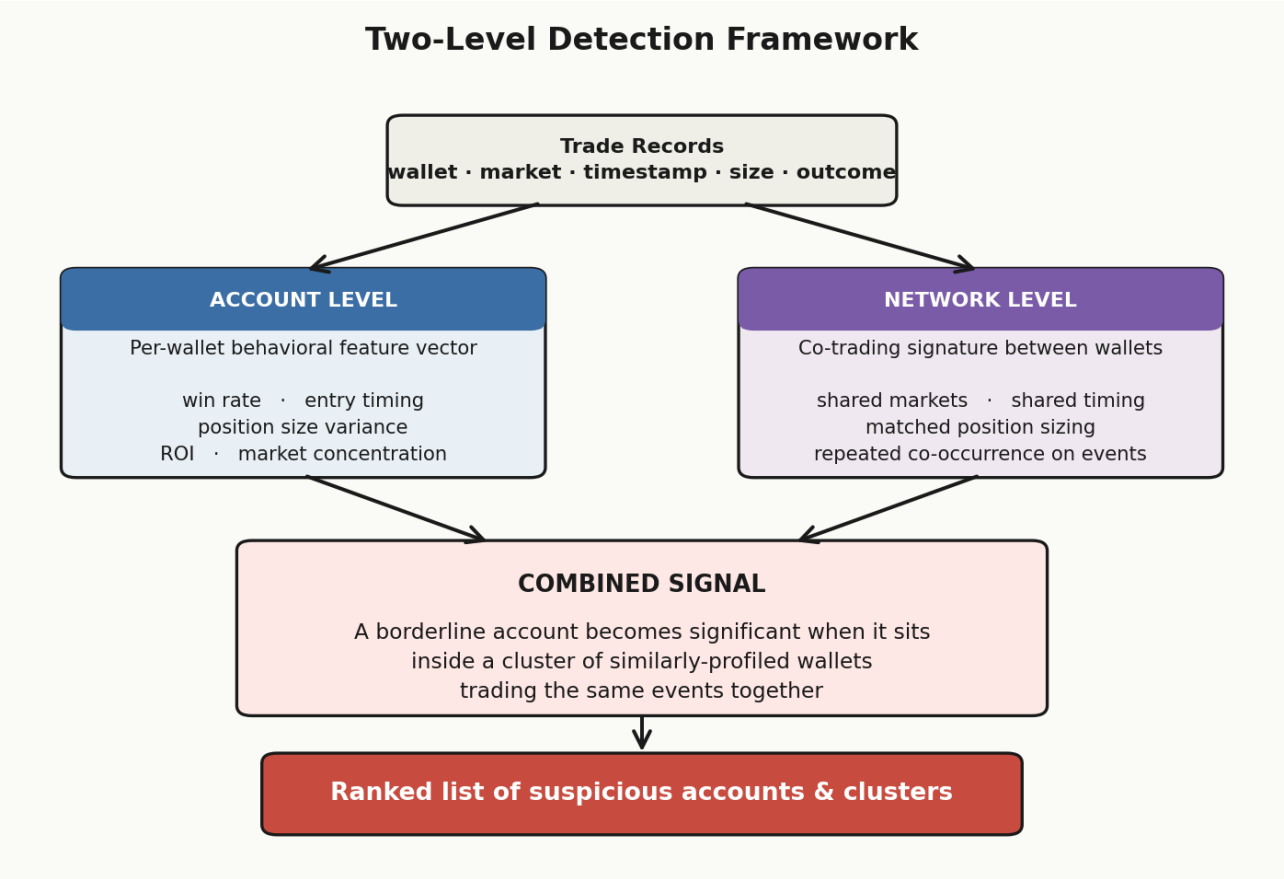


Now consider why prediction markets are structurally more vulnerable to this kind of abuse than traditional financial instruments. In a conventional financial market, regulatory infrastructure exists specifically to detect and prosecute information asymmetry. Trade surveillance systems flag unusual position changes ahead of corporate announcements. Reporting requirements create paper trails. The barrier to executing an insider trade without detection is high. On a decentralized prediction market, none of that infrastructure exists by default. Wallet addresses are pseudonymous. There is no requirement to register. Position entry and exit data is publicly available on-chain, which is actually an advantage for detection, but the enforcement mechanism does not exist. The result is that the cost of exploiting informational advantage is low and the probability

of formal consequence is close to zero. That asymmetry creates an environment where insider trading, if it occurs, is more likely to be persistent and patterned rather than isolated.

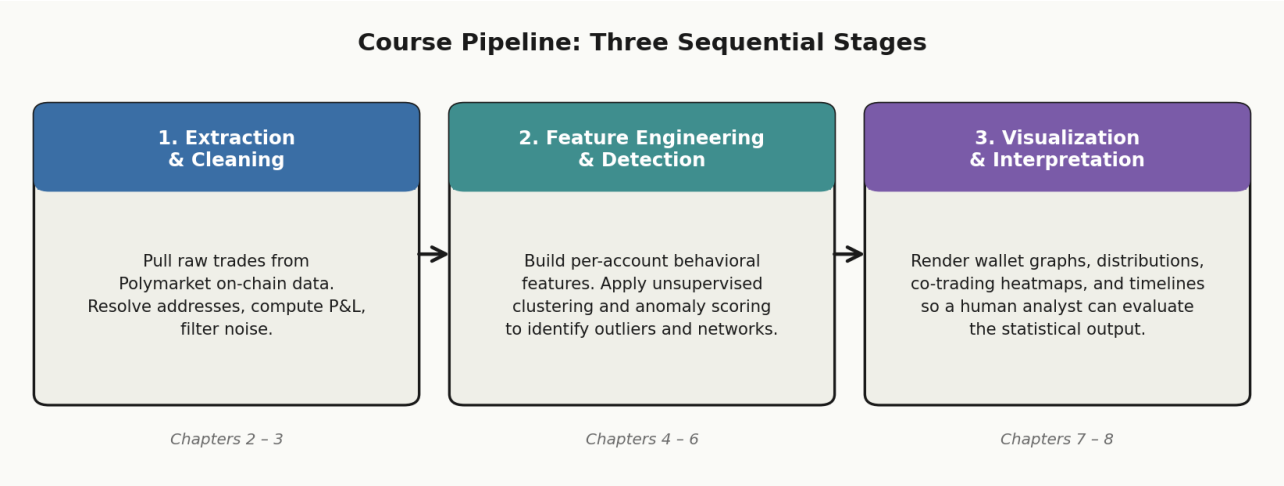
The coordinated cluster case compounds this. A single wallet with a 95% win rate across ten geopolitical markets is suspicious. A network of fifteen wallets, each with a 70% win rate across the same ten markets, entering positions within a narrow time window of each other and sizing them consistently, is more suspicious and harder to see without the right tools. The coordinated case is designed, whether consciously or not, to fragment the statistical signal across multiple identities. Each individual account looks less anomalous. The network does not.

This is why the detection framework we will build throughout this course operates at two levels simultaneously. At the account level, we build behavioral feature vectors that score individual wallets on dimensions like win rate, trade frequency on geopolitical markets, and entry timing relative to resolution. At the network level, we look for co-trading patterns, shared timing signatures, and clustering in the feature space that suggests coordination. The two levels reinforce each other. An account that is borderline suspicious on its own becomes more suspicious when it sits inside a cluster of similarly-profiled wallets that consistently traded the same events.



The framework has three sequential steps that the following chapters will build out in full.

1. **Extraction and cleaning**: pulling raw trade data from Polymarket's on-chain records and structuring it into an analysis-ready format. This means resolving wallet addresses, computing per-trade profit and loss, and filtering out the noise trades that would obscure the signal.
2. **Feature engineering and detection**: constructing per-account behavioral features and applying unsupervised clustering and anomaly scoring to identify wallets and networks that fall outside the normal distribution of trading behavior.
3. **Visualization and interpretation**: rendering the results in forms that make patterns legible — wallet relationship graphs, win-rate distributions, event-level co-trading heatmaps — so that the output of the statistical pipeline can be evaluated by a human analyst rather than treated as a black box.



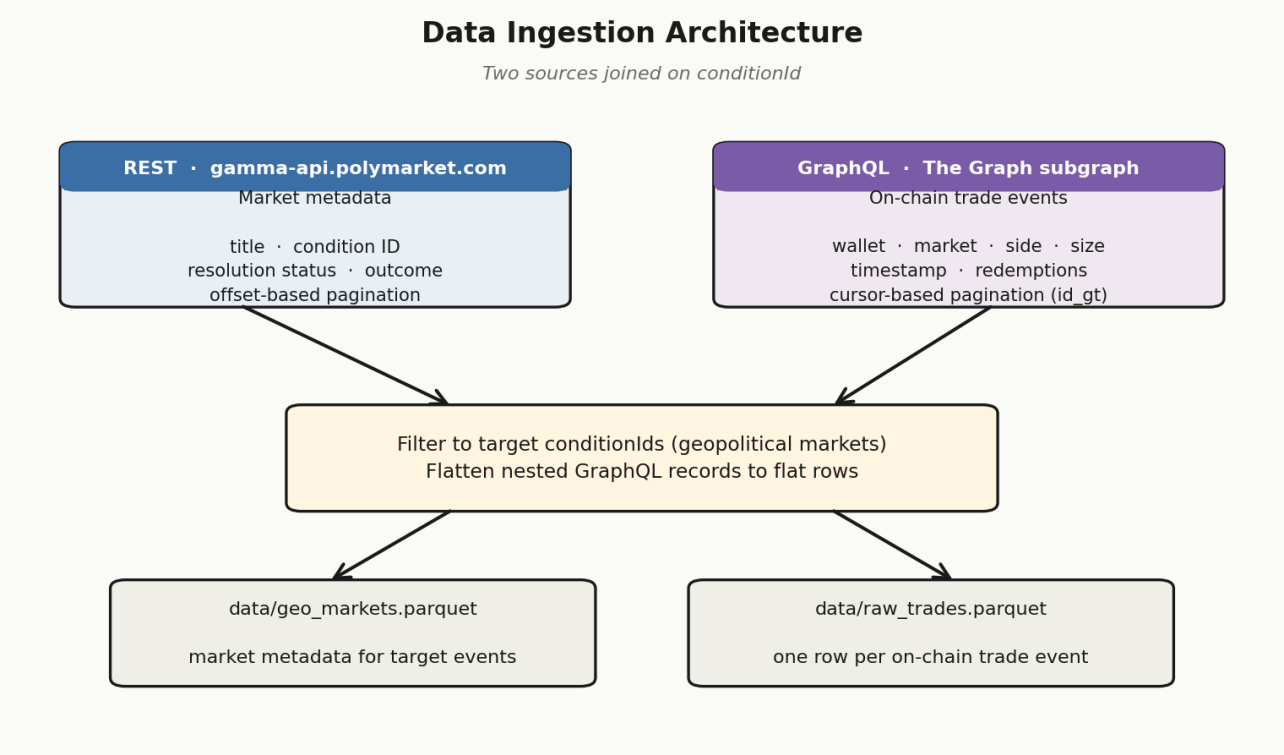
A few scoping notes before we move forward. We are not building a surveillance system for all of Polymarket. We are building a pipeline that operates on a defined corpus of geopolitical markets. The assumption throughout is that you have already identified which market IDs correspond to geopolitical events. The problem of classifying markets by type is real but separate. Our problem starts with a list of markets and asks: given all the trading activity on these markets, which accounts look like they knew something the rest of the market did not?

We are also not building a legal case. The output of this pipeline is a ranked list of suspicious accounts and clusters, scored by the strength of the anomaly signal. What you do with that list is outside the scope of this course. The engineering problem is to surface it reliably, at scale, with explainable features. That is hard enough.

The rest of this course builds toward a complete working pipeline. Chapter 2 starts where everything starts: pulling the data.

Chapter 2 — Data Extraction

Before you can detect anything, you need data. Polymarket exposes two primary access points for historical trade information: a REST API for market metadata and a GraphQL subgraph hosted on The Graph for on-chain event data. You will use both. The REST API gives you structured information about markets — titles, categories, resolution status, condition IDs. The subgraph gives you the low-level trade records: who bought what, when, at what price, and in what size. Getting comfortable with both interfaces is the first real engineering task.



This chapter walks through the full ingestion pipeline: querying the Polymarket API for market data, constructing and paginating GraphQL queries against the subgraph, filtering down to your target market IDs, and persisting the results in a format that the rest of the pipeline can consume cleanly. By the end, you will have a working Python script that produces structured trade records for a defined set of geopolitical markets.

Start with the market metadata.

Polymarket's REST API is available at `https://gamma-api.polymarket.com`. The endpoint you care about most at this stage is `/markets`, which returns a list of market objects including their unique identifiers, titles, condition IDs, resolution status, and timestamps. The condition ID is the critical linking field: it appears in both the REST response and the on-chain subgraph data, and it is how you will join market metadata to trade records later.

A basic request looks like this:

```
import requests
import time

BASE_URL = "https://gamma-api.polymarket.com"

def fetch_markets(limit=100, offset=0):
    params = {
        "limit": limit,
        "offset": offset,
    }
    response = requests.get(f"{BASE_URL}/markets", params=params)
    response.raise_for_status()
    return response.json()
```

The API returns paginated results. You will need to loop until you have pulled everything. The pagination here is offset-based, which means you increment by your limit on each call until you get back fewer results than you asked for, which signals that you have reached the end.

```
def fetch_all_markets():
    all_markets = []
    offset = 0
    limit = 100
    while True:
        batch = fetch_markets(limit=limit, offset=offset)
        if not batch:
            break
        all_markets.extend(batch)
        if len(batch) < limit:
            break
        offset += limit
        time.sleep(0.2)
    return all_markets
```

Once you have the full market list, you filter it down to your target set. The assumption from Chapter 1 holds here: you already have a list of market IDs or condition IDs corresponding to the geopolitical events you care about. Store those in a plain text file or a simple JSON list. The filtering step is just a set intersection:

```
import json

with open("geopolitical_market_ids.json") as f:
    target_ids = set(json.load(f))

all_markets = fetch_all_markets()
geo_markets = [m for m in all_markets if m["conditionId"] in target_ids]
```

Keep the full metadata for each market. You will want the resolution timestamp and the winning outcome later when you compute per-trade profit and loss in Chapter 3. Write the filtered market list to disk now so you do not have to re-fetch it during downstream steps.

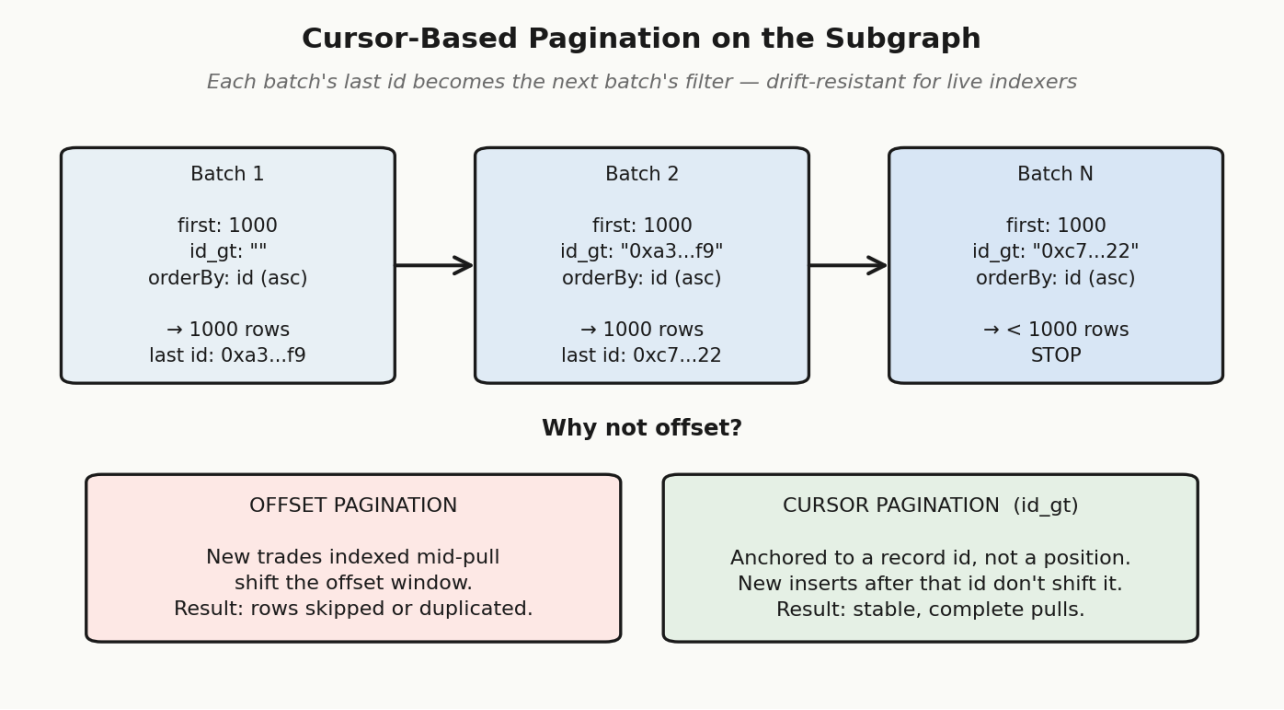
```
import pandas as pd

markets_df = pd.DataFrame(geo_markets)
markets_df.to_parquet("data/geo_markets.parquet", index=False)
```

Now move to the subgraph.

The Graph hosts an indexed version of Polymarket's on-chain activity. The relevant subgraph for Polymarket on Polygon is accessible at an endpoint in this form: `https://api.thegraph.com/subgraphs/name/polymarket/matic-markets`. The exact subgraph slug may shift as Polymarket updates its indexing infrastructure, so verify the current endpoint in Polymarket's developer documentation before running this in production. The schema is consistent even if the URL changes.

The entities you need from the subgraph are trades, also sometimes indexed under names like `fpmmTrades` or `orderFilledEvents` depending on the subgraph version. Each trade record contains the maker or taker wallet address, the condition ID linking to the market, the outcome index (which side was bought), the collateral amount, the share amount, and a Unix timestamp. These fields are sufficient to reconstruct the full trade history for any account on any market.

GraphQL queries against The Graph use cursor-based pagination, not offset pagination. The standard approach is to query a fixed batch size, take the ID of the last record returned, and pass that as a filter on the next query. This is more reliable than offset pagination for large datasets because it does not drift when new records are indexed during your pull.



