Project overview

Goal

A prediction-market “movers” platform that:

- continuously ingests prices/market metadata from **Polymarket + Kalshi**
- stores normalized time-series snapshots
- computes “top movers” across time windows (5m/15m/1h/24h)
- serves an auto-refreshing Streamlit dashboard with:
    - Top Movers table
    - Category movers charts (Politics / Sports / Crypto / etc.)
    - Market detail page with price chart
    - Alerts log + on-screen toasts

Non-goals (initially)

- automatic bet execution
- complicated multi-user auth
- heavy real-time order-book visualization (can add later)

High-level architecture (data flow)

Collectors (always-on)

1. Fetch market list + metadata (slow cadence)
2. Fetch latest prices (fast cadence)
3. Normalize to a shared schema
4. Write to DB (append-only snapshots)

Analytics

- Compute movers on demand (simple) OR
- Precompute movers every minute (fast UI + consistent alerts)

Streamlit

- Reads precomputed movers + latest snapshots
- Auto-refreshes
- Shows charts + tables
- Displays in-app notifications (toasts) based on “new alerts” rows

---

Components

1) `collector` service (background daemon)

Responsibilities

- Pull data from Polymarket + Kalshi on schedules
- Retry/backoff on failures
- Respect rate limits
- Normalize raw data → canonical schema
- Persist to DB

Key design choice

- Separate “market metadata” from “price snapshots”
    - metadata updates less frequently (e.g., every 10–30 min)
    - snapshots update frequently (e.g., every 10–60 sec depending on scale)

Suggested scheduling

- `sync_markets`: every 15 minutes
- `sync_prices`: every 15 seconds–60 seconds
- `rollups`: every 1 minute (optional OHLC / precomputes)
- `alerts`: every 15–60 seconds (optional backend alerts)

2) `db` (Postgres recommended; SQLite for local MVP)

Optional later: TimescaleDB extension for rollups, retention, downsampling.

3) `analytics` layer (shared library)

A Python package that provides:

- movement calculations (Δpp over windows)
- filters (min volume, max spread)
- category aggregation
- ranking logic (e.g., abs move weighted by liquidity)

This should be **pure functions + SQL queries**, not Streamlit-specific.

4) `dashboard` (Streamlit app)

Responsibilities

- query DB for:
  - top movers per window
  - category movers
  - market detail series
  - latest alerts
- render UI (tables + charts)
- auto-refresh every N seconds
- show toasts for newly triggered alerts

5) `notifier` (optional, later)

Even if Streamlit is primary, we can add “secondary channels” later:

- email / Slack / Discord
- a daily digest
- push notifications (with a component)
- Twitter / X

Architecturally: notifier reads from the same **alerts table** and sends externally.

Repo structure (recommended)

```
prediction-movers/
  README.md
  pyproject.toml
  .env.example
  docker-compose.yml

  apps/
    collector/
      main.py                # entrypoint (runs schedulers)
      jobs/
        polymarket_sync.py      # sync_markets, sync_prices
        kalshi_sync.py
        rollups.py             # optional OHLC aggregation
        alerts.py              # optional alert evaluation
      adapters/
        polymarket.py          # API client wrapper
```

```
    kalshi.py
  normalization/
    polymarket_norm.py      # raw -> canonical
    kalshi_norm.py
  settings.py             # env config
  logging.py
  metrics.py              # optional

 dashboard/
  app.py                  # Streamlit entry
  pages/
    1_Top_Movers.py
    2_Category_Trends.py
    3_Market_Detail.py
    4_Alerts_Log.py
  components/
    filters.py
    charts.py
    tables.py
  settings.py
  data_access.py          # DB query functions

packages/
 core/
  __init__.py
  models.py               # pydantic/dataclasses: CanonicalMarket, CanonicalSnapshot
  analytics/
    movers.py             # compute deltas, ranking
    categories.py
    scoring.py            # liquidity-weighting, volatility filters
  storage/
    db.py                 # connection helpers
    queries.py            # shared SQL
  utils/
    time.py
    retry.py
    dedupe.py

migrations/
 001_init.sql
 002_indexes.sql
 003_rollups.sql

tests/
```

test_normalization.py
test_movers.py
test_queries.py

Why this split works

- `packages/core` becomes the “brain” shared by both collector + dashboard.
- We can later add a FastAPI service without changing core logic.

Database schema (canonical)

We want **raw-ish** + **normalized** + **computed**.

