

Kalshi Quant Trading System - Complete Project Documentation

Table of Contents

1. [Project Overview](#project-overview)
2. [System Architecture](#system-architecture)
3. [Project Structure](#project-structure)
4. [Setup Instructions](#setup-instructions)
5. [Configuration](#configuration)
6. [Component Documentation](#component-documentation)
7. [Data Flow](#data-flow)
8. [Running the System](#running-the-system)
9. [Key Concepts Explained](#key-concepts-explained)
10. [Testing](#testing)
11. [Troubleshooting](#troubleshooting)

Project Overview

Purpose

The Kalshi Quant Trading System is an automated data ingestion platform designed to:

- Continuously collect orderbook data from Kalshi prediction markets
- Focus specifically on Federal Reserve interest rate decision markets (KXFEDDECISION series)
- Store historical orderbook snapshots in DuckDB for quantitative analysis
- Build a dataset of bid/ask spreads and pricing data for trading strategy development

What This System Does

1. ****Connects to Kalshi API****: Authenticates using RSA private key authentication
2. ****Scans Markets****: Finds the next 4 upcoming Fed meetings sorted by date
3. ****Fetches Orderbooks****: Retrieves live bid/ask prices for each market
4. ****Calculates Spreads****: Uses "Implied Ask" rule to derive ask prices when not provided
5. ****Stores Data****: Saves snapshots to DuckDB database every 60 seconds
6. ****Runs Continuously****: Operates in a loop until manually stopped

Key Features

- ****Low Memory Usage****: Opens/closes database connections per write operation
- ****Fault Tolerant****: Continues running even if individual API calls fail

- **Date-Based Sorting**: Intelligently parses and sorts markets by meeting dates
- **Volume Filtering**: Only tracks markets with sufficient liquidity (\$1000+ daily volume)
- **Modular Architecture**: Well-organized codebase with clear separation of concerns

System Architecture

High-Level Design

...



```
| market_data. |  
| duckdb       |  
└────────────────┘
```

...

Component Interactions

1. ****data_ingest.py**** (Main Loop)

- Initializes Kalshi client and authenticates
- Runs every 60 seconds
- Calls MarketScanner to get markets
- Calls DatabaseManager to store snapshots

2. ****MarketScanner**** (Market Discovery & Data Retrieval)

- Fetches markets from Kalshi API with pagination
- Filters by volume threshold
- Uses date parser to sort markets chronologically
- Retrieves orderbook data for each market

3. ****market_date_parser**** (Date Extraction & Sorting)

- Extracts dates from ticker strings (e.g., "KXFEDDECISION-26JAN-H0" → Jan 26, 2026)
- Sorts markets by meeting date (earliest first)

4. ****orderbook_parser**** (Price Extraction)

- Extracts best bid/ask prices from orderbook data
- Implements "Implied Ask" rule for binary markets
- Calculates spreads

5. ****DatabaseManager**** (Data Persistence)

- Manages DuckDB connections
- Creates schema on initialization
- Provides safe insert methods for low-memory operation

Project Structure

```

'''
Kalshi_Quant/
├── config/
│   ├── __init__.py
│   └── settings.py      # Configuration constants
├── database/
│   ├── __init__.py
│   └── db_manager.py    # DuckDB connection and operations
├── ingestion/
│   ├── __init__.py
│   ├── market_scanner.py  # Market discovery and orderbook retrieval
│   ├── market_date_parser.py # Date parsing from tickers
│   └── orderbook_parser.py # Price extraction from orderbooks
├── models/
│   ├── __init__.py
│   └── market_data.py     # Pydantic models for data validation
├── utils/
│   ├── __init__.py
│   └── auth.py           # RSA private key loading
├── venv/                 # Python virtual environment
├── connect_and_price.py  # Standalone testing script
├── main.py               # Original one-time ingestion script
├── data_ingest.py        # Continuous ingestion service (MAIN)
├── requirements.txt      # Python dependencies
├── .gitignore            # Git ignore patterns
└── PROJECT_DOCUMENTATION.md # This file
'''