A working query looks like this:

```
TRADE_QUERY = """
query GetTrades($conditionId: String!, $lastId: String!) {
  fpmmTrades(
    first: 1000,
    where: {
      fpmm_: { condition: $conditionId },
      id_gt: $lastId
    },
    orderBy: id,
    orderDirection: asc
  ) {
    id
    creator {
      id
    }
    outcomeIndex
    collateralAmount
    outcomeTokensAmount
    creationTimestamp
    fpmm {
      id
      condition {
        id
      }
    }
  }
}
"""
```

Note that the exact field names depend on which version of the Polymarket subgraph you are querying. The structure above matches the common schema at time of writing, but you should run an introspection query against the subgraph endpoint first to confirm the available fields and entity names. Introspection looks like this:

```
INTROSPECTION_QUERY = """
{
  __schema {
    types {
      name
      fields {
        name
      }
    }
  }
}
"""

def introspect_schema(endpoint):
    response = requests.post(endpoint, json={"query": INTROSPECTION_QUERY})
    response.raise_for_status()
    return response.json()
```

Run that once, inspect the output, and confirm that the field names in your trade query match what the subgraph actually exposes. Ten minutes here prevents hours of silent failures later.

With the query confirmed, the pagination loop is straightforward:

```
SUBGRAPH_URL = "https://api.thegraph.com/subgraphs/name/polymarket/matic-markets"

def fetch_trades_for_market(condition_id):
    all_trades = []
    last_id = ""
    while True:
        payload = {
            "query": TRADE_QUERY,
            "variables": {
                "conditionId": condition_id,
                "lastId": last_id
            }
        }
        response = requests.post(SUBGRAPH_URL, json=payload)
        response.raise_for_status()
        data = response.json()
        trades = data["data"]["fpmmTrades"]
        if not trades:
            break
        all_trades.extend(trades)
        last_id = trades[-1]["id"]
        if len(trades) < 1000:
            break
        time.sleep(0.1)
    return all_trades
```

Call this for each condition ID in your geopolitical market list:

```
all_trades = []
for market in geo_markets:
    condition_id = market["conditionId"]
    trades = fetch_trades_for_market(condition_id)
    all_trades.extend(trades)
    time.sleep(0.2)
```

The sleep calls matter. The Graph has rate limits, and hitting them mid-pull produces errors that are annoying to recover from. A short pause between requests keeps you well within the limits for a dataset of the size you are likely working with.

At this point you have a list of raw trade dictionaries. Before writing them to disk, do a minimal flattening pass. The nested structure from GraphQL is not convenient for pandas and downstream processing. Pull the fields you need to the top level:

```
def flatten_trade(trade):
    return {
        "trade_id": trade["id"],
        "wallet": trade["creator"]["id"],
        "condition_id": trade["fpmm"]["condition"]["id"],
        "outcome_index": int(trade["outcomeIndex"]),
        "collateral_amount": int(trade["collateralAmount"]),
        "outcome_tokens_amount": int(trade["outcomeTokensAmount"]),
        "timestamp": int(trade["creationTimestamp"]),
    }

flat_trades = [flatten_trade(t) for t in all_trades]
trades_df = pd.DataFrame(flat_trades)
trades_df.to_parquet("data/raw_trades.parquet", index=False)
```

Two notes on the numeric fields. Collateral amounts and token amounts from the subgraph are returned as large integers representing fixed-point values with 6 decimal places (USDC uses 6 decimals on Polygon). You are not dividing by 1,000,000 yet because that belongs in the cleaning step in Chapter 3. Keeping them as raw integers here avoids floating-point noise in the persisted data.

The timestamp field is a Unix timestamp in seconds. Keep it that way for now. You will convert it to datetime objects in Chapter 3 when you need to do time-based calculations.

There is one more data source worth pulling at this stage: position redemption events. When a market resolves and a winning account redeems their shares, that action also appears on-chain. Capturing redemption records gives you a direct signal of who actually collected winnings, which is more reliable than inferring profitability purely from the trade side. The subgraph typically exposes these as `fpmmFundingAdditions` or condition redemption events depending on the indexing approach. The query structure is identical to the trade query above; only the entity name and fields change. Pull those records now, flatten them the same way, and write them to a separate parquet file. You will join them to the trade data in Chapter 3.

```
REDEMPTION_QUERY = """
query GetRedemptions($conditionId: String!, $lastId: String!) {
  conditionRedemptions(
    first: 1000,
    where: {
      condition: $conditionId,
      id_gt: $lastId
    },
    orderBy: id,
    orderDirection: asc
  ) {
    id
    redeemer {
      id
    }
    condition {
      id
    }
    payout
    creationTimestamp
  }
}
"""
```

The pattern for paginating and flattening this is identical to the trade pull. The resulting redemption records give you ground truth on who was on the winning side and how much they collected. That is the foundation of the win-rate calculation that sits at the center of everything in Chapter 4.

Before moving on, run a quick sanity check on your raw data. Verify that the number of unique condition IDs in your trades parquet matches the number of markets you targeted. Check that the timestamp range makes sense. Count the number of unique wallet addresses. If any of these look obviously wrong, the most common causes are a malformed condition ID in your target list, a subgraph entity name mismatch, or a pagination loop that exited early due to a transient API error. Fix them now. The cleaning and feature engineering steps in the chapters ahead assume that the raw data is complete.

```
print(f"Unique markets: {trades_df['condition_id'].nunique()}")
print(f"Expected markets: {len(geo_markets)}")
print(f"Unique wallets: {trades_df['wallet'].nunique()}")
print(f"Total trades: {len(trades_df)}")
print(f"Timestamp range: {trades_df['timestamp'].min()} to
{trades_df['timestamp'].max()}")
```

When these numbers look right, your ingestion pipeline is done. You have market metadata and raw trade records for your full set of geopolitical markets, structured as flat parquet files, ready to be cleaned and transformed.

Chapter 2 Output: Raw Data Schemas

geo_markets.parquet

conditionId	*str*
title	*str*
resolved	*bool*
resolution_outcome	*str*
resolved_at	*int*
category	*str*

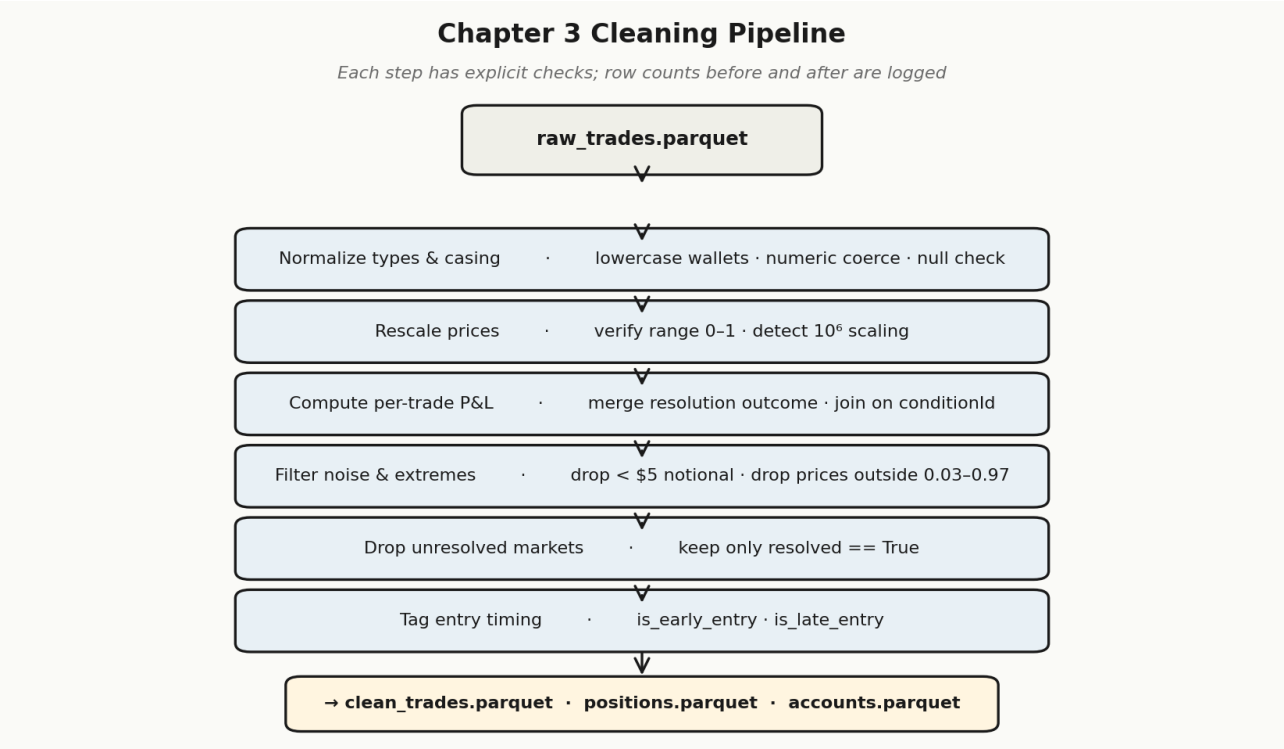
raw_trades.parquet

trade_id	*str*
wallet	*str*
condition_id	*str*
outcome_index	*int (0=YES, 1=NO)*
collateral_amount	*int (USDC × 10^6)*
outcome_tokens	*int (shares × 10^6)*
timestamp	*int (Unix sec)*

Chapter 3 takes it from here.

Chapter 3 — Data Engineering

Raw trade data from the subgraph is honest but inconvenient. The records are flat, the prices are in fractional USDC units, the wallet addresses are unsanitized, and nothing has been assembled into the account-level view that the feature engineering in Chapter 4 will need. This chapter closes that gap. By the end, you will have two structured tables: one at the level of individual trades, with profit and loss computed, and one at the level of accounts, aggregating behavior across all their geopolitical market activity. Both tables will be clean, typed correctly, and schematically stable enough to feed into machine learning downstream.



Start by loading what Chapter 2 produced.

```
import pandas as pd

trades = pd.read_parquet("raw_trades.parquet")
markets = pd.read_parquet("geo_markets.parquet")
```

Before touching anything else, check your column types. Timestamps should be numeric unix integers or datetime objects, not strings. Prices and amounts should be floats. Wallet addresses should be lowercase strings. In practice they rarely are, because the subgraph returns whatever the chain recorded, and chains are inconsistent about address casing. Fix that first.

```
trades["wallet"] = trades["wallet"].str.lower().str.strip()
trades["timestamp"] = pd.to_numeric(trades["timestamp"], errors="coerce")
trades["price"] = pd.to_numeric(trades["price"], errors="coerce")
trades["amount"] = pd.to_numeric(trades["amount"], errors="coerce")
```

Run a null check immediately after.

```
print(trades.isnull().sum())
```

Any nulls in wallet, timestamp, price, or amount are problems. Nulls in wallet usually mean the subgraph returned a transaction from a contract address that was not fully resolved. Nulls in price or amount usually mean a malformed record from a market that had a non-standard settlement flow. Drop these rows and log how many you lost.

```
n_before = len(trades)
trades = trades.dropna(subset=["wallet", "timestamp", "price", "amount"])
print(f"Dropped {n_before - len(trades)} rows with nulls")
```

If you dropped more than a couple of percent, go back and investigate before continuing. A high null rate usually indicates something structural, not random noise, and continuing on a corrupted base will produce subtly wrong features that are hard to debug later.

The price column deserves special attention. Polymarket prices represent the implied probability of a binary outcome, and they are expressed as a decimal between 0 and 1. A trade recorded at 0.73 means the buyer paid 73 cents per share, expecting a payout of 1 USDC if the market resolves YES. But the subgraph sometimes returns prices in a scaled integer format depending on which contract version the market used. You need to know which you have.

```
print(trades["price"].describe())
```

If the max is around 1.0, you are already in decimal form. If the max is around 100 or 1000000, the values are scaled and you need to divide. For markets using the CTF (Conditional Token Framework) with 6-decimal USDC, the typical scaling factor is 1,000,000. Check against a few known trades manually if you are uncertain. Getting this wrong will make every profit calculation wrong, so it is worth taking a minute to verify.

Assume from here that prices are in the range 0 to 1. If you had to rescale, do it now and re-run the describe check.

The amount column represents the number of outcome shares purchased or sold. To compute what a trader spent entering a position, you multiply shares by price. To compute their gross payout at resolution, you need to know the resolution outcome for the market. That lives in the markets table.

Merge the resolution outcome into the trades table.

```
trades = trades.merge(
    markets[["condition_id", "resolved", "resolution_outcome"]],
    on="condition_id",
    how="left"
)
```

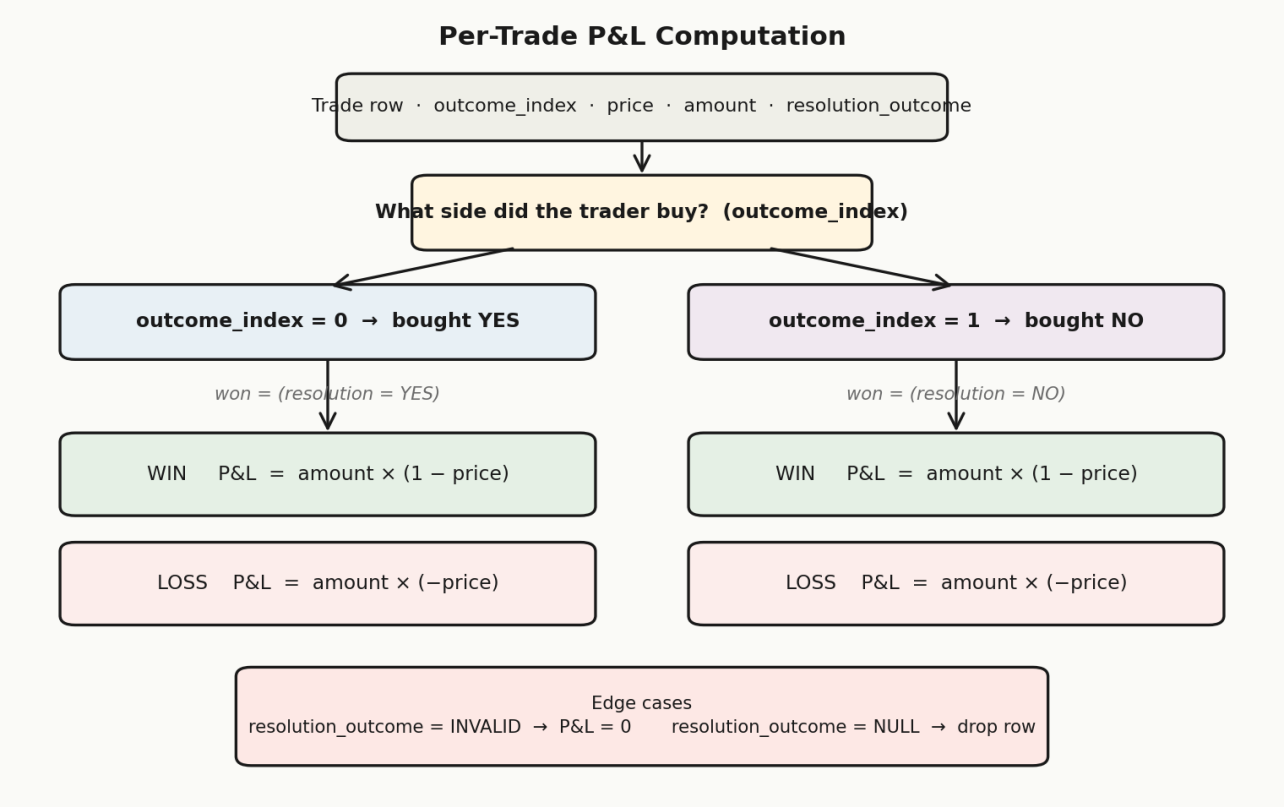
The `resolution_outcome` column should contain a string: `"YES"`, `"NO"`, or potentially `"INVALID"` for markets that were voided. If your markets table uses 1 and 0 instead, standardize to strings for readability before going further.

```
outcome_map = {1: "YES", 0: "NO"}
trades["resolution_outcome"] = trades["resolution_outcome"].map(outcome_map).fillna(trad
es["resolution_outcome"])
```

Now you can compute per-trade profit and loss. The logic is straightforward for resolved markets. If the trader bought YES shares and the market resolved YES, they receive 1 USDC per share and their profit is shares times (1 minus entry price). If the market resolved NO, they receive nothing and their loss is shares times entry price. You also need to handle the case where a trader bought NO shares, which is equivalent to buying YES shares at (1 minus the displayed price), depending on how your raw data represents it.

Polymarket's subgraph typically records all trades from the perspective of the token being bought. A YES token purchase at 0.73 and a NO token purchase at 0.27 are two separate records pointing at two separate token addresses. The `outcome_index` column, if present, tells you which token was purchased. If it is 0, the trader bought YES tokens. If it is 1, they bought NO tokens.

```
def compute_pnl(row):
    if row["resolution_outcome"] == "INVALID":
        return 0.0
    if pd.isna(row["resolution_outcome"]):
        return None  # unresolved, skip for now
    if row["outcome_index"] == 0:
        won = row["resolution_outcome"] == "YES"
    else:
        won = row["resolution_outcome"] == "NO"
    if won:
        return row["amount"] * (1 - row["price"])
    else:
        return row["amount"] * (-row["price"])

trades["pnl"] = trades.apply(compute_pnl, axis=1)
```



`apply` is slow on large frames. Once you have validated this logic on a sample, rewrite it as vectorized operations for production use. For now, correctness matters more than speed.

Drop trades where pnl is None, meaning the market has not resolved yet. These contribute nothing to a historical pattern analysis.

```
trades = trades.dropna(subset=["pnl"])
```

At this point you also want to filter out unresolved markets entirely from the markets table, since any account-level aggregations you build later should only reflect completed events.

```
markets = markets[markets["resolved"] == True]
valid_condition_ids = set(markets["condition_id"])
trades = trades[trades["condition_id"].isin(valid_condition_ids)]
```

Now handle noise trades. Not every transaction in the dataset is a meaningful position. Small speculative trades, transactions that appear to be testing wallet connectivity, and dust positions that accumulated through liquidity provision are all noise relative to the behavior patterns you are trying to detect. A reasonable noise filter removes any trade where the notional value, computed as shares times price, is below a minimum threshold. A threshold of 5 USDC is conservative. Some practitioners go as high as 20 USDC for geopolitical markets where meaningful positions tend to be larger.

```
trades["notional"] = trades["amount"] * trades["price"]
min_notional = 5.0
n_before = len(trades)
trades = trades[trades["notional"] >= min_notional]
print(f"Dropped {n_before - len(trades)} noise trades below ${min_notional} notional")
```

Also filter out trades at extreme prices. A trade at 0.99 or 0.01 implies the outcome was nearly certain at the time of entry and carries almost no information about predictive edge. These also inflate win rates when the outcome goes the expected way, because betting on a 99-cent YES position requires no special knowledge. A reasonable filter removes trades where price is above 0.97 or below 0.03.

```
trades = trades[(trades["price"] >= 0.03) & (trades["price"] <= 0.97)]
```

This is a judgment call. You can tune these thresholds later when you see the distribution of features in Chapter 4. Record what you chose and why, because these decisions will affect your downstream results and you will want to revisit them.

