1. Project Overview

This project is a highly modular, scalable, and production-grade **market-making system** for **Kalshi**, a real-money event trading platform.
The goal was to **autonomously trade** Kalshi prediction markets by dynamically quoting **bid/ask prices** using **statistical models**, **risk management**, **WebSocket streaming**, **backtesting frameworks**, and **real-time order execution.**

The system is split into **production bots** and **simulation/backtesting tools**, covering the entire research-to-deployment lifecycle.

---

2. Main Components

2.1 Production Trading Bots

The production side is responsible for **live trading** on Kalshi.
The code is split into **several versions** reflecting an **evolutionary development process**:

- `ind_research_1.py`: early basic bot
- `ind_research_2.py`: dynamic WebSocket subscriptions
- `ind_research_3.py`: top-market selection optimization
- `ind_research_4.py`: full asynchronous trading
- `ind_research.py`: final modular production bot

Each of these versions improved aspects like data handling, order management, market selection, risk control, and trading efficiency.

2.2 Simulation and Optimization

To improve the bots, the team also developed:

- `simulate.py`: a **realistic event-driven backtesting engine** to simulate Kalshi orderbooks and trade flows.

- `find_best_results.py`: analysis tools to **grid search** optimal model parameters (`gamma`, `kappa`) across many markets.
- `test_simulation.py`: unit tests ensuring the backtesting logic was accurate and reliable.

---

3. Technical Stack

- **Programming Language:** Python 3.11+
- **WebSocket Libraries:** `websockets`
- **HTTP Libraries:** `requests`
- **Cryptography:** `cryptography` (for Kalshi's RSA authentication)
- **Environment Management:** `python-dotenv`
- **Logging:** Python's built-in `logging`
- **Data Processing:** `pandas`, `matplotlib`
- **Backtesting/Simulation:** Pure Python (no external finance libraries)

---

4. Core Modules Description

4.1 Kalshi API Clients

- **HTTP Client (`KalshiHttpClient` from `clients.py`)**
  Authenticates and manages all REST calls (`get_markets`, `get_trades`, `create_order`, etc.).
- **WebSocket Client (`KalshiWSClient`)**
  Connects to Kalshi's real-time orderbook/ticker updates.
  Supports **dynamic subscriptions** — only subscribing to markets deemed interesting

for trading.

- **ExchangeClient (`KalshiClient.py`)**
  Combines both REST and WebSocket APIs into one higher-level client.

These clients included **rate limiting**, **RSA signature authentication**, and **error handling** to ensure robustness.

4.2 Market Data Handler

- Fetches market data from either **WebSocket cache** or **REST API**.
- Computes:
  - Mid Price
  - Bid-Ask Spread
  - Volatility Estimations (from recent trade prices)
  - 24-hour Volume
- Handles missing data gracefully by retrying or falling back to REST.

This is **critical** because price quoting depends on live orderbook conditions.

4.3 Risk Management

- Enforces **position limits** dynamically.
- Adapts allowed position sizes **based on market volatility**.
- **Caches volatility** estimates to avoid expensive recomputation.
- Provides **order size adjustments** to avoid breaching account limits.

A **RiskManagerConfig** allows quick tuning through environment variables.

4.4 Strategy Engine

- Implements a version of the **Avellaneda–Stoikov Model**, a renowned stochastic model for **optimal market-making**.
- Computes:
    - Reservation Price (inventory-adjusted fair value)
    - Optimal Spread (balancing profit vs inventory risk)
    - Inventory Skew (to rebalance positions toward zero)
- Parameters `gamma` (risk aversion) and `kappa` (order arrival rate) are tunable.

This ensures the bot quotes tighter spreads in liquid markets and wider spreads in volatile ones.

4.5 Order Manager

- Places, cancels, and tracks live limit orders.
- Automatically handles:
    - Bid vs Ask side.
    - Post-Only flags (to avoid taking liquidity unless intentional).
    - Cancels stale or filled orders proactively.

It also **maps ticker → active orders** to ensure that outdated quotes are efficiently replaced.

4.6 Market Maker

- **Single-market manager** that runs a trading cycle:
    1. Check if it's safe to trade (risk check)

2. Fetch data
3. Calculate optimal quotes
4. Place bid and ask orders
5. Cancel old orders after a short delay

Each market operates semi-independently, allowing **parallel trading across multiple Kalshi markets**.

4.7 Market Opportunity Bot

- Continuously scans all active markets.
- Selects **top markets** based on a custom scoring formula:
    - Prioritize **high volume**, **wide spreads**, **low volatility**.
- Dynamically **starts MarketMakers** on the best markets.
- Refreshes selections every few minutes.

In later versions (`ind_research_3.py` and `ind_research_4.py`), the bot **ranked markets** using:

score = (effective volume) / (estimated volatility + ε) × exp(-((spread - target spread)²) / (2σ²))

ensuring a probabilistic, machine-learning-inspired market selection.