```

File Descriptions

Configuration Files

***** config/settings.py *****

- Central configuration file
- Contains API credentials, file paths, thresholds
- All constants used across the system

requirements.txt

- Python package dependencies
- Used for `pip install -r requirements.txt`

Core Modules

utils/auth.py

- `load_private_key_pem()`: Reads RSA private key from PEM file
- Handles file not found and parsing errors

models/market_data.py

- `MarketPricing`: Pydantic model for bid/ask prices and spreads
- `OrderbookSnapshot`: Complete snapshot record for database storage
- Data validation and type safety

database/db_manager.py

- `DatabaseManager` class: Manages DuckDB operations
- `initialize_schema()`: Creates `orderbook_snapshots` table
- `insert_snapshot()`: Insert with persistent connection
- `insert_snapshot_safe()`: Insert with per-write connection (low memory)
- `insert_snapshots_batch()`: Batch insert for efficiency
- Context manager support (`with DatabaseManager(...)`)

ingestion/market_date_parser.py

- `parse_ticker_date()`: Extracts date from ticker string
 - Input: "KXFEDDECISION-26JAN-H0"
 - Output: `datetime(2026, 1, 26)`
- `sort_markets_by_date()`: Sorts markets chronologically
- Handles invalid dates gracefully

ingestion/orderbook_parser.py

- `extract_orderbook_prices()`: Extracts best bid/ask from orderbook data
- Implements "Implied Ask" rule:
 - Yes_Ask = 100 - Best_No_Bid
 - No_Ask = 100 - Best_Yes_Bid
- Returns `MarketPricing` object with spreads calculated

ingestion/market_scanner.py

- `MarketScanner` class: Main market discovery and data retrieval
- `scan_series_markets()`: Fetches all markets in a series with pagination
- `get_next_n_meetings()`: Gets next N Fed meetings sorted by date
- `get_orderbook_snapshot()`: Retrieves full orderbook data for a market
- `get_market_metadata()`: Fetches market details
- Handles API errors and Pydantic validation issues

Main Scripts

connect_and_price.py

- Standalone testing/validation script
- Tests API connection and pricing logic
- Displays orderbook data for all qualifying markets
- Useful for debugging and validation

main.py

- Original one-time ingestion script
- Uses `scan_and_store_markets()` method
- Processes all markets in series, stores in database, then exits

data_ingest.py ☆ **PRIMARY SERVICE**

- Continuous data ingestion service
- Runs every 60 seconds in a loop
- Fetches next 4 Fed meetings
- Stores snapshots using safe insert (low memory)
- Handles graceful shutdown (Ctrl+C)
- Error handling to continue on failures

Setup Instructions

Prerequisites

1. **Python 3.9+** (system uses Python 3.9.6)
2. **Kalshi Account** with API access

3. ****API Credentials****:

- API Key ID
- RSA Private Key file (PEM format)

Step-by-Step Setup

1. Clone or Create Project Directory

```
```bash
mkdir Kalshi_Quant
cd Kalshi_Quant
```
```

2. Create Virtual Environment

```
```bash
python3 -m venv venv
source venv/bin/activate # On macOS/Linux
OR
venv\Scripts\activate # On Windows
```
```

3. Install Dependencies

```
```bash
pip install -r requirements.txt
```
```

****Required packages**** (from `requirements.txt`):

- ``pydantic>=2.0.0``: Data validation and models
- ``kalshi-python-sync``: Official Kalshi Python SDK
- ``duckdb``: Embedded analytics database

4. Obtain Kalshi API Credentials

1. Log into your Kalshi account
2. Navigate to API settings

3. Generate API key pair:

- Save the **Key ID** (UUID format)
- Download the **Private Key** file (PEM format)

5. Configure Credentials

Option A: Update `config/settings.py` directly

Edit `config/settings.py`:

```
python
KEY_ID = "your-key-id-here" # Replace with your actual Key ID
KEY_FILE_PATH = Path("your_key_file.key") # Replace with your key filename
...
```

Option B: Place key file in project root

1. Place your private key file in the project root directory
2. Update `KEY_FILE_PATH` in `config/settings.py` to match the filename

6. Verify Setup

Test the connection:

```
bash
python connect_and_price.py
...
```

This should:

- Load your private key
- Authenticate with Kalshi API
- Fetch and display market data

If successful, you're ready to run the ingestion service.