The next complication is split positions. A single trader may enter a position on the same market at multiple price points across multiple timestamps. This is normal behavior, analogous to dollar-cost averaging, and it creates multiple rows in the trades table that belong to the same effective bet. For feature engineering purposes you will sometimes want to treat these as a single position rather than multiple independent trades.

Build a position table by grouping on wallet and `condition_id`.

```
positions = trades.groupby(["wallet", "condition_id"]).agg(
    total_shares=("amount", "sum"),
    total_notional=("notional", "sum"),
    total_pnl=("pnl", "sum"),
    first_entry=("timestamp", "min"),
    last_entry=("timestamp", "max"),
    trade_count=("amount", "count"),
    outcome_index=("outcome_index", "first"),
    resolution_outcome=("resolution_outcome", "first")
).reset_index()
```

The average entry price for a split position is the weighted average, not a simple mean.

```
positions["avg_price"] = positions["total_notional"] / positions["total_shares"]
```

A position is a win if `total_pnl` is positive. Add that flag.

```
positions["win"] = (positions["total_pnl"] > 0).astype(int)
```

You now have a positions table where each row represents one wallet's complete exposure to one market. This is the right level of abstraction for most of what follows.

Late liquidity is another edge case worth handling explicitly. Some markets on Polymarket attract a surge of trading volume in the final hours before resolution, as informed or speculative actors rush to take positions. If you are computing entry timing

relative to resolution, trades entered in this late window will cluster together and can skew timing features. You do not need to drop these trades, but you do need to flag them so that feature engineering can handle them correctly.

You will need the resolution timestamp for each market. If your markets table includes a `resolved_at` field, use it. If not, you can approximate it by taking the maximum trade timestamp within each `condition_id`, which is a rough proxy that works well enough for most markets.

```
resolution_times = trades.groupby("condition_id")
["timestamp"].max().rename("approx_resolution_ts")
positions = positions.merge(resolution_times, on="condition_id", how="left")
positions["hours_before_resolution"] = (
    (positions["approx_resolution_ts"] - positions["first_entry"]) / 3600
)
```

A position entered more than 72 hours before resolution is a clear early bet. A position entered within 6 hours is a late entry. Flag both.

```
positions["is_early_entry"] = (positions["hours_before_resolution"] >= 72).astype(int)
positions["is_late_entry"] = (positions["hours_before_resolution"] <= 6).astype(int)
```

Now build the account-level table. This aggregates across all a wallet's positions and is what Chapter 4 will use as its primary input for feature construction.

```
accounts = positions.groupby("wallet").agg(
    total_positions=("condition_id", "count"),
    win_count=("win", "sum"),
    total_pnl=("total_pnl", "sum"),
    total_notional=("total_notional", "sum"),
    avg_hours_before_resolution=("hours_before_resolution", "mean"),
    early_entry_count=("is_early_entry", "sum"),
    unique_markets=("condition_id", "nunique")
).reset_index()

accounts["win_rate"] = accounts["win_count"] / accounts["total_positions"]
accounts["roi"] = accounts["total_pnl"] / accounts["total_notional"]
```

These two columns, `win_rate` and `roi`, are the raw signal at the center of the detection problem. Chapter 4 will refine them into proper features, but already you can do a quick sanity check.

```
print(accounts[["win_rate", "roi", "total_positions"]].describe())
```

You expect most win rates to cluster around 0.5 to 0.6 for active traders. ROI should average negative, because market takers pay the spread and most traders lose. If the median win rate is above 0.7, something is wrong with your PnL logic or your noise filter is too loose. If the median ROI is positive, double-check that you are not accidentally including the market maker's liquidity provision trades in your dataset.

Wallet age is the last piece of account-level metadata you need at this stage. A wallet that was created shortly before a cluster of profitable trades is more suspicious than one with years of activity. The proxy for wallet age is the first timestamp any trade was recorded from that address, across all markets, not just geopolitical ones. You can compute this from your full trades table.

```
wallet_first_seen = trades.groupby("wallet")["timestamp"].min().rename("first_seen_ts")
accounts = accounts.merge(wallet_first_seen, on="wallet", how="left")
```

Convert to a human-readable date for inspection.

```
accounts["first_seen_date"] = pd.to_datetime(accounts["first_seen_ts"], unit="s")
```

You do not have true wallet creation dates from the chain without a separate on-chain lookup, but first trade timestamp is a reasonable approximation for this dataset. In Chapter 4, wallet age will be computed as the difference between `first_seen_ts` and the timestamp of the account's earliest geopolitical position.

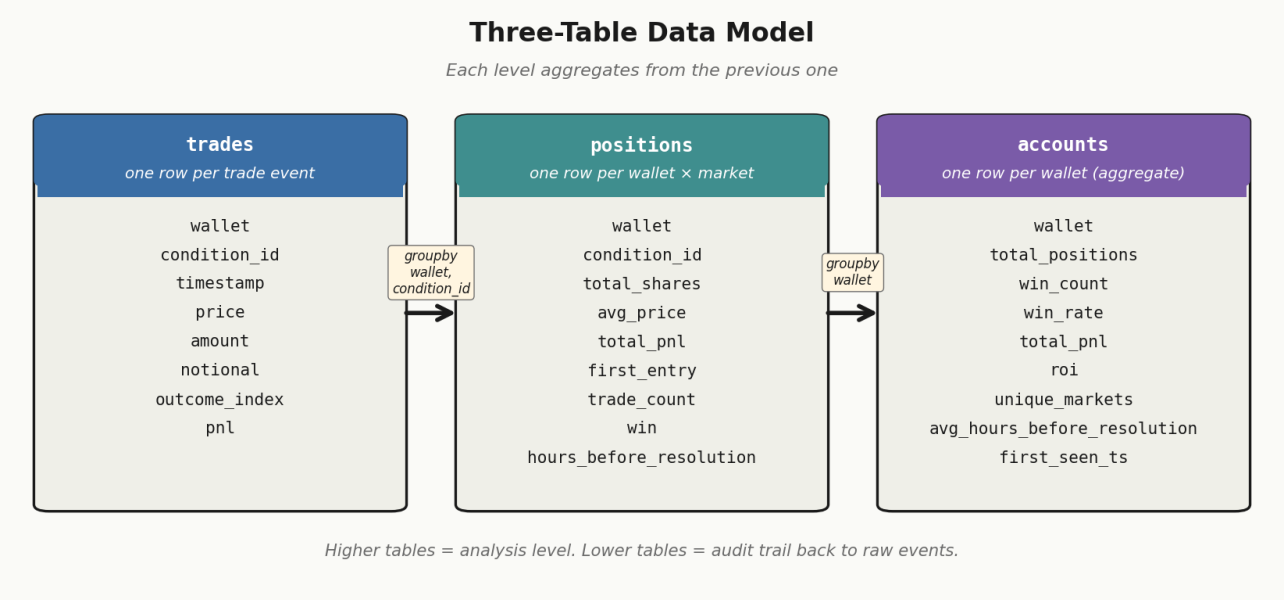
Before saving, make one final schema check. Your trades table should have at minimum these columns: `wallet`, `condition_id`, `timestamp`, `price`, `amount`, `outcome_index`, `notional`, `pnl`, `resolution_outcome`. Your positions table should have: `wallet`, `condition_id`, `total_shares`, `total_notional`, `total_pnl`, `avg_price`, `first_entry`, `hours_before_resolution`, `win`, `is_early_entry`, `is_late_entry`. Your accounts table should have: `wallet`, `total_positions`, `win_count`, `win_rate`, `total_pnl`, `roi`, `avg_hours_before_resolution`, `unique_markets`, `first_seen_ts`.

```
trades.to_parquet("clean_trades.parquet", index=False)
positions.to_parquet("positions.parquet", index=False)
accounts.to_parquet("accounts.parquet", index=False)
```

Print a summary of each.

```
for name, df in [("trades", trades), ("positions", positions), ("accounts", accounts)]:
    print(f"{name}: {len(df)} rows, {df['wallet'].nunique()} unique wallets")
```

If these numbers look consistent, your data engineering stage is complete. The trades table is the record of every meaningful transaction. The positions table collapses those into per-wallet, per-market exposures. The accounts table gives you one row per actor with their aggregate behavior across all geopolitical markets.

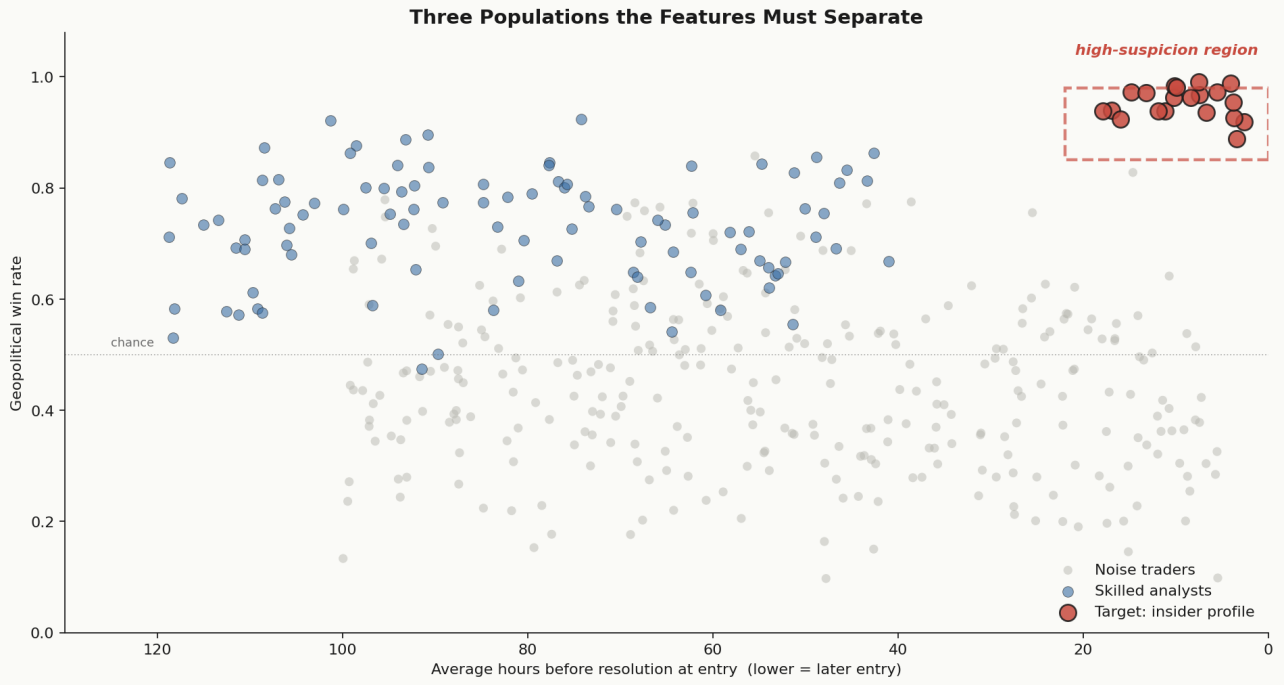


Chapter 4 takes these three tables and builds the feature vectors that will power detection.

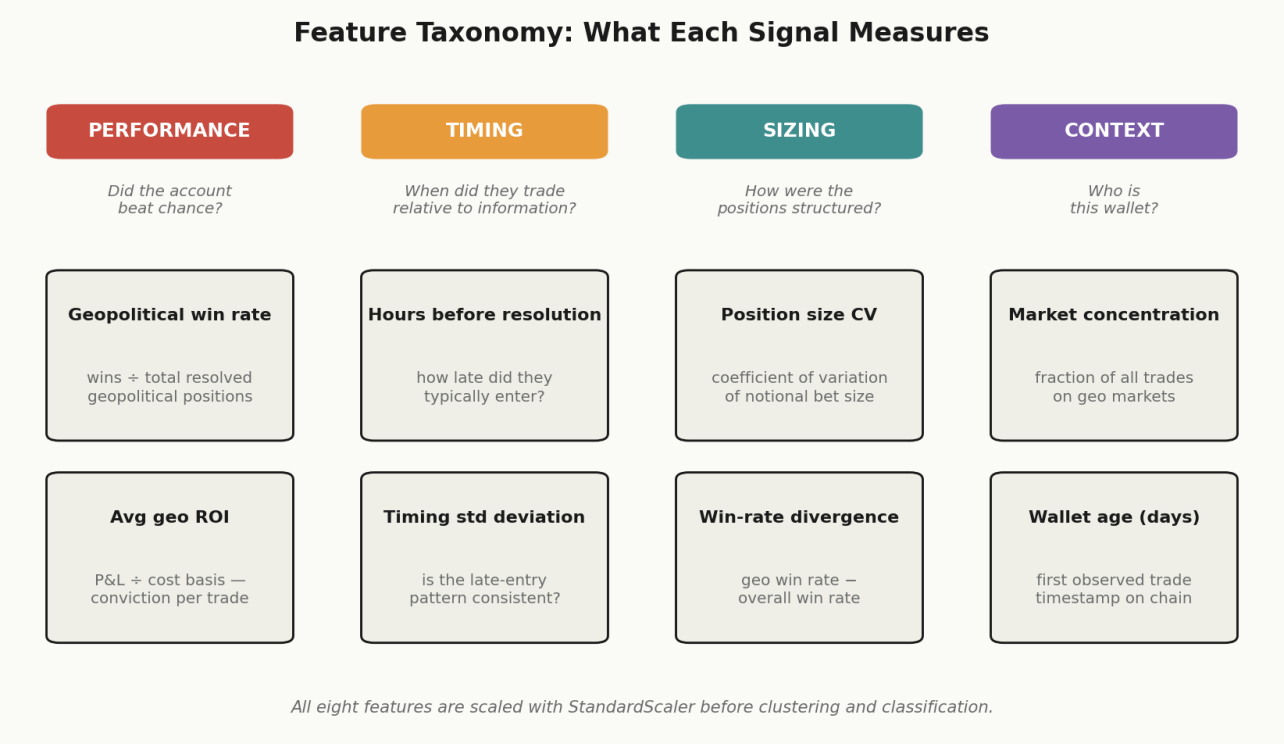
Chapter 4 — Feature Engineering

The accounts table you built in Chapter 3 already contains some aggregate statistics: total trades, win rate, ROI. Those numbers are a start, but they are too coarse to distinguish between someone who got lucky on five bets and someone who systematically bought correct positions on geopolitical events hours before resolution. Feature engineering is where you close that gap. You're going to take the three tables you have and construct a per-account feature vector that encodes behavioral patterns specific to information advantage.

Before writing any code, it's worth being precise about what you're trying to separate. Three populations exist in this data. The first is **noise traders**: low-volume accounts, high churn, mediocre win rates, no particular timing signal. The second is **skilled analysts**: high win rates that are explainable by volume and patience, no unusual timing concentration, diverse market participation. The third is the population you're hunting: accounts with high win rates concentrated on geopolitical events, entries that cluster suspiciously close to the moment information becomes public or before it does, position sizes that don't vary much because the actor already knows the answer, and sometimes coordinated behavior with other accounts. The features you build need to pull these three populations apart.



You'll construct eight features per account. Each one is doing a specific job.



Feature 1: Geopolitical Win Rate

You computed a raw win rate in Chapter 3, but it was across all markets. Recompute it using only positions where `market_type` equals `"geopolitical"` and the position was resolved. This matters because a trader might have average performance overall but extraordinary performance specifically on the events that require private information.

```
geo_positions = positions[positions["market_type"] == "geopolitical"].copy()

geo_wins = (
    geo_positions[geo_positions["resolved"] == True]
    .groupby("wallet")
    .apply(lambda x: (x["pnl"] > 0).sum() / len(x))
    .rename("geo_win_rate")
)
```

An account with a 0.55 overall win rate but a 0.91 geopolitical win rate is telling you something. The divergence between general and geopolitical performance is itself informative, so you'll add it as a separate feature later.

Feature 2: Number of Qualifying Trades

Raw counts matter for statistical weight. A 90% win rate over 5 trades is noise. Over 25 trades it deserves attention. Use the count of geopolitical positions that were resolved.

```
geo_trade_count = (
    geo_positions[geo_positions["resolved"] == True]
    .groupby("wallet")
    .size()
    .rename("geo_trade_count")
)
```

You'll apply a minimum threshold of 5 when you construct the final feature matrix, but keep the raw count as a feature so the classifier can weight it appropriately.

Feature 3: Average Entry Timing Relative to Resolution

This is the most discriminating single feature in the set. If an account consistently enters positions within a narrow window before resolution, that's the behavioral fingerprint of someone acting on late-breaking private information. You need two things: the trade timestamp and the market resolution timestamp. Both are in your trades table.

```
trades["hours_before_resolution"] = (
    (trades["resolution_time"] - trades["timestamp"]).dt.total_seconds() / 3600
)

timing_stats = (
    trades[trades["market_type"] == "geopolitical"]
    .groupby("wallet")["hours_before_resolution"]
    .agg(["mean", "std"])
    .rename(columns={"mean": "avg_hours_before_resolution", "std": "timing_std"})
)
```

Two sub-features come out of this. The mean tells you how close to resolution the account typically enters. The standard deviation tells you how consistent that timing is. Low mean and low standard deviation together describe someone who repeatedly enters late and does so reliably, not randomly. An account with an average entry of 3 hours before resolution and a standard deviation of 1.2 hours is very different from one averaging 72 hours with a standard deviation of 48 hours.

One edge case: some trades will have resolution times before trade times in the raw data, usually due to timestamp errors or late-reporting positions that were actually opened earlier. Filter these out before computing the feature.

```
trades = trades[trades["hours_before_resolution"] > 0]
```

Feature 4: Position Size Consistency

Informed traders don't vary their position sizes much. If you already know the outcome, there's no reason to bet small. Noise traders and analysts both show more variation because their conviction fluctuates. Compute the coefficient of variation of notional trade size per account across geopolitical markets.

```
size_consistency = (
    trades[trades["market_type"] == "geopolitical"]
    .groupby("wallet")["notional_usd"]
    .agg(lambda x: x.std() / x.mean() if x.mean() > 0 else None)
    .rename("position_size_cv")
)
```

Lower coefficient of variation means more consistent sizing. You're looking for accounts that bet in a narrow band regardless of the event. Invert this before adding it to the feature matrix if you want higher values to indicate more suspicion, or just let the model learn the direction from the training signal.

