ChatGPT

Technical Roadmap for an AI Trading Bot Platform

This roadmap outlines how an AI-driven crypto trading service (like the AIROBOT Team platform) could be built, focusing on the trading bot and core mechanisms. It covers user onboarding, deposit and Binance API integration, market analysis, buy/sell logic, tiered profit handling, and security safeguards. We also discuss architecture and tooling (Python backend vs. React frontend).

1. Account Registration & Authentication

- **User Sign-up Flow:** Provide a secure registration page (React frontend) where users register with email/password and complete email verification. On success, issue a short-lived JWT or session token. Follow OWASP recommendations (e.g. strong password hashing with bcrypt/scrypt, rate-limit login attempts, enforce HTTPS) [1] .
- **Multi-Factor Authentication:** Encourage or require 2FA (TOTP via Google Authenticator) for extra security, since the platform deals with funds.
- **Authentication Service:** Implement in Python (e.g. using FastAPI or Django) a service that handles login, password reset, and token issuance. Use an established library (e.g. Flask-JWT-Extended, Django Rest Framework) and set token expiration to minutes/hours (not days) [1] . Store refresh tokens if needed and revoke on logout.
- **API Key Management:** Once registered, allow users to save sensitive data (Binance API key/secret) in their account. Store such secrets encrypted (e.g. use a vault or encrypted database fields) and load them as needed. As a best practice, retrieve API keys via environment variables or secure vaults in production [2] . Use role-based access control so only the user (and the trading bot process) can access the keys.

2. Fund Deposit Mechanism & Binance API Binding

- **Deposit Workflow:** Users need funds (e.g. USDT) in the system to trade. Options include:
- **On-chain Deposit:** Generate unique crypto deposit addresses for each user (or use a shared address with memo/tag). Monitor the blockchain (via Web3.py, a node API, or a service like Infura) for incoming transactions. Once a deposit is confirmed (e.g. 2 network confirmations), credit the user's internal balance. (This is analogous to how exchanges use APIs to retrieve deposit addresses [3] .)
- **Direct Transfer via Exchange:** If using user-supplied Binance API keys, the user simply transfers their own Binance balance (e.g. USDT) in their Binance account; no on-chain deposit to the platform is needed. (The UI's "Recharge" button likely just reminds user to fund their Binance.)
- **Binance API Key Binding:** Ask the user to create a Binance API Key/Secret from their Binance account with **only "Enable Spot/Margin Trading"** permission and **disable Withdrawals**. Store these in the backend. Bind these keys to the user's account so the trading bot can act on their behalf. The python-backend (e.g. using the python-binance library) will initialize a `Client` with these keys for each user [4] .
- **API Key Security:** As python-binance docs suggest, keep API keys secret (e.g. environment variables or encrypted storage) [2] . Do **not** give withdrawal permission on these keys. Binance requires an IP

whitelist to enable withdrawals via API [5], so by default keep that off. Only trade permission is needed, ensuring funds can't be siphoned out via API.

- **Internal Ledger vs. Direct Trading:** Decide if the platform will custody funds or trade directly in users' Binance accounts. A safer design (and common for such bots) is to execute trades on the user's own Binance account via API. In this case, "deposits" just mean the user has funds in Binance, and profits remain in their Binance balances. The platform does not actually hold user funds, which avoids withdrawal complexity. (If a custodial model is used, implement a ledger in a database to track user balances and update it on trades or deposits.)

3. Market Analysis & Pair Selection

- **Market Data Sources:** Fetch real-time and historical market data for relevant pairs via Binance's API/WebSocket (or ccxt library). Use Binance's WebSocket streams for live price ticks and REST endpoints for history. This allows 24/7 data polling and historical backtesting. According to Binance API docs, Binance offers both REST and WebSocket endpoints for market data [6].
- **Pair Filtering:** Focus on high-liquidity, high-volatility pairs like BTC/USDT or ETH/USDT. These pairs are popular due to liquidity and volatility, which trading strategies often exploit [7]. The platform could allow configuration of multiple pairs. Use criteria such as trading volume, volatility, and correlation to select assets. For example, auto-choose top market-cap coins against USDT.
- **Technical Indicators:** Precompute indicators (RSI, Moving Averages, MACD, Bollinger Bands, etc.) using TA-Lib or pandas. These indicators turn price data into signals [8]. For instance, "buy low" might use oversold RSI (<30), while "sell high" might use overbought RSI (>70) or price above a moving average. TA-Lib is widely used and offers 150+ indicators [9]. Tools like `pandas_ta` or `bta-lib` can also compute indicators in Python.
- **AI/ML Models (Optional):** If "AI" is a goal, one could train a machine learning model (using TensorFlow/PyTorch) on historical data to predict short-term price moves. E.g. a neural network or tree-based model that inputs recent candlestick patterns and outputs a buy/sell signal. However, caution is needed as markets are noisy. At minimum, use quantitative analysis like walk-forward backtesting to validate strategies [10].
- **Sentiment/Data Feeds:** (Advanced) Optionally integrate external data (news sentiment, social media) for alpha. For example, use a news API or crypto sentiment data to adjust trade signals. This is complex, so it's a later consideration once basic technical analysis is in place.