Configuration

```
### Configuration File: `config/settings.py`

```python
API Authentication
KEY_ID = "0ac60c80-d575-480e-979b-aa5050a61c1b" # Your Kalshi API Key ID
KEY_FILE_PATH = Path("My_First_API_Key.key") # Path to RSA private key file

Database Configuration
DATABASE_PATH = "market_data.duckdb" # DuckDB database file path

Market Filtering
MIN_DAILY_VOLUME = 100000 # Minimum 24-hour volume in cents ($1000)

Data Ingestion Configuration
INGESTION_INTERVAL_SECONDS = 60 # How often to fetch snapshots
NUM_FED_MEETINGS = 4 # Number of upcoming Fed meetings to track
```
```

Configuration Parameters Explained

KEY_ID

- Your Kalshi API Key ID (UUID format)
- Found in Kalshi account API settings
- Required for authentication

KEY_FILE_PATH

- Path to your RSA private key file (PEM format)
- Should be in project root directory
- File should be kept secure (not committed to git)

DATABASE_PATH

- Location of DuckDB database file
- Will be created automatically if it doesn't exist
- Relative path: creates in project root

MIN_DAILY_VOLUME

- Minimum 24-hour trading volume threshold (in cents)

- Default: 100000 cents = \$1000
- Only markets meeting this threshold are tracked
- Filters out illiquid markets

****`INGESTION_INTERVAL_SECONDS`****

- How often the service fetches new snapshots
- Default: 60 seconds (1 minute)
- Lower = more frequent data, higher API usage
- Higher = less frequent data, lower API usage

****`NUM_FED_MEETINGS`****

- Number of upcoming Fed meetings to track
- Default: 4
- System automatically sorts by date and selects earliest N

Component Documentation

Database Schema

****Table: `orderbook_snapshots`****

| Column | Type | Description |
|----------------------|-----------|--|
| ----- ----- ----- | | |
| `snapshot_timestamp` | TIMESTAMP | When the snapshot was taken (UTC) |
| `ticker` | VARCHAR | Market ticker (e.g., "KXFEDDECISION-26JAN-H0") |
| `market_title` | VARCHAR | Full market title/description |
| `series_ticker` | VARCHAR | Series identifier (e.g., "KXFEDDECISION") |
| `best_yes_bid` | REAL | Best Yes bid price (cents) |
| `best_yes_ask` | REAL | Best Yes ask price (cents) |
| `best_no_bid` | REAL | Best No bid price (cents) |
| `best_no_ask` | REAL | Best No ask price (cents) |
| `yes_spread` | REAL | Yes spread = yes_ask - yes_bid (cents) |
| `no_spread` | REAL | No spread = no_ask - no_bid (cents) |
| `volume_24h` | INTEGER | 24-hour trading volume (cents) |

```
**Primary Key**: (`snapshot_timestamp`, `ticker`)
```

```
### Data Models
```

```
#### `MarketPricing`
```