Feature 5: Win Rate Divergence

Compute the difference between geopolitical win rate and overall win rate. This separates informed traders from generally skilled ones. A person with genuinely good judgment about world events might score high on geopolitical win rate, but their overall win rate will also be elevated. An insider is often mediocre everywhere except the domain where they have an edge.

```
overall_win_rate = (
    positions[positions["resolved"] == True]
    .groupby("wallet")
    .apply(lambda x: (x["pnl"] > 0).sum() / len(x))
    .rename("overall_win_rate")
)

divergence = (geo_wins - overall_win_rate).rename("win_rate_divergence")
```

This feature will be noisy for accounts with few overall trades, which is another reason the `geo_trade_count` feature needs to be in the vector alongside it.

Feature 6: Market Concentration

If an account trades in many different market categories and also wins on geopolitical ones, they might just be broadly informed. If they trade almost exclusively on geopolitical markets, that's a narrower profile worth flagging. Compute the fraction of total trades that fall in the geopolitical category.

```
total_trade_count = trades.groupby("wallet").size().rename("total_trades")
geo_trade_count_all = (
    trades[trades["market_type"] == "geopolitical"]
    .groupby("wallet")
    .size()
    .rename("geo_trades_raw")
)

market_concentration = (geo_trade_count_all / total_trade_count).rename("geo_concentrati
on")
```

On its own this doesn't mean much, but combined with high win rate and tight timing, a concentration value above 0.8 is a meaningful signal.

Feature 7: Wallet Age

Freshly created wallets that immediately begin trading geopolitical events with high win rates are suspicious in a way that older accounts are not. Older accounts accumulate noise from earlier, less suspicious behavior. New accounts purpose-built for a specific information advantage tend to appear, trade aggressively on a cluster of events, and go quiet. Compute wallet age in days from the first observed trade.

```
wallet_first_trade = trades.groupby("wallet")
["timestamp"].min().rename("first_trade_date")
reference_date = trades["timestamp"].max()
wallet_age_days = ((reference_date - wallet_first_trade).dt.days).rename("wallet_age_day
s")
```

The reference date should be the last date in your dataset, not today's date, so this feature is stable and reproducible.

Feature 8: Average ROI on Geopolitical Positions

Win rate counts wins and losses equally regardless of size. ROI captures magnitude. An account that makes 3x on correct positions and loses minimally on incorrect ones is expressing more conviction per trade than one with an identical win count but flat returns. Compute mean ROI per resolved geopolitical position.

```
geo_roi = (
    geo_positions[geo_positions["resolved"] == True]
    .assign(roi=lambda x: x["pnl"] / x["cost_basis"])
    .groupby("wallet")["roi"]
    .mean()
    .rename("avg_geo_roi")
)
```

Make sure `cost_basis` is never zero here. If you see division errors, filter those rows out before computing ROI. They typically represent dust positions or accounting artifacts from the cleaning stage.

Assembling the Feature Matrix

Join all eight features on wallet and handle the missing values that will inevitably appear. Accounts that only traded non-geopolitical markets will have NaN for geopolitical features. Drop them: they're outside the scope of your detection target. Accounts with fewer than 5 resolved geopolitical trades should also be dropped at this stage.

```
feature_frames = [
    geo_wins,
    geo_trade_count,
    timing_stats,
    size_consistency,
    divergence,
    market_concentration,
    wallet_age_days,
    geo_roi,
]

features = geo_wins.to_frame()
for f in feature_frames[1:]:
    features = features.join(f, how="left")

features = features.join(overall_win_rate, how="left")
features["win_rate_divergence"] = features["geo_win_rate"] -
features["overall_win_rate"]

features = features[features["geo_trade_count"] >= 5].dropna()
```

At this point you have a matrix with one row per qualifying account and eight numeric columns. Print the shape and inspect the distributions.

```
print(features.shape)
print(features.describe())
```

What you're looking for is whether the features have meaningful spread. If `geo_win_rate` is clustered tightly around 0.5 with very few outliers, your data might be thin or your filtering thresholds too aggressive. If `avg_hours_before_resolution` has a bimodal distribution, that's already interesting and worth noting before you ever run a clustering algorithm.

Scale the features before passing them downstream. Clustering algorithms in Chapter 5 are distance-sensitive, and `wallet_age_days` is in hundreds of days while `geo_win_rate` is between 0 and 1. Use `StandardScaler`.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
feature_cols = [
    "geo_win_rate",
    "geo_trade_count",
    "avg_hours_before_resolution",
    "timing_std",
    "position_size_cv",
    "win_rate_divergence",
    "geo_concentration",
    "wallet_age_days",
    "avg_geo_roi",
]

features_scaled = features[feature_cols].copy()
features_scaled[feature_cols] = scaler.fit_transform(features_scaled[feature_cols])
```

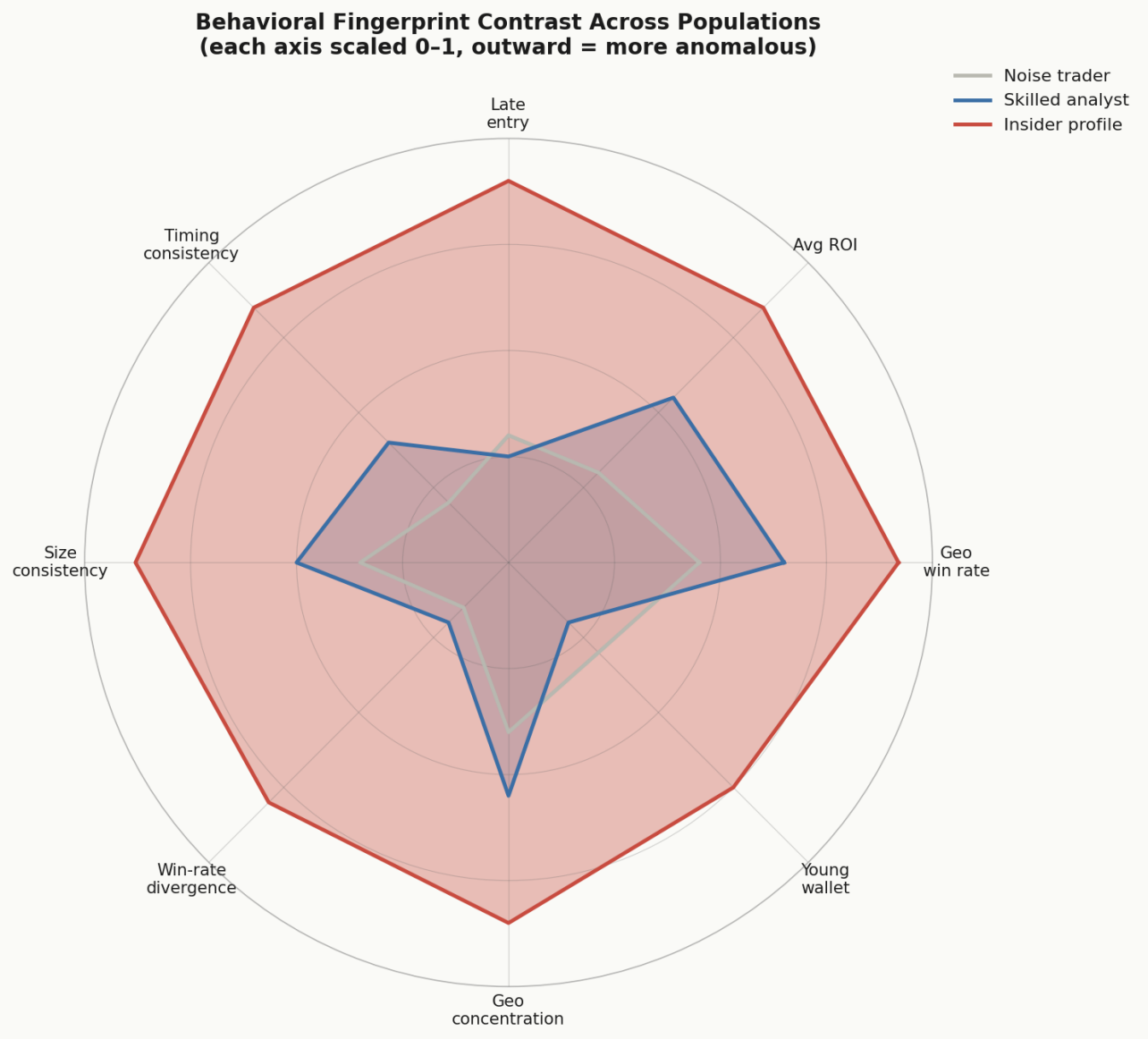
Save both the raw and scaled versions. You'll need the raw version for human-readable reporting and the scaled version for clustering and classification.

```
features.to_parquet("features_raw.parquet")
features_scaled.to_parquet("features_scaled.parquet")
```

One thing to check before you move on: look at the correlation matrix across your features. Highly correlated features aren't fatal but they compress information and can distort distance-based clustering. If `avg_hours_before_resolution` and `timing_std` are correlated above 0.85, consider whether you want to keep both or drop one. In practice they tend to carry different signal — the mean captures the typical behavior and the standard deviation captures the reliability of it — but verify this in your actual data.

```
import seaborn as sns
import matplotlib.pyplot as plt

corr = features[feature_cols].corr()
sns.heatmap(corr, annot=True, fmt=".2f", cmap="coolwarm")
plt.tight_layout()
plt.savefig("feature_correlation.png")
```

The feature matrix is now ready. Each row is an account. Each column is a behavioral signal. The accounts with the highest suspicion scores will tend to show up in a particular region of this space: high `geo_win_rate`, high `avg_geo_roi`, low `avg_hours_before_resolution`, low `timing_std`, low `position_size_cv`, and a `wallet_age` that is either very young or shows a sharp change in activity pattern.



Chapter 5 will use this matrix to find groups of accounts that move together.

Chapter 5 — Clustering

The feature matrix from Chapter 4 tells you about individual accounts. A high geopolitical win rate, tight entry timing, consistent position sizing — these are the markers of an account worth watching. But insider trading on prediction markets rarely happens in isolation. The more interesting pattern is coordination: multiple wallets entering the same position on the same event within hours of each other, executing a similar trade size, and collecting their winnings in parallel. Individual anomalies might be explainable. Coordinated anomalies are harder to dismiss.

This chapter finds those groups. We'll work in two stages. First, we'll build a co-trading similarity matrix that captures how frequently pairs of accounts trade together on the same geopolitical events with similar timing and sizing. Then we'll apply clustering algorithms to that matrix to surface groups of accounts that behave like a coordinated unit, whether or not they share any obvious on-chain relationship.

The distinction between the two stages matters. Feature-space clustering groups accounts that look similar behaviorally. Co-trading clustering groups accounts that actually traded together. Both are useful. Used together, they give you coordinated accounts that also share suspicious behavioral fingerprints, which is the strongest signal the pipeline produces.

Building the co-trading similarity matrix

A co-trading event between two accounts occurs when both accounts took a position on the same market within a defined time window. The tighter the window, the more meaningful the co-occurrence. A window of 24 hours before resolution captures early positioning on fast-moving events. A window of 72 hours gives you more signal on slower markets but more noise from coincidental co-trading. Start at 24 hours and adjust after looking at your distribution.

Start with the trades dataframe from Chapter 3, filtered to geopolitical markets and restricted to accounts that appear in your feature matrix. The schema you need is at minimum: `account_id`, `market_id`, `entry_timestamp`, `position_size`, and outcome (winning/losing).

```
import pandas as pd
import numpy as np
from itertools import combinations

geo_trades = trades[trades["market_id"].isin(geo_market_ids)].copy()
geo_trades = geo_trades[geo_trades["account_id"].isin(feature_matrix.index)]

# attach resolution timestamp to each trade
geo_trades = geo_trades.merge(
    market_metadata[["market_id", "resolution_timestamp"]],
    on="market_id",
    how="left"
)

geo_trades["hours_before_resolution"] = (
    geo_trades["resolution_timestamp"] - geo_trades["entry_timestamp"]
).dt.total_seconds() / 3600

window_hours = 24
geo_trades["in_window"] = geo_trades["hours_before_resolution"] <= window_hours
window_trades = geo_trades[geo_trades["in_window"]].copy()
```

Now, for each market, collect all accounts that traded within the window and generate all pairs. For each pair, record whether they traded the same side (both YES or both NO) and how similar their position sizes were.

```
records = []

for market_id, group in window_trades.groupby("market_id"):
    accounts = group["account_id"].unique()
    if len(accounts) < 2:
        continue
    for a1, a2 in combinations(accounts, 2):
        t1 = group[group["account_id"] == a1].iloc[0]
        t2 = group[group["account_id"] == a2].iloc[0]
        same_side = int(t1["outcome_side"] == t2["outcome_side"])
        size_ratio = min(t1["position_size"], t2["position_size"]) / (
            max(t1["position_size"], t2["position_size"]) + 1e-9
        )
        records.append({
            "account_a": a1,
            "account_b": a2,
            "market_id": market_id,
            "same_side": same_side,
            "size_ratio": size_ratio,
        })

cotrades = pd.DataFrame(records)
```

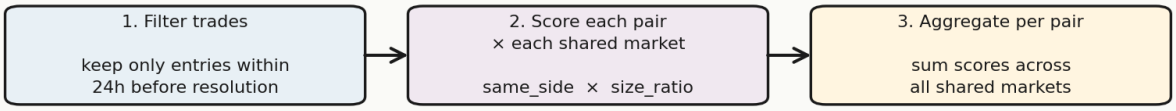
The `size_ratio` is 1 when positions are identical and approaches 0 when one is much larger than the other. Same-side co-trading that is size-matched is the strongest individual co-trade signal.

From the per-event records, aggregate up to a per-pair similarity score. The simplest version sums a weighted co-trade count across all shared markets.

```
cotrades["co_trade_score"] = cotrades["same_side"] * cotrades["size_ratio"]

pair_scores = (
    cotrades.groupby(["account_a", "account_b"])["co_trade_score"]
    .sum()
    .reset_index()
    .rename(columns={"co_trade_score": "similarity"})
)
```

Building the Co-Trading Similarity Matrix

For every pair of accounts, score the markets they traded together in the same window



Example: accounts A and B share three markets within the window

market	A side	B side	size A	size B	same_side × size_ratio
evt_017	YES	YES	$1,200	$1,150	1.00 × 0.96 = 0.96
evt_042	YES	YES	$2,000	$1,900	1.00 × 0.95 = 0.95
evt_088	NO	NO	$800	$850	1.00 × 0.94 = 0.94

→ similarity(A, B) = 0.96 + 0.95 + 0.94 = 2.85

You now have a long-form similarity table. The next step is turning it into a square matrix. Get the full list of accounts and index into them.

```
accounts = sorted(
    set(pair_scores["account_a"]).union(set(pair_scores["account_b"]))
)
n = len(accounts)
idx = {a: i for i, a in enumerate(accounts)}

sim_matrix = np.zeros((n, n))
for _, row in pair_scores.iterrows():
    i = idx[row["account_a"]]
    j = idx[row["account_b"]]
    sim_matrix[i, j] = row["similarity"]
    sim_matrix[j, i] = row["similarity"]

sim_df = pd.DataFrame(sim_matrix, index=accounts, columns=accounts)
```

Before you cluster, look at the distribution of pairwise similarity scores. Most pairs will have zero or near-zero similarity. A small number will have high scores. Those are the pairs worth clustering. If the distribution looks flat, your window is too wide or you have too few markets to generate meaningful co-occurrence signal.

DBSCAN on the similarity matrix

DBSCAN is the right first choice here. Unlike k-means, it does not require you to specify the number of clusters in advance. It also naturally handles noise: accounts that do not fit into any cluster are labeled `-1` rather than forced into a group. That matters because most accounts in your dataset are not coordinated, and you do not want to create spurious clusters to accommodate them.

DBSCAN works on a distance matrix. Similarity and distance are inverses, so convert before clustering. Use 1 minus normalized similarity.

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.cluster import DBSCAN

sim_normalized = sim_matrix / (sim_matrix.max() + 1e-9)
dist_matrix = 1 - sim_normalized

db = DBSCAN(eps=0.3, min_samples=2, metric="precomputed")
labels = db.fit_predict(dist_matrix)

cluster_assignments = pd.Series(labels, index=accounts, name="cluster_id")
```

The two parameters to tune are `eps` and `min_samples`. `eps` is the maximum distance between two accounts for one to be considered in the neighborhood of the other. `min_samples` is the minimum number of accounts in a neighborhood to form a dense region. With `eps=0.3` and `min_samples=2`, you are saying: two accounts are in the same cluster if their distance is at most 0.3, and any two connected accounts constitute a valid cluster. This is intentionally permissive because insider trading clusters can be small, sometimes just two or three wallets.

If you are getting too many clusters and they look like noise, raise `eps` or raise `min_samples`. If you are getting one giant cluster that absorbs most accounts, lower `eps`. The right output has a small number of clusters with 2 to 10 members each, plus a large noise category.