4. Trading Logic & Strategy

- **Buy/Sell Conditions:** Implement the core trading logic as rule-based or AI-driven signals. This could follow known strategies:
- **Buy Low, Sell High (Long Strategy):** Example: *buy* when the price dips below a moving average or RSI is low and then *sell* when the price rallies above a threshold. For instance, if a 10-day RSI < 30 (oversold) and price closes above a 10-period moving average in the next hour, trigger a buy order. Later *sell* when RSI > 70 or price hits a take-profit level. This aligns with classic mean-reversion or breakout tactics. Use indicators from TA-Lib to code these rules [8].
- **Sell High, Buy Low (Short Strategy):** In addition to longs, allow shorting (if using margin or futures). For example, on Binance Futures, the bot can *sell* (short) when a coin looks overbought and *buy back* (cover) when it drops. Crypto shorting means borrowing and selling high, then repurchasing cheaper [11]. Implement via Binance Futures API (python-binance supports futures trading) or by using a stablecoin position (sell coin for USDT when high, buy back when low).

- **Stop-Loss / Risk Management:** For every trade, set stop-loss orders to cap downside (e.g. 1–3% below entry) and take-profit or trailing stops. This mitigates losses in volatile moves. Also enforce position sizing rules (never use >X% of account in one trade).
- **Order Execution:** Use limit orders or market orders via Binance API (`client.order_limit_buy/sell` or `client.order_market_buy/sell`) based on the strategy. For speed, market orders execute immediately, but limit orders can reduce slippage.
- **Signal Processing:** Combine multiple factors: e.g., a scoring system that weighs RSI, MACD crossovers, volume spikes, etc., to filter out noise [8]. This can be a simple sum-of-signals or a small ML classifier that outputs "buy/sell/hold."
- **Backtesting:** Before going live, backtest the bot's rules on historical data. Simulate the buy/sell logic on past price series to measure drawdown and profitability. Only after satisfactory backtesting/paper-trading move to live.

5. Tiered Profit Logic (Package System)

- **Investment Tiers:** Implement a tiered subscription or "investment pack" system where users get different profit-sharing or fee rates based on deposit size. For example, a "VIP1" plan might require a minimum $25 deposit and entitle the user to a certain profit share, while "VIP2" at $100 grants a higher share. These mimic MLM-style tiers seen in similar platforms [12].
- **Profit Sharing Formula:** Define how profits are split. For instance, if a user's trades generate $100 profit, the platform might take a commission on that profit differently per tier (e.g. Tier1 pays 10% of profit to platform, Tier2 pays 22.5%, Tier3 pays 24.5%, etc., matching the logic like "$25=10% fee, $100=22.5% fee" as given). Store these percentages in a config or database. After each closed trade, subtract the platform's cut accordingly and credit the remainder to the user.
- **Payment Out of Profits:** To implement "keep profit within Binance," the bot might immediately convert a portion of profit to a stablecoin balance on the user's Binance account, effectively forcing users to leave profits on Binance. Alternatively, maintain a counter so users only see credited profits in their dashboard and must use Binance's own withdrawal to get cash out.
- **Upgrades & Rebates:** If needed, allow users to upgrade tiers by depositing more and adjust future profit splits. Keep clear logic in backend to recompute their tier on each deposit or at defined checkpoints. (Note: Tiered MLM features are secondary to the bot's logic and can be handled as business logic within the Python backend.)

6. Safeguards & Security

- **API Permissions:** As noted, **do not enable withdrawal permissions** on user API keys [5]. This ensures the bot can only trade, and cannot pull funds out of the user's exchange account. Binance requires special whitelisting for withdrawals via API, so simply don't configure that.
- **Transactional Safety:** Every trade/order should be confirmed. Use Binance's order response to verify fills and handle any API errors (retry or rollback logic). Keep detailed logs of all orders and account changes. In case of anomalies (e.g. repeated rejections), pause the bot and alert the user/admin.
- **Profit Custody:** By trading directly in the user's Binance account, profits never pass through the platform's own wallets. This keeps money "in Binance" as a safeguard – the platform doesn't custody the funds, it only triggers trades. Users must withdraw via Binance when they wish (outside the platform). This avoids needing a withdraw button on the platform; the "Withdraw" in the UI could be disabled or simply direct users to Binance.