Core tables

`markets`

- `market_id` (PK) – canonical ID we define (e.g., `POLY:<id>` / `KALSHI:<ticker>`)
- `source` (`polymarket`, `kalshi`)
- `source_market_id` (original platform ID/ticker)
- `title`
- `category` (Politics, Sports, Crypto, …)
- `status` (open/closed/resolved)
- `url`
- `created_at`, `updated_at`

`market_tokens` (for YES/NO or multi-outcome)

- `token_id` (PK) – canonical token/outcome ID
- `market_id` (FK)
- `outcome` (YES/NO or label)
- `symbol` / `source_token_id`

`snapshots` (append-only time series)

- `ts` (timestamp, indexed)
- `token_id` (FK, indexed)
- `price` (float) – implied probability proxy
- `bid`, `ask` (float, optional)

- `spread` (float, computed or stored)
- `volume_24h` / `liquidity` (optional if available)
- `source_latency_ms` (optional)
- PRIMARY KEY: (`token_id`, `ts`) or a surrogate id + index

Computed tables (recommended for speed)

`movers_cache` (precomputed movers for standard windows)

- `as_of_ts`
- `window` (e.g. 300, 900, 3600, 86400 seconds)
- `token_id`
- `price_now`, `price_then`
- `move_pp`, `abs_move_pp`
- `rank`
- indexed on (`as_of_ts`, `window`, `rank`)

`alerts`

- `id` (PK)
- `created_at`
- `token_id`
- `window`
- `move_pp`
- `threshold_pp`
- `reason` (e.g., “abs_move_pp >= threshold AND volume >= min_volume”)
- `acknowledged_at` (nullable)

Optional rollups

`ohlc_1m`, `ohlc_5m`, etc.

- `bucket_ts`
- `token_id`
- `open`, `high`, `low`, `close`
- `volume_proxy` (if available)

Movement & ranking logic (how “top movers” stays useful)

A solid default:

- Movement: `move_pp = (price_now - price_then) * 100`
- Rank by `abs(move_pp)` but filter noise:
  - `min_volume_24h` (or liquidity)
  - `max_spread_pp`
  - `market_status == open`
- Optional “quality score”:
  - `score = abs_move_pp * log1p(volume_24h)`
    This prevents illiquid micro markets from dominating.

Runtime model

Local dev (fastest)

- SQLite DB file
- Run collector in one terminal
- Run Streamlit in another

Production (recommended)

Docker Compose with:

- `postgres`
- `collector`
- `dashboard`

Scaling knobs

- If we track *lots* of markets, batch price calls and/or shard collectors:
  - collector A: politics + world
  - collector B: sports
  - collector C: crypto, etc.

Configuration & secrets

Use `.env` (never commit real keys):

- `DB_URL=postgresql+psycopg://...`
- `POLYMARKET_API_*` (if needed)
- `KALSHI_API_KEY=...`
- `COLLECTOR_PRICE_INTERVAL=15`
- `COLLECTOR_MARKET_INTERVAL=900`
- `ALERT_THRESHOLD_PP=6`
- `MIN_VOLUME_24H=10000`
- `MAX_SPREAD_PP=2`

Observability & reliability

Logging

- structured logs (JSON) with job name, duration, counts ingested

Metrics (optional but nice)

- ingested snapshots per run
- API errors / retries
- lag between `now()` and latest `snapshot.ts`

Data quality checks

- drop/flag snapshots with missing/invalid prices
- detect “stale markets” (no updates in X minutes)

Retention

- keep 15-second snapshots for 7–14 days
- keep 1-minute OHLC for 6–12 months (cheap + great for charts)

Streamlit UX plan

Pages

1. **Top Movers**
    - window selector
    - filters (category, volume, spread)
    - table with link-out
2. **Category Trends**
    - bar chart: avg abs move by category
    - top movers within selected category
3. **Market Detail**
    - time-series chart (last 24h / 7d)
    - move stats + volume/spread
4. **Alerts**
    - newest alerts + “acknowledge” button
    - toasts for unacknowledged alerts

Auto-refresh

- 5–15 seconds in dashboard (since DB reads are cheap)

Implementation roadmap (practical)

Phase 1 (MVP)

- Postgres + migrations
- collector: sync_markets + sync_prices (store snapshots)
- dashboard: Top Movers + Market Detail (compute movers on the fly)

Phase 2 (performance + polish)

- movers_cache job (precompute windows)
- alerts job + alerts table
- category analytics page

Phase 3 (scale + nice-to-haves)

- OHLC rollups + retention policy
- external notifications (Discord/Telegram/X as optional)
- basic auth for dashboard (if hosted publicly)