Check your cluster sizes and which accounts ended up together.

```
cluster_summary = (
    cluster_assignments[cluster_assignments >= 0]
    .reset_index()
    .rename(columns={"index": "account_id"})
    .groupby("cluster_id")["account_id"]
    .apply(list)
)

for cid, members in cluster_summary.items():
    print(f"Cluster {cid}: {len(members)} accounts")
    print(members)
```

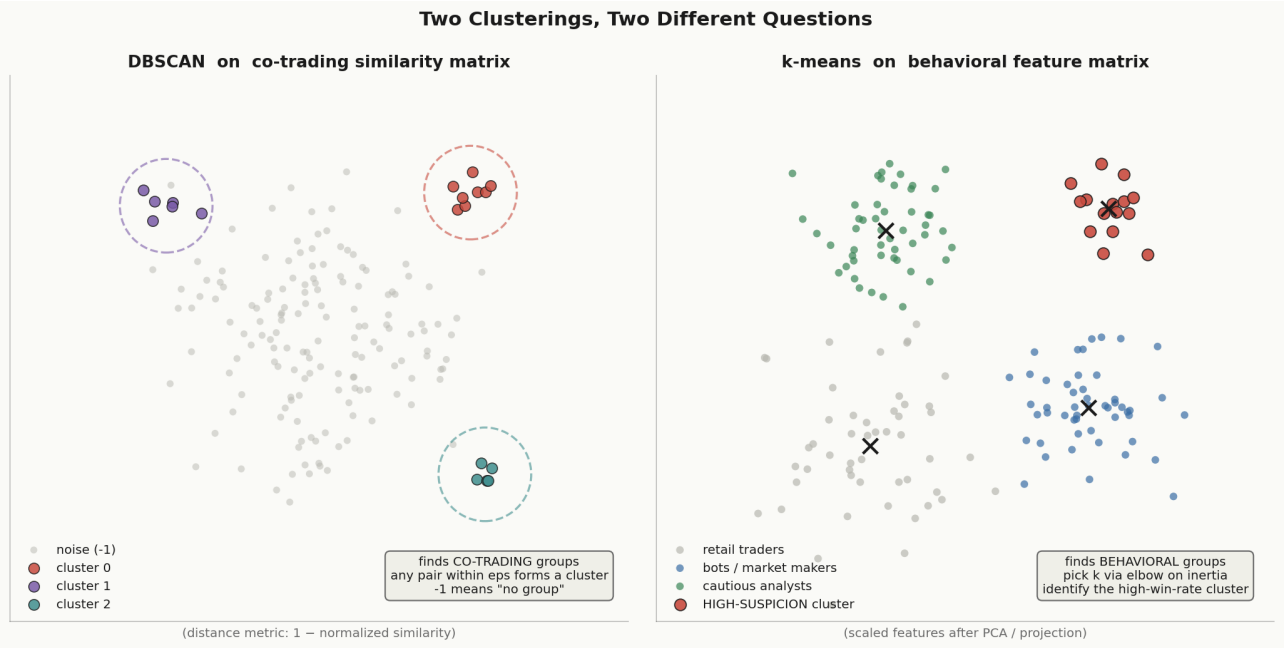
Cross-reference the cluster members against your feature matrix immediately. A cluster is only interesting if its members also show up in the high-suspicion region of the feature space: high `geo_win_rate`, low `avg_hours_before_resolution`, high `avg_geo_roi`. A cluster of accounts with mediocre win rates who happen to co-trade frequently is probably a group of bots running a market-making strategy, not insiders.

```
feature_matrix["cluster_id"] = cluster_assignments.reindex(feature_matrix.index).values

cluster_feature_means = (
    feature_matrix[feature_matrix["cluster_id"] >= 0]
    .groupby("cluster_id")[feature_cols]
    .mean()
)
print(cluster_feature_means)
```

This table is your first interpretive tool. Look for clusters whose mean `geo_win_rate` exceeds 0.80, whose mean `avg_hours_before_resolution` is below 12, and whose mean `avg_geo_roi` is above 1.5. Those are the clusters to investigate.

k-means on the feature matrix

DBSCAN on the similarity matrix finds coordination. k-means on the feature matrix finds behavioral similarity. They answer different questions and will not always agree, which is useful: when an account appears in a suspicious DBSCAN cluster and also clusters with high-suspicion accounts in feature space, you have convergent evidence.



k-means requires you to choose `k`. A reasonable starting point is to run it for `k` from 2 to 10 and use the elbow method on inertia to pick a value.

```python
from sklearn.cluster import KMeans

X = feature_matrix[feature_cols].values

inertia = []
k_range = range(2, 11)

for k in k_range:
    km = KMeans(n_clusters=k, random_state=42, n_init=10)
    km.fit(X)
    inertia.append(km.inertia_)

# plot inertia vs k to find the elbow
import matplotlib.pyplot as plt

plt.plot(list(k_range), inertia, marker="o")
plt.xlabel("k")
plt.ylabel("inertia")
plt.tight_layout()
plt.savefig("kmeans_elbow.png")
```

In practice, the elbow on prediction market behavioral data tends to appear around `k=4` or `k=5`. You will typically see one cluster of heavy retail traders with low win rates, one cluster of bots with high trade counts and variable win rates, one cluster of cautious legitimate analysts with moderate win rates and long entry timing, and one or two clusters of accounts with high win rates and early entry timing. Those last clusters are where the insiders tend to live.

Fit k-means with your chosen `k` and attach the labels.

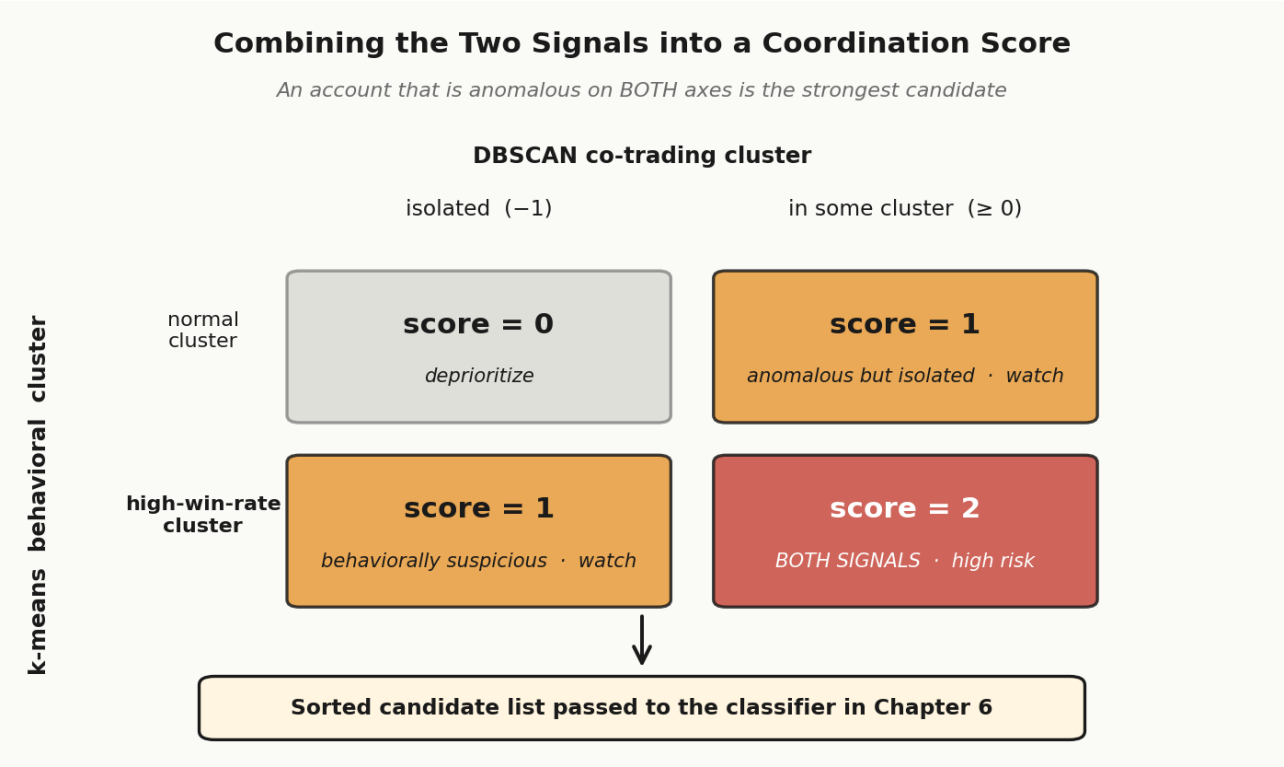
```python
k = 5
km = KMeans(n_clusters=k, random_state=42, n_init=10)
feature_matrix["kmeans_cluster"] = km.fit_predict(X)

kmeans_means = feature_matrix.groupby("kmeans_cluster")[feature_cols].mean()
print(kmeans_means)
```

Do not over-interpret the cluster numbers. They are arbitrary labels. What matters is the mean feature profile of each cluster. Identify which cluster has the highest `geo_win_rate` and lowest `avg_hours_before_resolution` and treat that as your high-suspicion cluster.

Combining the two signals

You now have two cluster labels per account: one from DBSCAN on co-trading similarity, one from k-means on behavioral features. Combine them into a simple coordination score.

```
suspicious_kmeans_cluster = kmeans_means["geo_win_rate"].idxmax()

feature_matrix["behavioral_suspicious"] = (
    feature_matrix["kmeans_cluster"] == suspicious_kmeans_cluster
).astype(int)

feature_matrix["coordinated"] = (
    feature_matrix["cluster_id"] >= 0
).astype(int)

feature_matrix["coordination_score"] = (
    feature_matrix["behavioral_suspicious"] + feature_matrix["coordinated"]
)
```



Accounts with a `coordination_score` of 2 are both behaviorally anomalous and co-trading with at least one other anomalous account. These are the accounts the classifier in Chapter 6 will prioritize. Accounts with a score of 1 in either dimension are worth tracking but require more evidence. Accounts with a score of 0 can be deprioritized.

One practical note about dataset size: co-trading similarity matrices grow quadratically in the number of accounts. For a dataset of 10,000 accounts, the distance matrix is 10,000 by 10,000 and DBSCAN over a precomputed matrix will be slow. If you hit performance limits, apply a preliminary filter to reduce the account set before building the similarity matrix. Only include accounts that appear at least three times in the `window_trades` dataframe. This removes one-off traders who cannot form a meaningful co-trading signal and substantially reduces the matrix size without losing relevant accounts.

```
active_accounts = (
    window_trades.groupby("account_id")["market_id"]
    .nunique()
)
active_accounts = active_accounts[active_accounts >= 3].index

window_trades_filtered = window_trades[
    window_trades["account_id"].isin(active_accounts)
]
```

Rerun the pair generation on the filtered set. The clustering output will be faster and cleaner.

A note on parameter sensitivity

Clustering is sensitive to its inputs in ways that classification is not. A small change in `eps` can merge two distinct clusters or split a real one. A different choice of `k` can shift which accounts fall into the high-suspicion group. Before treating any cluster output as reliable, run a sensitivity check.

For DBSCAN, try `eps` values of 0.2, 0.3, and 0.4 and compare the cluster membership for accounts you already suspect based on their feature profiles. If the same pairs consistently appear in the same cluster across all three settings, the co-trading signal is robust. If cluster membership changes dramatically with small `eps` adjustments, the co-trading similarity scores are noisy and you should revisit your window size or scoring formula.

For k-means, try `k` values of 4, 5, and 6. Check whether the same accounts end up in the high-suspicion cluster across all three runs. Stable membership across `k` values means the behavioral separation is real. Unstable membership means the feature space does not have a clean high-suspicion cluster, which could indicate that your features need adjustment or that the dataset contains fewer anomalous accounts than expected.

The goal is not to find the perfect clustering. The goal is to generate a set of accounts that score high on both dimensions consistently, regardless of reasonable parameter variation. Those accounts go into the classifier in Chapter 6 as the strong positive candidates.

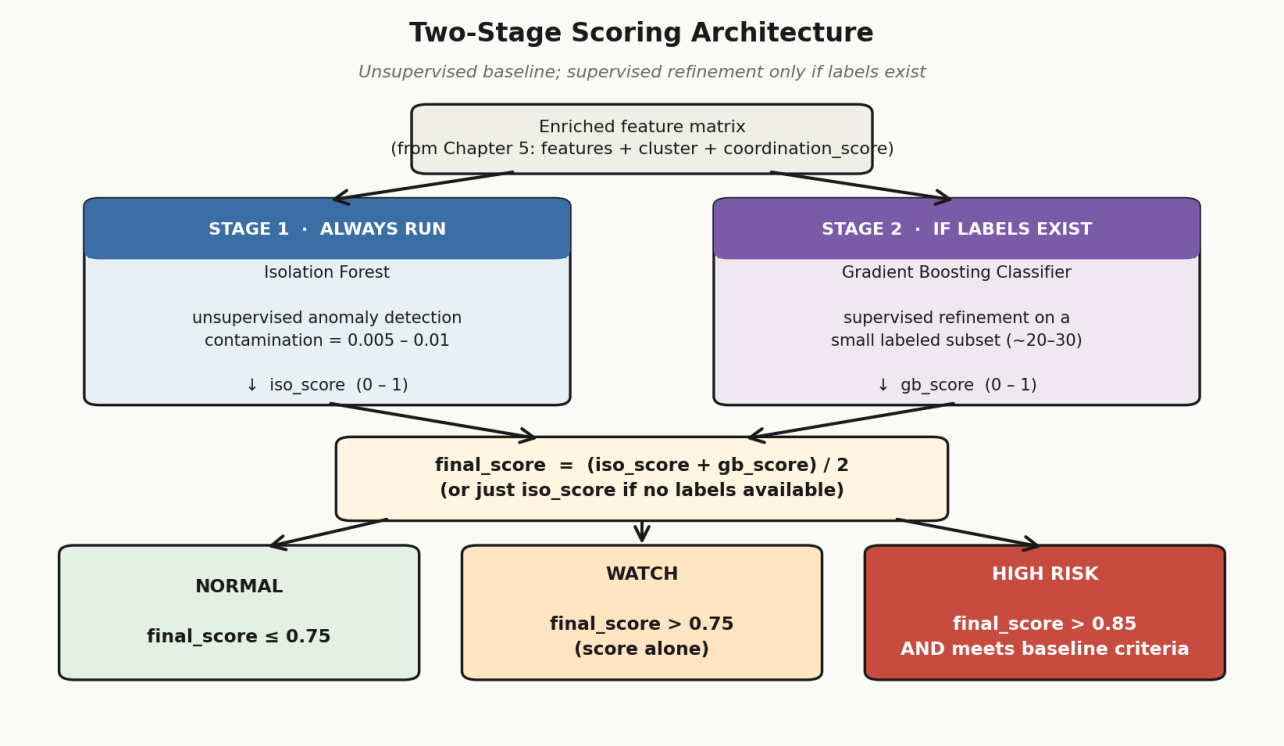
The feature matrix now has cluster assignments, coordination scores, and behavioral flags. Chapter 6 will take that enriched matrix and build a classifier that produces a single per-account insider trading likelihood score.

Chapter 6 — Classification

The feature matrix coming out of Chapter 5 is rich but not yet actionable. You have per-account behavioral features, coordination scores, cluster assignments, and a handful of binary flags. What you do not have is a single number that tells you how worried to be about a given wallet. That is what the classifier produces.

This chapter builds that classifier. The challenge is the same one that comes up in any fraud detection context: you almost certainly do not have reliable ground-truth labels. No one has handed you a list of confirmed insider traders on Polymarket. You are working with signals, not proof. That shapes every decision in this chapter, from model selection to threshold setting to how you interpret output.

The approach here has two stages. First, you apply an isolation forest as an unsupervised anomaly scorer to generate a baseline likelihood score for every account. Second, if you have even a small set of manually reviewed accounts you are willing to treat as labeled examples, you layer a gradient boosting classifier on top to refine those scores. If you have no labeled examples at all, you stay in the isolation forest stage and treat the output as a risk ranking rather than a classification.



Either way, the output is a per-account score between 0 and 1, where higher values indicate higher insider trading likelihood. You set a threshold to produce a flagged set, and you validate that flagged set against what you know qualitatively about the accounts it contains.

Starting with the isolation forest is the right call because it makes no assumptions about what insider trading looks like in terms of a class boundary. It finds accounts that are anomalous relative to the overall population. The implicit assumption is that most accounts are not insiders, which is almost certainly true on a platform with tens of thousands of active wallets. Insiders are rare. Anomaly detection is built for rare.

Load your enriched feature matrix and prepare it for the model. Every feature column should be numeric at this point. The coordination score, cluster label (encoded as a binary suspicious cluster flag rather than a raw integer), win rate, average entry offset, trade count, and wallet age are your core inputs. Drop any columns that are redundant or that encode the same information twice.

```
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler
import pandas as pd
import numpy as np

df = pd.read_csv("feature_matrix_clustered.csv", index_col="wallet")

feature_cols = [
    "win_rate",
    "trade_count",
    "avg_entry_offset_hours",
    "median_position_size",
    "wallet_age_days",
    "coordination_score",
    "in_suspicious_cluster"
]

X = df[feature_cols].copy()
X = X.fillna(X.median())

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

The `fillna` call matters. Accounts with very few trades may have missing values on some features. Imputing with the median keeps those accounts in the dataset rather than silently dropping them. A wallet with only three trades but a coordination score of 0.9 is still interesting.

Now fit the isolation forest. The `contamination` parameter tells the model what fraction of the population to treat as anomalous. Set it conservatively. If you have 10,000 accounts and you expect at most a few dozen insiders, something like 0.005 to 0.01 is appropriate. Setting it too high means you flag hundreds of accounts that are merely unusual traders, not insiders. Setting it too low means you might miss coordinated clusters that fall just outside the decision boundary.

```
iso = IsolationForest(
    n_estimators=200,
    contamination=0.01,
    random_state=42
)
iso.fit(X_scaled)

raw_scores = iso.decision_function(X_scaled)
```

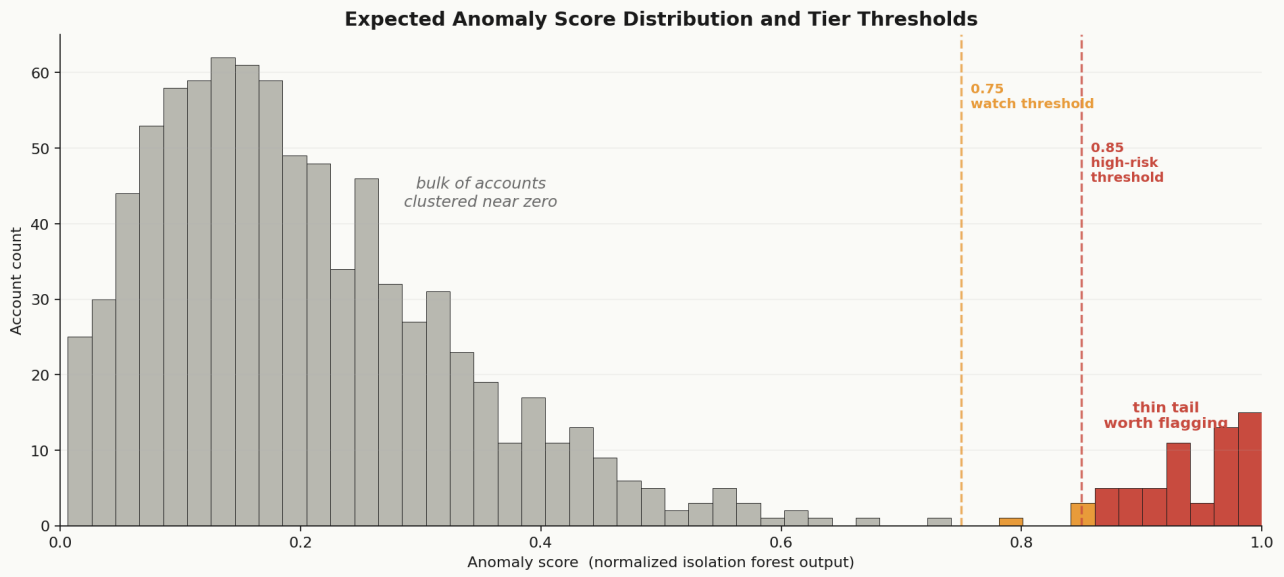
The `decision_function` output is the raw anomaly score. Lower values mean more anomalous. You want to invert and normalize this into a 0 to 1 range where 1 means most anomalous.