Pydantic model representing bid/ask prices and spreads:

```
```python
```

```
class MarketPricing(BaseModel):
```

```
 best_yes_bid: float # Best Yes bid (cents)
```

```
 best_yes_ask: Optional[float] # Best Yes ask (cents)
```

```
 best_no_bid: float # Best No bid (cents)
```

```
 best_no_ask: Optional[float] # Best No ask (cents)
```

```
 yes_spread: Optional[float] # Calculated: yes_ask - yes_bid
```

```
 no_spread: Optional[float] # Calculated: no_ask - no_bid
```

```
```
```

```
#### `OrderbookSnapshot`
```

Complete snapshot record for database storage:

```
```python
```

```
class OrderbookSnapshot(BaseModel):
```

```
 snapshot_timestamp: datetime # When snapshot was taken
```

```
 ticker: str # Market ticker
```

```
 market_title: Optional[str] # Market title
```

```
 series_ticker: Optional[str] # Series identifier
```

```
 best_yes_bid: float # Best Yes bid
```

```
 best_yes_ask: Optional[float] # Best Yes ask
```

```
 best_no_bid: float # Best No bid
```

```
 best_no_ask: Optional[float] # Best No ask
```

```
 yes_spread: Optional[float] # Yes spread
```

```
 no_spread: Optional[float] # No spread
```

```
 volume_24h: Optional[int] # 24-hour volume (cents)
```

```
```
```

```
### Key Functions
```

```
##### `parse_ticker_date(ticker: str) -> Optional[datetime]`
```

Extracts and parses date from market ticker.

****Input****: `"KXFEDDECISION-26JAN-H0"`

****Output****: `datetime(2026, 1, 26)`

****Logic****:

1. Splits ticker by `'-':` `["KXFEDDECISION", "26JAN", "H0"]`
2. Extracts date portion: `"26JAN"`
3. Uses regex to parse: `'(\d{1,2})([A-Z]{3})'` → day=26, month=JAN
4. Maps month abbreviation to number: JAN → 1
5. Determines year: current year if month hasn't passed, else next year

****Edge Cases****:

- Invalid format → returns `None`
- Invalid date (e.g., Feb 30) → returns `None`

```
##### `extract_orderbook_prices(...) -> Optional[MarketPricing]`
```

Extracts best bid/ask prices from orderbook data.

****Input****: Lists of `[price, quantity]` pairs for Yes/No bids and asks

****Logic****:

1. Extracts best Yes bid: Last element of `yes_bids` array (highest price)
2. Extracts best No bid: Last element of `no_bids` array (highest price)
3. Extracts best Yes ask: First element of `yes_asks` array (lowest price)
 - If no asks available: Calculates using Implied Ask rule
4. Extracts best No ask: First element of `no_asks` array (lowest price)
 - If no asks available: Calculates using Implied Ask rule
5. Calculates spreads: `ask - bid` for Yes and No

****Implied Ask Rule**** (for binary prediction markets):

- `Yes_Ask = 100 - Best_No_Bid`
- `No_Ask = 100 - Best_Yes_Bid`

This works because in binary markets, Yes + No must equal 100 cents.

```
#### `get_next_n_meetings(...)` -> List[Market]
```

Gets the next N Fed meetings sorted by date.

Process:

1. Calls `scan_series_markets()` to fetch all markets with volume filtering
2. Uses `sort_markets_by_date()` to sort chronologically
3. Returns first N markets (earliest dates)

Example:

```
```python
```

```
markets = scanner.get_next_n_meetings("KXFEDDECISION", n=4)
```

```
Returns: [Market(26JAN), Market(5FEB), Market(15MAR), Market(30APR)]
```

```
...
```

```

```

## Data Flow

### Complete Data Flow Diagram

```
...
```

1. data\_ingest.py starts

```
|
|----> Initialize KalshiClient with authentication
|----> Create MarketScanner(client)
|----> Create DatabaseManager(db_path)
|
|----> [LOOP: Every 60 seconds]
|
|----> MarketScanner.get_next_n_meetings()
| |
| |----> scan_series_markets() [API call]
| |
| |----> Fetch markets with pagination
| |
| |----> Filter by volume >= MIN_DAILY_VOLUME
| |
| |
```



1. **Market Discovery**:
  - Call `get_next_n_meetings("KXFEDDECISION", n=4)`
  - API fetches all open KXFEDDECISION markets
  - Filter by volume threshold (\$1000+)
  - Parse dates from tickers
  - Sort by date (earliest first)
  - Return top 4 markets
2. **Data Collection**:
  - For each of the 4 markets:
    - Fetch orderbook via API
    - Extract best bid/ask prices
    - Calculate spreads
    - Create OrderbookSnapshot object
3. **Data Storage**:
  - For each snapshot:
    - Open DuckDB connection
    - Insert snapshot
    - Close connection (ensures data is persisted)
4. **Wait and Repeat**:
  - Sleep for 60 seconds
  - Repeat from step 1

---

## Running the System

### Start Continuous Ingestion Service