---

5. Simulation & Backtesting Framework

Because deploying a trading strategy live without simulation is **risky**, a fully event-driven simulation engine was also developed.

It allows:

- Testing various strategies offline.
- Fine-tuning model parameters like risk aversion (`gamma`) and orderbook elasticity (`kappa`).
- Identifying strategies that maximize profitability while minimizing inventory risk.

The simulation is handled across several files:

- `simulate.py`
- `find_best_results.py`
- `test_simulation.py`

5.1 Simulation Engine Overview (`simulate.py`)

Key Components:

- **Event-Driven Simulation:**
    - Both **orderbook updates** and **trades** are replayed in timestamp order.
    - Trades and orderbook changes trigger re-evaluation of orders.
- **Custom Event Model:**
    - `SimulationEvent` class wraps events (either "orderbook" or "trade") and orders them by time.
    - If two events have the same timestamp, **orderbook events** are prioritized (logical for market making).
- **Strategy Engine:**
    - Same **Avellaneda–Stoikov-based model** as production bots.

    - Determines dynamic **bid/ask prices** and **order sizes** based on current inventory and volatility.

- **Inventory and Cash Management:**
    - Tracks filled trades.
    - Updates inventory.
    - Simulates cash impact precisely.
- **Volatility Estimation:**
    - Maintains a **rolling window** of last 60 seconds of trade prices.
    - Calculates standard deviation (`statistics.stdev`) as volatility.

5.2 Optimization Grid Search

- Performed **parameter sweeps** over combinations of:
    - `gamma` values (risk aversion) from **0.05 to 1.0**.
    - `kappa` values (order flow elasticity) from **0.5 to 5.0**.
- For each (gamma, kappa) pair:
    - Simulate all historical data.
    - Record resulting `cash` and `inventory`.

Stored outputs into:

- `market_results_2.json`
- `overall_results_2.json`

5.3 Finding Best Parameters (`find_best_results.py`)

- **Best Overall Strategy:**
  Selects the `(gamma, kappa)` pair with the **highest average cash** across all markets.

- **Best Per-Market Strategy:**
  For each market individually, picks the best simulation based on:

    - Highest cash.

    - If cash tied, smallest absolute inventory (closer to neutral).

Outputs:

- `best_overall_2.json`

- `best_by_market_2.json`

5.4 Testing Simulation (`test_simulation.py`)

- **Unit Tests** verifying:

    - Correct parsing of orderbooks and trades.

    - Correct calculation of volatility.

    - Correct updating of inventory and cash on fills.

    - Correct reservation price and spread computation.

Examples:

- Simulate a market with known trades.

- Confirm that placing a bid at a computed price fills appropriately.

- Ensure a gamma change affects the quoting logic as expected.

This testing suite is **critical** for ensuring confidence in backtest reliability before live trading.

---

6. Additional Tools & Data Collection

The project also included helper scripts to **gather and process real Kalshi data** for simulations:

6.1 create_orderbook_history.py

- **Archives Real-Time Data:**
  Repeatedly polls Kalshi APIs every few seconds.
- For each market ticker:
  - Saves **orderbook snapshots**.
  - Saves **trade data**.
- Saves as `.jsonl` files for efficient event-driven replay.

This allowed the simulations to **replay realistic market conditions**, not theoretical assumptions.

6.2 main.py

- **Quick Data Collection and Analysis:**
  - Pulls live market data.
  - Measures:
    - Current spreads
    - 1-hour trade volume averages

        - 1-hour highs/lows

- Can generate datasets for quick filtering and identifying **high-potential markets** for new trading experiments.

---

7. Improvements Over Time

Each "generation" of the bots (`ind_research_1.py` → `ind_research_4.py`) made major architectural improvements:

Version	Key Improvements
`ind_research_1.py`	Basic REST-only bot, static risk limits.
`ind_research_2.py`	Added WebSocket live orderbook feeds; dynamic risk sizing based on volatility.
`ind_research_3.py`	Dynamic top-market selection based on custom scoring formula (volume/spread/volatility tradeoff).
`ind_research_4.py`	Full asynchronous trading: updating market list and trading concurrently without delays.
`ind_research.py`	Final consolidated version for production deployment; modular and clean.

This evolution led to:

- Faster reaction to market changes.

- Smarter market selection.
- Safer trading via adaptive risk management.
- Better spread capture efficiency.

---

8. Architectural Highlights

✅ **WebSocket-first architecture**: live and low-latency.
✅ **Risk-aware quoting**: no uncontrolled runaway inventories.
✅ **Volatility-adaptive sizing**: larger quotes in calm markets, smaller in volatile ones.
✅ **Asynchronous multi-market trading**: scaling to dozens of markets simultaneously.
✅ **Offline simulation for optimization**: better strategies without risking real funds.
✅ **Robust error handling and rate-limiting**: production stability.