```
normalized_scores = 1 - (raw_scores - raw_scores.min()) / (raw_scores.max() -
raw_scores.min())
df["iso_score"] = normalized_scores
```

Take a look at the distribution before doing anything else.

```
import matplotlib.pyplot as plt

plt.hist(df["iso_score"], bins=50)
plt.xlabel("Isolation Forest Anomaly Score")
plt.ylabel("Account Count")
plt.title("Distribution of Anomaly Scores")
plt.show()
```

You expect a heavy right skew. Most accounts should cluster near 0, with a thin tail at the high end. If the distribution is roughly uniform or bimodal, something is wrong with your feature preparation. A uniform distribution usually means your features are not actually separating anomalous accounts from normal ones. Go back and check whether your coordination scores and win rates are behaving as expected before continuing.



If the distribution looks right, check who is in the tail. Pull the top 1% of accounts by `iso_score` and look at their raw feature values.

```
top_flagged = df.nlargest(int(len(df) * 0.01), "iso_score")
print(top_flagged[feature_cols + ["iso_score"]].to_string())
```

This is your first sanity check. The accounts in this list should have characteristics that match your intuition about what suspicious looks like: high win rates, early entry timing, high coordination scores, membership in suspicious clusters. If the top-flagged accounts have win rates around 0.5 and low coordination scores, the model is finding a different kind of anomaly than the one you care about. You would need to either reweight your features or add additional constraints.

One way to add constraints is to combine the isolation forest score with a simple rule-based filter. An account should not get flagged unless it meets a minimum threshold on the features you care about most.

```
df["candidate"] = (
    (df["iso_score"] > 0.7) &
    (df["win_rate"] > 0.65) &
    (df["trade_count"] >= 5)
)
```

The win rate and trade count thresholds here match the definitions from Chapter 1. You are not relying solely on the model. You are requiring that flagged accounts also satisfy the baseline criteria that motivated this whole pipeline in the first place.

Now consider whether you have any labeled data to work with. Even a small number of manually reviewed accounts can dramatically improve your precision. The label does not have to be "confirmed insider." It can be "clearly suspicious on manual review" versus "appears normal on manual review." Twenty to thirty accounts reviewed by a human analyst who understands the domain is enough to train a basic gradient boosting classifier.

If you have labels, structure them as a binary column in your dataframe.

```
labeled = df[df["label"].notna()].copy()
X_labeled = labeled[feature_cols]
y_labeled = labeled["label"]

from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import StratifiedKFold, cross_val_score

gb = GradientBoostingClassifier(
    n_estimators=100,
    max_depth=3,
    learning_rate=0.05,
    random_state=42
)

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(gb, X_labeled, y_labeled, cv=cv, scoring="roc_auc")
print(f"CV AUC: {scores.mean():.3f} +/- {scores.std():.3f}")
```

With very small labeled sets you should not trust a high cross-validation AUC. A 20-example dataset will produce wildly variable estimates. What you are checking is whether the model is doing better than chance (AUC above 0.5) and whether the variance is small enough that the signal seems real. If AUC is consistently above 0.75 across folds, that is encouraging. If it swings between 0.4 and 0.9, your labels are either inconsistent or your sample is too small to generalize.

If cross-validation looks reasonable, fit on all labeled data and generate probability scores for the full population.

```
gb.fit(X_labeled, y_labeled)
df["gb_score"] = gb.predict_proba(X_scaled)[:, 1]
```

Now you have two scores per account: the unsupervised isolation forest score and the supervised gradient boosting score. Combine them into a final score. A simple average works unless you have reason to weight one more heavily.

```
df["final_score"] = (df["iso_score"] + df["gb_score"]) / 2
```

If you only have the isolation forest and no labeled data, your final score is just `iso_score`. Do not pretend otherwise.

Setting the threshold for flagging accounts is a judgment call. There is no precision-recall curve you can draw without ground truth labels. Instead, use a tiered approach. Set a high-confidence tier and a watch list tier.

```
df["tier"] = "normal"
df.loc[df["final_score"] > 0.75, "tier"] = "watch"
df.loc[(df["final_score"] > 0.85) & (df["candidate"] == True), "tier"] = "high_risk"
```

The high-risk tier requires both a high model score and satisfaction of the baseline criteria from Chapter 1. The watch tier captures accounts that score highly on the model but may not fully meet the threshold criteria, perhaps because they have four qualifying trades instead of five. Both tiers are worth examining manually. Only the high-risk tier should be treated as a strong positive signal.

Print a summary to understand what you are working with.

```
print(df["tier"].value_counts())
print(df[df["tier"] == "high_risk"][feature_cols + ["final_score"]].describe())
```

Interpreting results without ground truth labels requires a different discipline than standard model evaluation. You cannot compute precision. You cannot compute recall. What you can do is check consistency and coherence.

Consistency means that accounts flagged as high-risk should have the characteristics you expected them to have before you ran the model. If a flagged account has a 0.45 win rate and joined the platform two years ago with hundreds of trades spread across all market types, something is wrong. Either the feature pipeline has a bug or the model is finding a different pattern than the one you intended.

Coherence means that high-risk accounts should cluster together in the co-trading graph from Chapter 5. If your top twenty flagged wallets show no overlap in the co-trading matrix, they are anomalous in different directions. That might be fine, but it should prompt investigation. Insiders operating as a network should appear proximate in the wallet graph.

Check this explicitly.

```
high_risk_wallets = df[df["tier"] == "high_risk"].index.tolist()
cotrade_subset = cotrade_matrix.loc[high_risk_wallets, high_risk_wallets]
print(cotrade_subset)
```

If the flagged accounts have high pairwise co-trading scores, your clustering and classification are telling a consistent story. If they do not, either the coordination signal is weak for this particular group or the classifier is surfacing behavioral anomalies unrelated to coordination.

Feature importance from the gradient boosting model is another interpretability tool. Even without labels for validation, understanding which features drive the score tells you whether the model has learned what you want it to learn.

```
importances = pd.Series(gb.feature_importances_, index=feature_cols)
print(importances.sort_values(ascending=False))
```

In a well-functioning model, `win_rate` and `avg_entry_offset_hours` should be among the top features. `coordination_score` and `in_suspicious_cluster` should also rank highly if coordinated accounts are the primary anomaly type in your data. If `wallet_age_days` is the dominant feature, the model may be overfitting to the fact that new wallets look unusual for reasons unrelated to insider trading.

If you are using only the isolation forest, you do not have feature importances in the traditional sense. You can approximate the contribution of each feature by computing how much the anomaly score drops when you shuffle that feature while holding others constant. This is a basic permutation importance approach.

```
baseline_scores = iso.decision_function(X_scaled)

perm_importances = {}
for col_idx, col in enumerate(feature_cols):
    X_permuted = X_scaled.copy()
    np.random.shuffle(X_permuted[:, col_idx])
    permuted_scores = iso.decision_function(X_permuted)
    perm_importances[col] = np.mean(np.abs(baseline_scores - permuted_scores))

print(pd.Series(perm_importances).sort_values(ascending=False))
```

Features with high permutation importance are the ones the isolation forest is relying on most. The same sanity check applies: you want win rate and timing features to dominate, not incidental attributes.

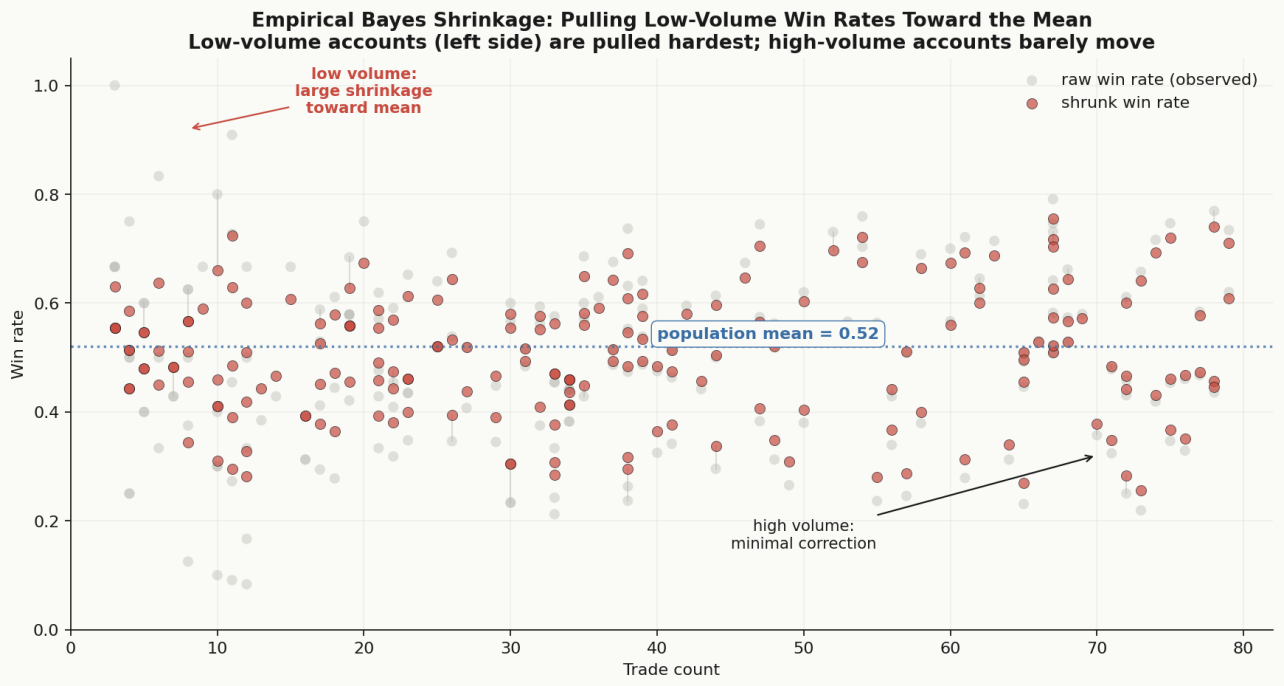
A common failure mode at this stage is over-flagging prolific traders. Accounts with many trades naturally have more variance in their feature values, and some of that variance can look anomalous. Add a check.

```
print(df[df["tier"] == "high_risk"]["trade_count"].describe())
```

If your high-risk accounts have suspiciously low trade counts across the board, the model may be flagging noise rather than signal. Accounts with only five or six qualifying trades can have artificially high win rates by chance. The win rate threshold in Chapter 1 exists precisely to guard against this, but it only covers the minimum trade count. If you see accounts with six trades and a 100% win rate dominating your high-risk tier, consider raising the minimum trade count threshold or applying a shrinkage correction to win rates for low-volume accounts.

A basic empirical Bayes shrinkage keeps win rates for low-volume accounts closer to the population mean.

```
population_mean_wr = df["win_rate"].mean()
k = 10

df["win_rate_shrunk"] = (
    (df["win_rate"] * df["trade_count"] + population_mean_wr * k) /
    (df["trade_count"] + k)
)
```



Replace `win_rate` with `win_rate_shrunk` in your feature matrix and rerun the models. High-volume accounts with genuinely high win rates will be barely affected. Low-volume accounts with fluky win rates will shrink toward the mean and drop out of the high-risk tier.

By the end of this chapter you have a scored feature matrix with tier assignments, a set of high-risk accounts that satisfy both model-based and rule-based criteria, and enough interpretability checks to trust that the model is responding to the right signals. The output is a dataframe with a wallet address, a final score, a tier label, and the underlying feature values for every account in your dataset.

Save it.

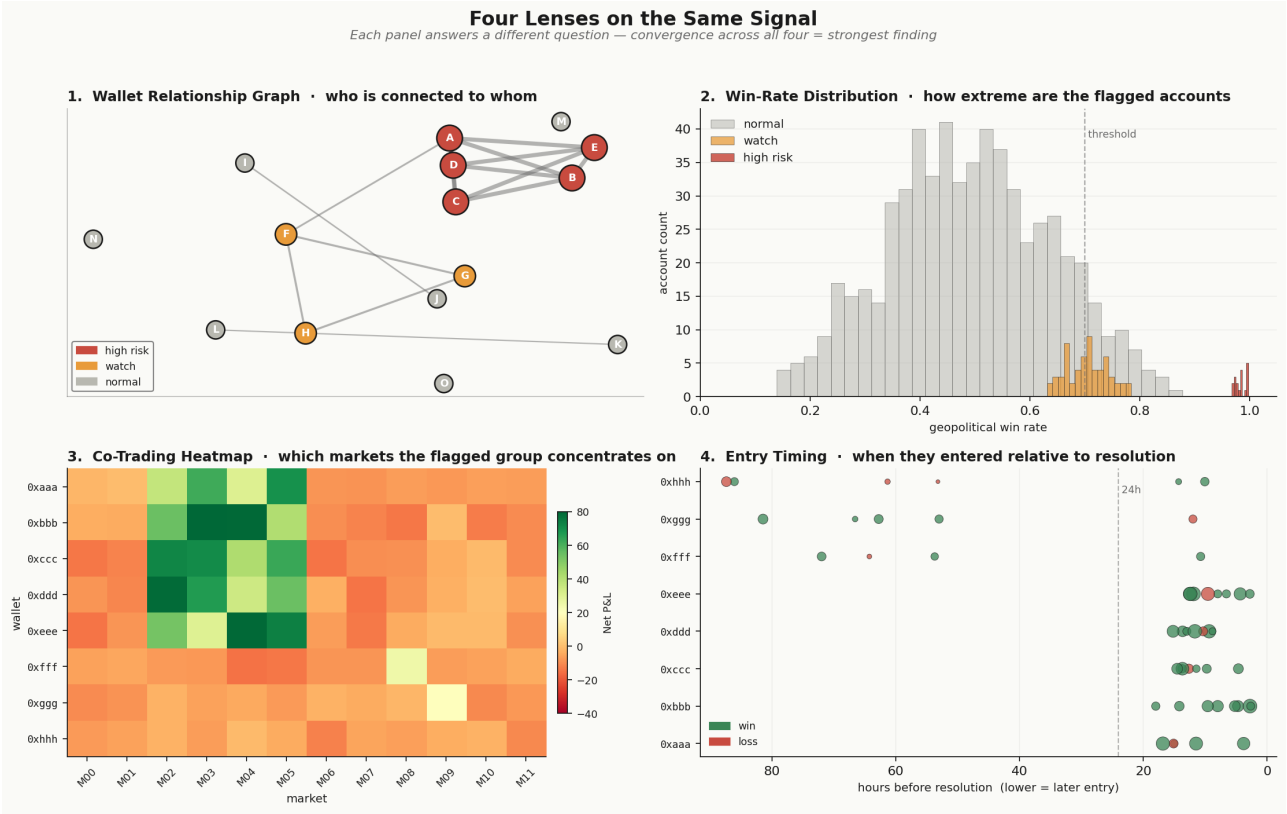
```
df.to_csv("scored_accounts.csv")
```

Chapter 7 will take this output and make it visual. A score in a spreadsheet is evidence. A wallet graph where the same flagged accounts cluster tightly together, with edge weights mapping to co-trading intensity and node color mapping to final score, is a finding.

Chapter 7 — Visualization

A score in a spreadsheet is evidence. A wallet graph where the same flagged accounts cluster tightly together, with edge weights mapping to co-trading intensity and node color mapping to risk score, is a finding. The difference matters when you're trying to communicate patterns to people who didn't build the pipeline, and it matters when you're trying to see things yourself that the numbers alone won't show you. Visualization is not a presentation layer you bolt on at the end. It's a second analysis pass, and it catches things the model misses.

This chapter walks through four visualization types: wallet relationship graphs, win-rate distribution charts, event-level co-trading heatmaps, and timeline plots of account activity relative to event resolution. Each one answers a different question. Together they give you a complete picture of what the flagged accounts are doing and how they relate to each other.



You'll need `pyvis`, `matplotlib`, `seaborn`, and `pandas` throughout. Install anything missing before starting.

```
pip install pyvis matplotlib seaborn pandas
```

Wallet Relationship Graphs

The co-trading similarity matrix you built in Chapter 5 encodes relationships between accounts: how often they traded the same markets at similar times with similar sizing. The graph visualization makes those relationships spatially legible. Accounts that co-trade heavily end up near each other. Clusters that were mathematically identified in Chapter 5 become visually obvious. And when those clusters overlap with the high-risk accounts from Chapter 6, you have something worth showing.

Start with the similarity matrix and the scored accounts dataframe.

```
import pandas as pd
import numpy as np
from pyvis.network import Network

similarity_matrix = pd.read_csv("similarity_matrix.csv", index_col=0)
scored = pd.read_csv("scored_accounts.csv", index_col=0)
```

You only want to show edges where the similarity is meaningful. A threshold around 0.3 to 0.4 works well in practice, but you should tune it for your dataset. Too low and you get a hairball. Too high and isolated high-risk accounts look unconnected when they're actually part of a loose cluster.

```
threshold = 0.35

edges = []
wallets = similarity_matrix.index.tolist()

for i in range(len(wallets)):
    for j in range(i + 1, len(wallets)):
        weight = similarity_matrix.iloc[i, j]
        if weight >= threshold:
            edges.append((wallets[i], wallets[j], float(weight)))
```

Now map each account to a color based on its risk tier. High-risk accounts get red. Watch-list accounts get orange. Normal accounts get a neutral gray. If an account doesn't appear in the scored dataframe for some reason, default to gray.

```
def get_color(wallet, scored_df):
    if wallet not in scored_df.index:
        return "#aaaaaa"
    tier = scored_df.loc[wallet, "risk_tier"]
    if tier == "high_risk":
        return "#e84040"
    elif tier == "watch":
        return "#f0a030"
    else:
        return "#aaaaaa"
```

Build the network. Set node size proportional to the number of qualifying trades so that heavy traders are visually prominent, and set edge width proportional to similarity weight so the strongest co-trading relationships stand out.

```
net = Network(height="800px", width="100%", bgcolor="#1a1a2e", font_color="white")
net.barnes_hut()

added_nodes = set()

for src, dst, weight in edges:
    for wallet in [src, dst]:
        if wallet not in added_nodes:
            color = get_color(wallet, scored)
            size = 10
            if wallet in scored.index:
                n_trades = scored.loc[wallet, "n_trades"]
                size = max(10, min(40, n_trades * 2))
            net.add_node(wallet, label=wallet[:8], color=color, size=size)
            added_nodes.add(wallet)
    net.add_edge(src, dst, value=weight, title=f"similarity: {weight:.2f}")

net.show("wallet_graph.html")
```

Open the resulting HTML file in a browser. What you're looking for is a tight red cluster: several high-risk accounts that are densely connected to each other, loosely connected to watch-list accounts, and largely disconnected from the gray background. If that pattern is present, the unsupervised clustering and the classifier are agreeing with each other, which is the strongest signal you have without ground truth labels.

If high-risk accounts are scattered across the graph with no coherent grouping, that's worth investigating. Either the similarity threshold is too low and you're seeing spurious edges, or the accounts that scored highly did so for individual behavioral reasons rather than coordinated activity. Both are possible. Check the feature vectors for the scattered high-risk accounts and look at whether they're genuinely isolated actors or whether they share markets with each other but just didn't hit the similarity threshold.

For larger graphs, `pyvis` can get slow. If you're working with more than a few hundred nodes, export to a GEXF file and use Gephi instead.