```
```bash
```

```
# Activate virtual environment
```

```
source venv/bin/activate
```

```
# Run the ingestion service
```

```
python data_ingest.py
```

...

****Expected Output**:**

...

=====

KALSHI QUANT - Data Ingestion Service

=====

--- 🛠️ INITIALIZING KALSHI CONNECTION ---

- ✓ Loaded private key from 'My_First_API_Key.key'
- ✓ Authentication configured
- ✓ MarketScanner initialized
- ✓ DatabaseManager initialized

--- 📄 CONFIGURATION ---

Database: market_data.duckdb

Series: KXFEDDECISION

Meetings tracked: 4

Min volume: \$1000.00

Interval: 60 seconds

--- 🚀 STARTING INGESTION LOOP ---

Press Ctrl+C to stop gracefully

=====

Iteration #1 - 2025-01-XX XX:XX:XX

=====

📊 Fetching next 4 Fed meetings...

- ✓ Found 1 markets to process

Processing: KXFEDDECISION-26JAN-H0

- ✓ Stored: Yes Bid 45.23¢ | No Bid 54.77¢

--- Iteration Summary ---

- ✓ Success: 1 snapshots stored


⌚ Waiting 60 seconds until next iteration...

...

Stop the Service

Press `Ctrl+C` for graceful shutdown:

...

 Shutdown requested...

☒ Ingestion service stopped gracefully

Total iterations: 42

...

Other Scripts

****Test Connection and Pricing Logic**:**

```
```bash
```

```
python connect_and_price.py
```

```
```
```

****One-Time Data Ingestion**:**

```
```bash
```

```
python main.py
```

```
```
```

Key Concepts Explained

Binary Prediction Markets

Kalshi markets are binary - they have two outcomes: "Yes" or "No". For example:

- Market: "Will the Fed hike rates by 0bps at their January 2026 meeting?"
- Yes contracts: Pay \$1 (100 cents) if the answer is Yes
- No contracts: Pay \$1 (100 cents) if the answer is No

****Pricing**:**

- Yes price + No price = 100 cents (always)
- If Yes is trading at 45 cents, No must be at 55 cents

Implied Ask Rule

The Kalshi API typically only returns **bids** (what buyers are willing to pay), not **asks** (what sellers are asking for).

The Rule:

- $\text{Yes_Ask} = 100 - \text{Best_No_Bid}$
- $\text{No_Ask} = 100 - \text{Best_Yes_Bid}$

Why This Works:

In binary markets, $\text{Yes} + \text{No} = 100$. If someone is bidding 55 cents for No, they're effectively offering to sell Yes for 45 cents ($100 - 55 = 45$).

Example:

- Best Yes Bid: 45 cents (buyers willing to pay 45 for Yes)
- Best No Bid: 55 cents (buyers willing to pay 55 for No)
- Implied Yes Ask: $100 - 55 = 45$ cents
- Implied No Ask: $100 - 45 = 55$ cents

Spread Calculation

Spread = Ask Price - Bid Price

Represents the cost of trading:

- **Yes Spread**: $\text{Yes_Ask} - \text{Yes_Bid}$
- **No Spread**: $\text{No_Ask} - \text{No_Bid}$

Lower spreads = more liquid markets = easier to trade

Date Parsing Logic

Market tickers encode dates: `KXFEDDECISION-26JAN-H0`

- Series: `KXFEDDECISION`
- Date: `26JAN` (26th day of January)
- Suffix: `H0` (additional identifier)