```
import networkx as nx

G = nx.Graph()

for wallet in added_nodes:
    color = get_color(wallet, scored)
    n_trades = scored.loc[wallet, "n_trades"] if wallet in scored.index else 1
    G.add_node(wallet, color=color, size=float(n_trades))

for src, dst, weight in edges:
    G.add_edge(src, dst, weight=weight)

nx.write_gexf(G, "wallet_graph.gexf")
```

In Gephi, open the file, apply a Force Atlas 2 layout, and color nodes by the color attribute. Use the edge weight as the edge thickness. The result is a higher-quality visualization that handles large graphs better and lets you apply additional community detection passes interactively.

Win-Rate Distribution Charts

The wallet graph shows relationships. The win-rate distribution chart shows how extreme the flagged accounts are relative to the broader population. This is the visualization that answers the luck question.

If the population of accounts with five or more geopolitical trades has a win-rate distribution centered around 0.55 with a long right tail, and the flagged accounts sit at 0.90 and above, that's a clear visual argument that something other than skill or luck is driving their performance. If the flagged accounts sit inside the normal distribution, the scoring thresholds probably need revisiting.

```
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

all_accounts = scored[scored["n_trades"] >= 5].copy()

fig, ax = plt.subplots(figsize=(10, 5))

normal = all_accounts[all_accounts["risk_tier"] == "normal"]["win_rate"]
watch = all_accounts[all_accounts["risk_tier"] == "watch"]["win_rate"]
high_risk = all_accounts[all_accounts["risk_tier"] == "high_risk"]["win_rate"]

ax.hist(normal, bins=30, alpha=0.5, color="#aaaaaa", label="normal")
ax.hist(watch, bins=20, alpha=0.6, color="#f0a030", label="watch")
ax.hist(high_risk, bins=10, alpha=0.8, color="#e84040", label="high_risk")

ax.set_xlabel("Win Rate (geopolitical trades)")
ax.set_ylabel("Account Count")
ax.set_title("Win-Rate Distribution by Risk Tier")
ax.legend()

plt.tight_layout()
plt.savefig("win_rate_distribution.png", dpi=150)
plt.show()
```

Read this chart carefully. The shape of the normal distribution tells you what baseline performance looks like on your dataset. A distribution centered significantly above 0.5 might mean your geopolitical market filter is pulling in markets where the outcome was heavily favored from the start, and win rate is not a clean signal. A distribution centered near 0.5 confirms that prediction is genuinely difficult across most of the population, which makes the right tail more meaningful.

Overlay a vertical line at whatever win-rate threshold you used in the scoring rules to make the cutoff explicit.

```
ax.axvline(x=0.70, color="white", linestyle="--", alpha=0.7, label="score threshold")
```

You can also plot this as a kernel density estimate if you prefer smooth curves over histograms. Use seaborn's `kdeplot` with the same color scheme. The histogram is usually more honest about sample sizes.

Event-Level Co-Trading Heatmaps

The win-rate chart is account-centric. The co-trading heatmap is event-centric. It shows which events the high-risk accounts concentrated their activity on, and whether that activity clustered in time.

Pull the trade-level data for flagged accounts and build a matrix of accounts by markets.

```
import seaborn as sns

trades = pd.read_csv("trades_clean.csv")

flagged_wallets = scored[scored["risk_tier"].isin(["high_risk", "watch"])].index.tolist(
)
flagged_trades = trades[trades["wallet"].isin(flagged_wallets)]

pivot = flagged_trades.pivot_table(
    index="wallet",
    columns="market_id",
    values="profit_loss",
    aggfunc="sum",
    fill_value=0
)
```

Truncate wallet addresses for display so the labels don't overwhelm the chart.

```
pivot.index = [w[:10] for w in pivot.index]
```

Optionally trim to the markets where at least two flagged accounts traded, so the heatmap isn't mostly empty columns.

```
active_markets = (pivot != 0).sum(axis=0)
pivot = pivot.loc[:, active_markets >= 2]
```

Trim market ID labels similarly.

```
pivot.columns = [str(m)[:12] for m in pivot.columns]
```

Now plot it.

```
fig, ax = plt.subplots(figsize=(14, 8))

sns.heatmap(
    pivot,
    cmap="RdYlGn",
    center=0,
    linewidths=0.5,
    linecolor="#333333",
    ax=ax,
    cbar_kws={"label": "Net P&L"}
)

ax.set_title("Co-Trading Heatmap: Flagged Accounts by Market")
ax.set_xlabel("Market ID")
ax.set_ylabel("Wallet")

plt.tight_layout()
plt.savefig("cotrading_heatmap.png", dpi=150)
plt.show()
```

What you're looking for is vertical banding: multiple flagged accounts all showing positive P&L on the same markets. If several high-risk accounts have deep green cells in the same columns, they are winning on the same events. That's the co-trading pattern made visible. If the heatmap is random and the green cells don't line up, the accounts may have been flagged for individual behavioral reasons rather than shared event exposure, which is a qualitatively different finding.

You can run a hierarchical clustering pass on the rows and columns before plotting to reorder them so accounts with similar patterns end up adjacent. Seaborn's `clustermap` handles this automatically.

```
g = sns.clustermap(
    pivot,
    cmap="RdYlGn",
    center=0,
    linewidths=0.5,
    linecolor="#333333",
    figsize=(14, 10),
    cbar_kws={"label": "Net P&L"}
)

g.savefig("cotrading_clustermap.png", dpi=150)
```

The clustermap reorders rows and columns by similarity, which often makes the banding pattern more visible than the raw pivot table.

Timeline Plots

The wallet graph shows who. The win-rate chart shows how extreme. The heatmap shows which markets. The timeline plot shows when, which is often the most damning dimension. An account that enters a position on a geopolitical event 18 hours before

resolution, at a moment when the market probability still shows significant uncertainty, and then wins, has a different profile from an account that enters early in the market's lifecycle when the uncertainty is genuine.

You built entry timing features in Chapter 4: hours before resolution, fraction of the market's lifetime elapsed at entry. The timeline plot makes those timing patterns spatial.

For each flagged account, plot their trades as points on a timeline where the x-axis is hours before resolution and the y-axis is the account (or a normalized profit-loss value), colored by outcome.

```
flagged_trades = trades[trades["wallet"].isin(flagged_wallets)].copy()
flagged_trades["wallet_short"] = flagged_trades["wallet"].str[:10]
flagged_trades["outcome_color"] = flagged_trades["profit_loss"].apply(
    lambda x: "#4caf50" if x > 0 else "#e84040"
)
```

```
fig, ax = plt.subplots(figsize=(12, 6))

for _, row in flagged_trades.iterrows():
    ax.scatter(
        row["hours_before_resolution"],
        row["wallet_short"],
        color=row["outcome_color"],
        alpha=0.7,
        s=abs(row["profit_loss"]) * 0.5 + 20,
        edgecolors="none"
    )

ax.invert_xaxis()
ax.set_xlabel("Hours Before Resolution")
ax.set_ylabel("Wallet")
ax.set_title("Trade Entry Timing: Flagged Accounts")
ax.axvline(x=24, color="white", linestyle="--", alpha=0.5, label="24h before
resolution")

win_patch = mpatches.Patch(color="#4caf50", label="win")
loss_patch = mpatches.Patch(color="#e84040", label="loss")
ax.legend(handles=[win_patch, loss_patch])

plt.tight_layout()
plt.savefig("entry_timing.png", dpi=150)
plt.show()
```

Point size maps to position size so large winning bets close to resolution are visually prominent. What you want to see is a cluster of large green dots concentrated in the right portion of the chart: late entries, large sizes, winning outcomes. That pattern, repeated across multiple accounts on the same events, is the timing signature of an information advantage.

If the winning trades are evenly distributed across the timeline, the accounts may simply be skilled at reading market trends rather than acting on non-public information. That distinction matters, and the timeline is the cleanest way to see it.

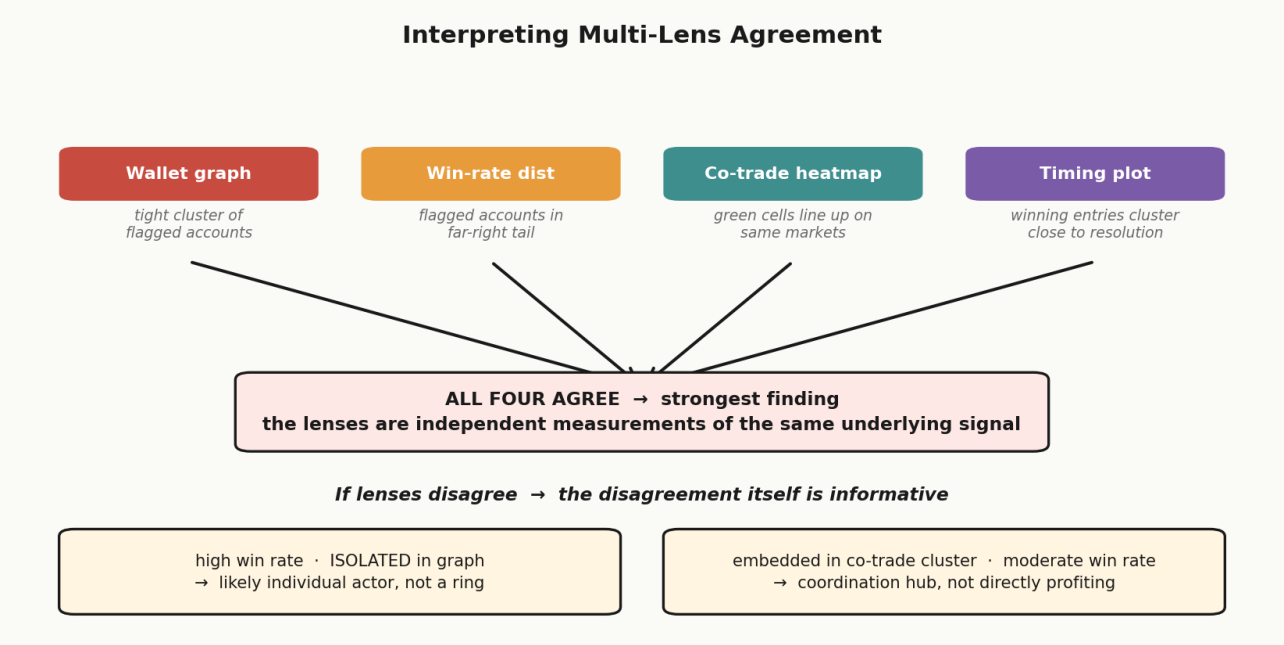
You can facet this plot by market to show that the late-entry, high-win pattern recurs across different events rather than being driven by a single lucky trade. Use matplotlib's subplot grid or seaborn's `FacetGrid` with the market ID as the faceting variable.

```
markets_to_show = flagged_trades["market_id"].value_counts().head(6).index.tolist()
subset = flagged_trades[flagged_trades["market_id"].isin(markets_to_show)]

g = sns.FacetGrid(subset, col="market_id", col_wrap=3, height=4, sharey=False)
g.map_dataframe(
    lambda data, **kwargs: plt.scatter(
        data["hours_before_resolution"],
        data["profit_loss"],
        c=data["outcome_color"],
        s=data["profit_loss"].abs() * 0.5 + 20,
        alpha=0.7
    )
)
g.set_axis_labels("Hours Before Resolution", "P&L")
g.set_titles(col_template="Market {col_name}")
plt.savefig("timing_by_market.png", dpi=150)
plt.show()
```

Putting the Visualizations Together

Each of the four visualizations is a lens on the same underlying signal. The wallet graph tells you about network structure. The win-rate distribution tells you about statistical extremity. The co-trading heatmap tells you about event overlap. The timeline tells you about temporal proximity to information.

A finding is strongest when all four lenses agree: the flagged accounts sit in a tight cluster in the wallet graph, their win rates are in the far right tail of the distribution, they share deep green cells on the same markets in the heatmap, and their winning trades are concentrated close to resolution with large position sizes.



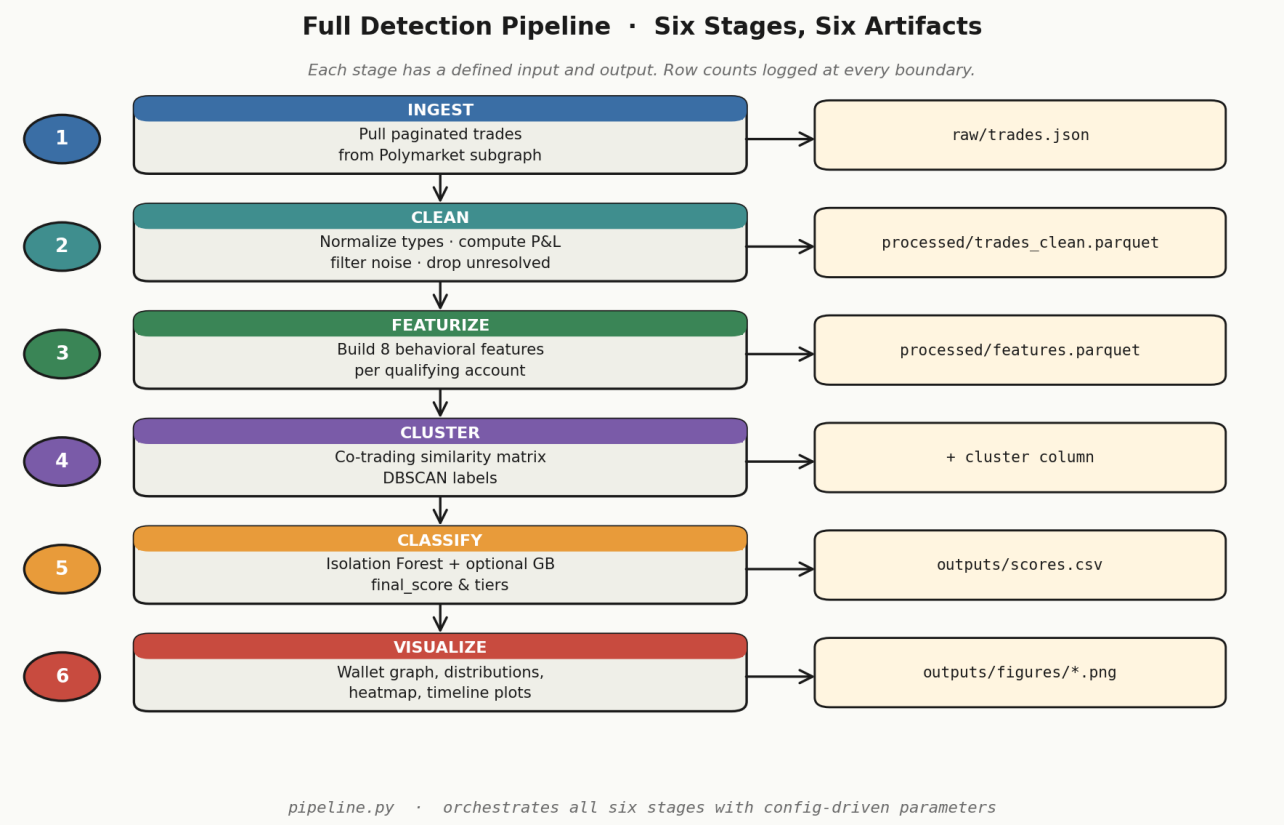
When the lenses disagree, you've found something to investigate. An account that scores high on win rate but sits isolated in the wallet graph might be an individual actor rather than part of a coordinated ring. An account that is deeply embedded in the co-trading cluster but shows a moderate win rate might be a wallet that helps coordinate without itself taking large directional positions. These are different threat models, and the visual layer is where they become distinguishable.

Save all visualizations to disk alongside the scored accounts CSV. You now have everything you need for the final chapter: a working pipeline that ingests data, engineers features, clusters accounts, scores them, and renders the findings in a form that makes the signal legible to anyone looking at it.

Chapter 8 — Putting It Together

You have all the pieces. What remains is understanding how they connect, where they can break, and how to run them as a single coherent system rather than a collection of scripts you stitch together by hand each time. This chapter walks through the full pipeline from first query to final output, treats a realistic working example, and covers the failure modes that will eventually find you.

Let's start with the architecture and then run through it.



The pipeline has six stages. Each stage takes a well-defined input and produces a well-defined output, and each output becomes the next stage's input. That discipline is not incidental. Prediction market data is messy at the source, and if you let stage boundaries blur, bugs travel silently downstream and the final scores become impossible to trace back to the data that produced them.

The six stages are:

1. **Ingest**: pull raw trade data from the Polymarket subgraph and store it.
2. **Clean**: resolve addresses, compute P&L per trade, filter noise, emit a normalized trades table.
3. **Featurize**: compute per-account behavioral features and emit a feature matrix.
4. **Cluster**: run DBSCAN on the co-trading similarity matrix, attach cluster labels to accounts.

5. **Classify**: run isolation forest plus any gradient-boosted model on the feature matrix, emit risk scores.
6. **Visualize**: produce the four visual outputs and save them alongside the scored CSV.

You can run these in sequence with a single orchestration script, or you can treat them as discrete jobs and run them on a schedule. For a first implementation, sequential is fine.

Start with a project structure like this:

```
polymarket_detection/
    data/
        raw/
        processed/
    outputs/
        figures/
        scores/
    src/
        ingest.py
        clean.py
        features.py
        cluster.py
        classify.py
        visualize.py
        pipeline.py
    config.py
```

The config file holds all tunable parameters in one place: API endpoints, market ID lists, the minimum trade count threshold for inclusion, DBSCAN epsilon and min_samples, isolation forest contamination rate, and output paths. Never hardcode these inside stage scripts. When you return to this pipeline six months later and need to adjust a threshold, you want one file to touch.

Here is a minimal config:

```python
# config.py

SUBGRAPH_URL = "https://api.thegraph.com/subgraphs/name/polymarket/matic"
MARKET_IDS = [
    "0xabc123...",
    "0xdef456...",
    # add geopolitical market IDs here
]
MIN_TRADES = 5
DBSCAN_EPS = 0.35
DBSCAN_MIN_SAMPLES = 2
ISO_FOREST_CONTAMINATION = 0.05
RAW_DATA_PATH = "data/raw/trades.json"
PROCESSED_PATH = "data/processed/trades_clean.parquet"
FEATURES_PATH = "data/processed/features.parquet"
SCORES_PATH = "outputs/scores/scored_accounts.csv"
FIGURES_DIR = "outputs/figures/"
```

Pull those values into every downstream script with a simple import. That alone will save you a painful afternoon debugging a run where half your scripts used one epsilon value and half used another.

Stage one: ingestion. The ingest script from Chapter 2 pulls trades paginated through the GraphQL endpoint. For the full pipeline run, you want it to either fetch fresh data or detect that a recent fetch already exists and skip. Add a simple freshness check:

```
import os, json, time
from config import RAW_DATA_PATH

def data_is_fresh(path, max_age_hours=6):
    if not os.path.exists(path):
        return False
    age = (time.time() - os.path.getmtime(path)) / 3600
    return age < max_age_hours

def run_ingest():
    if data_is_fresh(RAW_DATA_PATH):
        print("Raw data is fresh, skipping ingest.")
        return
    trades = fetch_all_trades()  # your paginated fetch from Chapter 2
    with open(RAW_DATA_PATH, "w") as f:
        json.dump(trades, f)
    print(f"Ingested {len(trades)} trades.")
```

This keeps pipeline reruns fast during development, when you are iterating on downstream stages and do not want to hammer the API each time.

Stage two: cleaning. Load the raw JSON, apply the normalization logic from Chapter 3, and write a parquet file. Parquet is better than CSV here because it preserves numeric types and reads significantly faster when your trade table grows large.