****Parsing Process**:**

1. Split by `:`: `["KXFEDDECISION", "26JAN", "H0"]`
2. Extract date part: `"26JAN"`
3. Parse with regex: day=26, month=JAN
4. Determine year:
 - If current month is January and we're past the 26th → next year
 - If current month is after January → next year
 - Otherwise → current year

Low Memory Architecture

****Problem**:** Keeping database connections open uses memory

****Solution**:** `insert_snapshot_safe()` method

- Opens connection
- Writes data
- Closes connection immediately
- Ensures data is persisted (committed to disk)
- Reduces memory footprint

****Trade-off**:** Slight performance cost (open/close overhead) vs. lower memory usage

Testing

Test Date Parser

```
```bash
python test_date_parser.py
```
```

****What it tests**:**

- Parsing various ticker formats
- Handling invalid dates
- Sorting markets chronologically

Test Market Scanner

```
```bash
```

```
python test_market_scanner.py
```

```
```
```

****What it tests**:**

- API connection
- Fetching next N meetings
- Date sorting
- Volume filtering

Test Connection and Pricing

```
```bash
```

```
python connect_and_price.py
```

```
```
```

****What it tests**:**

- Full authentication flow
- Market discovery
- Orderbook retrieval
- Price extraction
- Spread calculation

Verify Database

****Using DuckDB CLI**** (if installed):

```
```bash
```

```
duckdb market_data.duckdb
```

```
```
```

****SQL Queries**:**

```
```sql
```

```
-- View all snapshots
```

```
SELECT * FROM orderbook_snapshots ORDER BY snapshot_timestamp DESC LIMIT 10;
```

```

-- Count snapshots per market
SELECT ticker, COUNT(*) as snapshot_count
FROM orderbook_snapshots
GROUP BY ticker
ORDER BY snapshot_count DESC;

-- Latest prices for each market
SELECT ticker, best_yes_bid, best_yes_ask, best_no_bid, best_no_ask, snapshot_timestamp
FROM orderbook_snapshots
WHERE snapshot_timestamp = (SELECT MAX(snapshot_timestamp) FROM orderbook_snapshots);

-- Calculate average spreads
SELECT ticker,
 AVG(yes_spread) as avg_yes_spread,
 AVG(no_spread) as avg_no_spread
FROM orderbook_snapshots
WHERE yes_spread IS NOT NULL AND no_spread IS NOT NULL
GROUP BY ticker;
'''

Using Python:
'''python
import duckdb

conn = duckdb.connect("market_data.duckdb")
result = conn.execute("SELECT COUNT(*) FROM orderbook_snapshots").fetchone()
print(f"Total snapshots: {result[0]}")
conn.close()
'''

Troubleshooting

Common Issues

1. ModuleNotFoundError

```

**\*\*Error\*\***: `ModuleNotFoundError: No module named 'pydantic'`

**\*\*Solution\*\***:

```
```bash
```

```
source venv/bin/activate
```

```
pip install -r requirements.txt
```

```
```
```

#### #### 2. Authentication Error

**\*\*Error\*\***: `✖ AUTHENTICATION ERROR`

**\*\*Check\*\***:

- `KEY\_ID` in `config/settings.py` matches your Kalshi API Key ID
- Private key file exists at path specified in `KEY\_FILE\_PATH`
- Private key file is in PEM format
- You're using the correct key file (not expired/revoked)

#### #### 3. No Markets Found

**\*\*Error\*\***: `⚠ No markets found meeting criteria`

**\*\*Possible Causes\*\***:

- Volume threshold too high: Lower `MIN\_DAILY\_VOLUME` in config
- Markets are closed: System only fetches "open" markets
- API rate limiting: Wait and retry

**\*\*Solution\*\***: Check market status and volume:

```
```bash
```

```
python connect_and_price.py
```

```
```
```

#### #### 4. Database Permission Error

**\*\*Error\*\***: `PermissionError: [Errno 13] Permission denied`

**\*\*Solution\*\*:**

- Check file permissions on database file
- Ensure write access to directory
- Close any other processes using the database

#### #### 5. Python Interpreter Issue

**\*\*Error\*\*:** Using system Python instead of venv Python

**\*\*Solution\*\*:**

```
```bash
```

```
# Always activate venv first
```

```
source venv/bin/activate
```

```
# Or use venv Python directly
```

```
./venv/bin/python3 data_ingest.py
```

```
```
```

#### #### 6. Import Errors in Scripts

**\*\*Error\*\*:** Scripts can't find modules

**\*\*Solution\*\*:**

- Ensure you're in project root directory
- Verify virtual environment is activated
- Check that all `__init__.py` files exist in package directories

#### ### Debug Mode

Add verbose logging to troubleshoot:

```
```python
```

```
# In data_ingest.py, add at top:
```

```
import logging
```

```
logging.basicConfig(level=logging.DEBUG)
```

```
```
```

### ### API Rate Limits

Kalshi API has rate limits. If you see `'429 Too Many Requests'`:

- Reduce `'INGESTION_INTERVAL_SECONDS'` (increase wait time)
- Check Kalshi documentation for your tier's limits

---

### ## Additional Resources

#### ### Kalshi API Documentation

- Official Docs: <https://docs.kalshi.com>
- API Reference: <https://docs.kalshi.com/reference>
- Authentication Guide: [https://docs.kalshi.com/getting\\_started/quick\\_start\\_authenticated\\_requests](https://docs.kalshi.com/getting_started/quick_start_authenticated_requests)

#### ### DuckDB Documentation

- Official Docs: <https://duckdb.org/docs/>
- Python API: <https://duckdb.org/docs/api/python/overview>

#### ### Project Structure Best Practices

- Modular design with clear separation of concerns
- Configuration centralized in `'config/settings.py'`
- Data models in `'models/'` for validation
- Business logic in `'ingestion/'` modules
- Utilities in `'utils/'` for reusable functions

---

### ## Next Steps / Future Enhancements

#### ### Potential Improvements

##### 1. **Real-time WebSocket Integration**

- Replace polling with WebSocket subscriptions
- Get instant updates instead of 60-second polling

##### 2. **Advanced Analysis**

- Calculate term structure of Fed expectations
- Track spread evolution over time
- Detect arbitrage opportunities

### 3. **Trading Integration**

- Connect orderbook data to trading signals
- Implement automated trading strategies
- Portfolio management

### 4. **Data Visualization**

- Real-time dashboards
- Historical spread charts
- Market depth visualization

### 5. **Backtesting Framework**

- Historical strategy testing
- Performance metrics
- Risk analysis

### 6. **Multi-Series Support**

- Track multiple market series beyond KXFEDDECISION
- Cross-series analysis

### 7. **Database Optimization**

- Partitioning by date
- Indexing for faster queries
- Compression for historical data

---

## ## Summary

This system provides a complete foundation for quantitative analysis of Kalshi prediction markets:

- ✓ **Automated Data Collection**: Continuously gathers orderbook snapshots
- ✓ **Intelligent Market Selection**: Focuses on next 4 Fed meetings by date
- ✓ **Robust Architecture**: Modular, maintainable, fault-tolerant

- ✓ **Low Memory Usage**: Efficient database connection management
- ✓ **Production Ready**: Error handling, graceful shutdown, logging

The codebase is designed to be:

- **Understandable**: Clear structure and documentation
- **Maintainable**: Modular components, separation of concerns
- **Extensible**: Easy to add new features
- **Reliable**: Error handling and data persistence

You now have everything needed to build trading strategies, analyze market dynamics, and develop quantitative models using real Kalshi market data.

---

**Document Version**: 1.0

**Last Updated**: 2025-01-XX

**Project**: Kalshi Quant Trading System