```
import pandas as pd
from config import RAW_DATA_PATH, PROCESSED_PATH

def run_clean():
    with open(RAW_DATA_PATH) as f:
        raw = json.load(f)
    df = normalize_trades(raw)  # your cleaning function from Chapter 3
    df = compute_pnl(df)
    df = filter_noise_trades(df)
    df.to_parquet(PROCESSED_PATH, index=False)
    print(f"Cleaned dataset: {len(df)} trades, {df['wallet'].nunique()} wallets.")
```

The print statement is not cosmetic. Run the pipeline with logging at each stage boundary and you will know immediately which stage produced a row count you did not expect. A sudden drop from 40,000 trades to 4,000 after cleaning is either correct or a bug, and you want to see it before it silently infects your feature matrix.

Stage three: featurization. Load the cleaned parquet, compute the per-account features from Chapter 4, and write the feature matrix.

```
import pandas as pd
from config import PROCESSED_PATH, FEATURES_PATH, MIN_TRADES

def run_features():
    df = pd.read_parquet(PROCESSED_PATH)
    features = build_feature_matrix(df, min_trades=MIN_TRADES)
    features.to_parquet(FEATURES_PATH, index=False)
    print(f"Feature matrix: {len(features)} accounts.")
```

The feature matrix should have one row per account that passed the minimum trade threshold. Every feature value should be numeric and finite. Before writing the file, add a validation step:

```
assert features.isnull().sum().sum() == 0, "NaN values in feature matrix"
assert (features == float('inf')).sum().sum() == 0, "Inf values in feature matrix"
```

These assertions will fail loudly rather than let a malformed feature matrix propagate into your models and produce silently wrong scores. Common sources of NaN at this stage: accounts with zero variance in position size (where you divided by std), or accounts that traded entirely in a single day (where timing features collapse). Handle those edge cases in the feature builder before they reach this check.

Stage four: clustering. Load the feature matrix, build the co-trading similarity matrix, run DBSCAN, and attach cluster labels to the feature frame.

```
import pandas as pd
from config import FEATURES_PATH, DBSCAN_EPS, DBSCAN_MIN_SAMPLES

def run_cluster():
    features = pd.read_parquet(FEATURES_PATH)
    sim_matrix = build_cotrade_similarity(features)  # from Chapter 5
    labels = run_dbscan(sim_matrix, eps=DBSCAN_EPS, min_samples=DBSCAN_MIN_SAMPLES)
    features["cluster"] = labels
    features.to_parquet(FEATURES_PATH, index=False)
    n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
    n_noise = list(labels).count(-1)
    print(f"Clustering: {n_clusters} clusters, {n_noise} noise points.")
```

Overwriting the features file with the cluster column attached keeps the schema clean for the classifier stage, which needs both the feature columns and the cluster label as input. You could instead write a separate file; either approach works as long as you are consistent.

Stage five: classification. Load the augmented feature matrix, run the anomaly detection and scoring logic from Chapter 6, and write the scored accounts CSV.

```
import pandas as pd
from config import FEATURES_PATH, SCORES_PATH, ISO_FOREST_CONTAMINATION

def run_classify():
    features = pd.read_parquet(FEATURES_PATH)
    scores = score_accounts(features, contamination=ISO_FOREST_CONTAMINATION)
    scores.to_csv(SCORES_PATH, index=False)
    flagged = (scores["risk_score"] >= 0.7).sum()
    print(f"Scored {len(scores)} accounts. {flagged} flagged above 0.7 threshold.")
```

Stage six: visualization. Load the scored accounts and the cleaned trade data, and generate all four visual outputs from Chapter 7.

```
import pandas as pd
from config import SCORES_PATH, PROCESSED_PATH, FIGURES_DIR

def run_visualize():
    scores = pd.read_csv(SCORES_PATH)
    trades = pd.read_parquet(PROCESSED_PATH)
    generate_wallet_graph(scores, output_dir=FIGURES_DIR)
    generate_winrate_distribution(scores, output_dir=FIGURES_DIR)
    generate_cotrade_heatmap(scores, trades, output_dir=FIGURES_DIR)
    generate_timeline_plot(scores, trades, output_dir=FIGURES_DIR)
    print(f"Visualizations saved to {FIGURES_DIR}.")
```

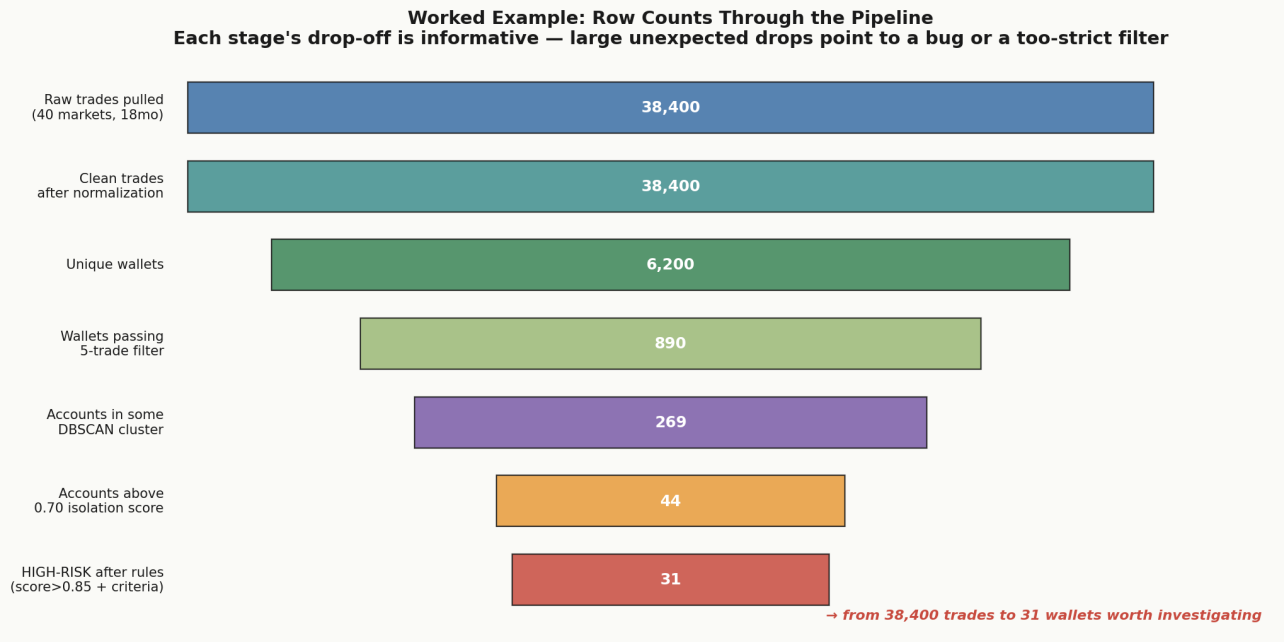
The orchestration script ties it together:

```
# pipeline.py

from src.ingest import run_ingest
from src.clean import run_clean
from src.features import run_features
from src.cluster import run_cluster
from src.classify import run_classify
from src.visualize import run_visualize

def run_pipeline():
    print("Stage 1: Ingest")
    run_ingest()
    print("Stage 2: Clean")
    run_clean()
    print("Stage 3: Featurize")
    run_features()
    print("Stage 4: Cluster")
    run_cluster()
    print("Stage 5: Classify")
    run_classify()
    print("Stage 6: Visualize")
    run_visualize()
    print("Pipeline complete.")

if __name__ == "__main__":
    run_pipeline()
```

Run `python pipeline.py` and the full detection pass executes from raw API data to visual outputs.

A worked example

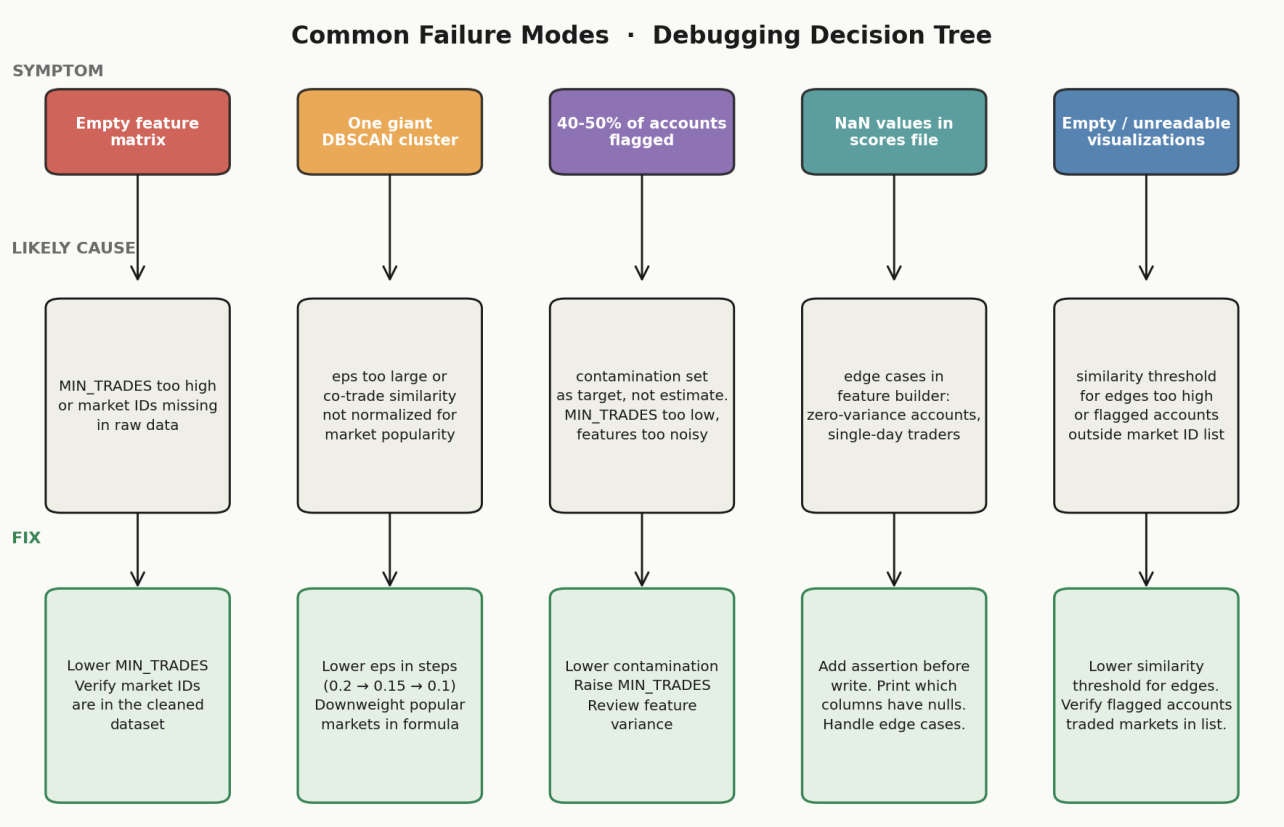
Now for the worked example. Suppose you have pulled trade data across 40 geopolitical markets over an 18-month window. After cleaning, you have 38,400 trades from 6,200 unique wallets. After applying the minimum 5-trade filter, 890 wallets remain in the feature matrix. DBSCAN at epsilon 0.35 with `min_samples` 2 returns 14 clusters and 621 noise points, meaning 269 accounts are in some cluster. The isolation forest scores 44 accounts above the 0.7 risk threshold.



Of those 44 accounts, the wallet graph shows that 31 sit in three tight clusters. The remaining 13 are embedded in larger, looser clusters or are near-isolates. The win-rate distribution shows that 9 of the 44 have win rates above 88 percent across at least 10 trades, which sits well into the right tail. The heatmap shows that the three tight clusters each concentrate their trading on 3 to 5 markets and those markets are largely non-overlapping across clusters, suggesting three separate information networks rather than one. The timeline plots for the highest-scoring accounts show winning trades entered 4 to 12 hours before resolution on events where public information was thin until the last few hours.

This is the pattern you are looking for. Multiple lenses agree. The accounts are not randomly distributed in feature space. They are not winning on a wide range of markets by chance. They are winning on specific markets, with each other, at times that precede the information becoming public.

Common failure modes



The first is an empty or near-empty feature matrix. This usually means the market ID filter in ingest is too narrow, or the `MIN_TRADES` threshold is set too high. Check the row count logged after cleaning. If you have 38,000 trades but only 12 accounts pass the minimum trades filter, your filter is stripping out almost everything. Lower the threshold or verify your market IDs are actually present in the raw data by inspecting the first few records directly.

The second is DBSCAN returning a single giant cluster. This means your epsilon is too large. The similarity matrix has all values near 1, and DBSCAN connects everything. Lower epsilon in small steps (try 0.2, then 0.15) and watch what happens to the cluster count. If even epsilon 0.1 returns one large cluster, the problem is upstream in how the similarity matrix is constructed. Check whether your co-trading matrix normalizes for market popularity. If 90 percent of accounts traded on the same two high-volume markets, co-trading on those markets is not informative and should be downweighted or excluded.

The third is isolation forest flagging 40 to 50 percent of accounts. The contamination parameter is supposed to represent the expected fraction of anomalies, not a target. If you set contamination to 0.05 and still get scores clustered near 0.7 for a large fraction of accounts, your feature matrix has low variance. Many accounts look similar. This often happens when the minimum trade threshold is too low and you are including accounts with only 5 or 6 trades, whose feature values are high-variance and noisy. Raise the threshold and rerun.

The fourth is NaN values appearing in the scores file. This means a feature was not properly handled before being passed to the model. Revisit the assertion in the featurization stage. Add verbose logging there to print which columns contain null values. Common culprits are timing features for accounts whose only trades occurred on a single day, or position size variance for accounts with perfectly uniform bet sizes.

The fifth is visualizations that are empty or unreadable. Wallet graphs with zero edges mean the cosine similarity threshold for edge inclusion is too high. Lower it. Heatmaps with all white cells mean no co-trading was detected for the flagged accounts, which in turn usually means the flagged accounts traded on markets not in your market ID list. Check that your market IDs cover the same set of markets represented in the flagged accounts' trade history.

Extensions worth building once the base pipeline is stable

Temporal tracking. The current pipeline treats the full 18-month window as a single snapshot. But coordinated networks evolve. Wallets rotate out, new ones appear, cluster compositions shift. To track this, slice the data into rolling windows (say, 60-day windows with 30-day steps) and run the full pipeline on each window independently. Store cluster membership per window, then look for accounts that consistently appear in flagged clusters across windows. Accounts that are transiently suspicious are less interesting than accounts that appear in a different coordinated cluster every election cycle.

New wallet alerting. Add a stage after featurization that flags accounts with wallet age under 30 days that already have win rates above 80 percent on geopolitical markets. Fresh wallets with anomalous early performance are a specific sub-pattern and worth tracking separately from mature wallets with long histories. You can build a simple filter on the feature matrix before clustering runs.

Cross-market linking. The current pipeline looks for co-trading on the same market. A more sophisticated extension looks for accounts that trade on different markets but with correlated timing and outcome patterns, suggesting access to a single information source that covers multiple domains. This requires computing account-level correlations on trade timing rather than co-occurrence on shared markets, which is a heavier computation but surfaces a different class of actor.

Network centrality features. In the wallet graph, accounts with high betweenness centrality might not themselves be high-win-rate traders. They might be coordination hubs. A wallet that bridges three sub-clusters but takes small positions might be facilitating information sharing rather than directly profiting from it. Adding network centrality as a feature in the classification stage catches this pattern.

Labeling via external events. If you have access to post-hoc reporting on specific geopolitical events where insider information was known to be in play (news reports, legal filings, public disclosures), you can use those events to build a small labeled dataset. A handful of confirmed cases is enough to tune your classifier thresholds and

evaluate false positive rates. Without any labels, the pipeline operates entirely as anomaly detection and you cannot know your false positive rate, only that the flagged accounts are statistically unusual.

A note on what the pipeline cannot tell you

Statistical anomaly is not legal guilt. An account that wins 92 percent of its geopolitical trades and co-trades with five other accounts that also win 92 percent is doing something that deserves scrutiny. It is not, on its own, proof of anything. The pipeline's job is to surface the accounts that warrant deeper investigation, not to render verdicts. The output is a sorted list of accounts, a set of visualizations, and a body of evidence that a human investigator can use to decide what to look at more closely.

That framing matters when you decide how to act on the output. The scored accounts CSV is not a blacklist. It is a ranked queue. The accounts at the top of the queue have the highest combination of anomalous feature values, cluster membership, and timing patterns. Some of them will turn out to have benign explanations. Some will not. The pipeline cannot tell the difference. A human with context about the specific events, the timing of relevant information releases, and the structure of the networks involved can.

What the pipeline gives you is the starting point for that investigation, surfaced from data that no human reviewer could have scanned by hand.

Every trade on Polymarket leaves a trace. This course started with that observation, and the pipeline you have just built is a systematic way to follow those traces to their source. The stack is not complicated. The ideas behind it are not mysterious. Prediction market data is structured, the behavioral fingerprints of informed trading are consistent, and the gap between raw GraphQL responses and a ranked list of suspicious accounts turns out to be a few hundred lines of Python and a clear head about what you are actually measuring.

The signal is there. You now have the tools to find it.