- **Data Security:** Use HTTPS for all client-server and API-server communication. Sanitize all inputs and protect against common web vulnerabilities (SQL injection, XSS). Regularly update dependencies. Use environment variables or a secrets manager for API keys and database credentials.
- **Monitoring and Alerts:** Monitor running bots and server health. Implement checks: if a bot is disconnected or fails to place trades, notify admins. Track metrics like uptime, trade success, and drawdown. Maintain an admin dashboard or logging (e.g. with ELK or a hosted monitoring tool) to review system behavior.
- **Legal/Compliance (Advisory):** Although not a technical measure, display disclaimers. (For example, regulators warn that "AI trading bots" guaranteeing high returns are often scams [13].) Encourage users to do KYC/AML if raising large amounts (depending on jurisdiction). This isn't a software feature, but good practice for a real platform.

7. System Architecture & Tech Stack

- **Overall Design:** Use a **frontend–backend** architecture. The React frontend handles the user interface (registration, dashboard, charts), while the Python backend powers the trading logic and APIs. Communication occurs over REST (or GraphQL) and WebSockets for real-time updates.
- **Frontend (React):**
- *Responsibilities:* User registration/login forms, Binance API key input, display balances and trade history, charts of asset prices, profit metrics, and deposit status. Possibly pages for "Quantitative Account" overview and tier info.
- *Tools:* React with Hooks or Redux for state. UI libraries like Material-UI or Ant Design for layout. Charting library (e.g. Chart.js, Recharts, or TradingView Charting Library) to plot live price data. Axios or Fetch for API calls to Python backend. React Router for navigation.
- *Security:* Implement route guards (redirect to login if unauthenticated). Store JWT in HTTP-only cookies or secure storage. Use HTTPS.
- **Backend (Python):**
- *Framework:* FastAPI or Flask for the REST API. FastAPI is recommended for async support (handling WebSockets and many concurrent users). Use Flask or Django if you prefer.
- *Components:*
    - **Auth Service:** Handles user accounts, JWT issuance, password reset, 2FA verification.
    - **Trading Service:** The core bot engine. Could be a separate process or service that fetches market data and makes trade decisions. This service uses the Binance API (via `python-binance` or `ccxt`) to place orders with each user's keys. It may run scheduled tasks (cron/Celery beat) or listen to WebSocket events.
    - **Data Storage:** A database (e.g. PostgreSQL or MongoDB) to store user profiles, API keys (encrypted), tier info, trade logs, historical signals, etc. Use an ORM (SQLAlchemy or Django ORM) for ease.
    - **Task Queue:** Use Celery (with Redis/RabbitMQ) for background jobs – e.g. polling prices, executing trades, sending email alerts. This decouples long-running or periodic tasks from the main web server.
    - **WebSocket/Stream Processor:** An async component (could be in FastAPI or separate) to connect to Binance's WebSocket stream for live price ticks. It processes data in real-time for the bot.
    - **Business Logic:** A module to compute technical indicators (using TA-Lib/Pandas), decide trades, apply tier commission rules, and update balances.

- *Libraries:*
  - **Binance API:** `python-binance` or `ccxt` for unified exchange API (ccxt can also connect to Binance). Use these to get balances, prices, and create orders [4].
  - **Data Analysis:** `pandas`, `numpy` for data; `TA-Lib` for indicators [9]; `scikit-learn` or `TensorFlow` if using ML.
  - **Authentication:** `PyJWT` or built-in FastAPI security for JWTs. `bcrypt` or `passlib` for password hashing.
  - **Infrastructure:** Docker containers (one for React app, one for FastAPI, one for worker queue). Deployment on AWS/GCP/Azure using Docker Compose or Kubernetes for scalability.
- **Component Responsibilities (Summary):**
- **Web Server (Python API):** Exposes endpoints: login, register, bind API key, get account info, show trade history. Also serves WebSocket updates to frontend (or the frontend can poll).
- **Trading Bot Worker:** Runs the market analysis and trading logic continuously. Loads each user's Binance keys from the database, fetches market data, executes buy/sell orders, and records results.
- **Database:** Stores all user data, tiers, and records. Ensures ACID compliance for transactions (e.g. updating balances).
- **Frontend (React):** Fetches user data from API, displays live charts (can stream updates via WebSocket), and allows user actions (deposit instructions, tier upgrade).

Example Technology Stack: Python 3.9+, FastAPI, PostgreSQL, Celery+Redis, Binance python API, Pandas/TA-Lib; React (CRA or Next.js), Redux, Chart.js, Axios.

Recommended Tools and Libraries

- **Exchange APIs:** Binance Spot/Futures API (via [python-binance] or [ccxt]) [6] [4].
- **Technical Analysis:** TA-Lib (150+ indicators) [9], pandas_ta or bta-lib for indicators; NumPy/Pandas for data manipulation.
- **Web Framework:** FastAPI (async) or Django/Flask for backend API.
- **Task Queue:** Celery with Redis/RabbitMQ for scheduled and background tasks (e.g. polling market data, executing periodic trades).
- **Database:** PostgreSQL or MySQL (via SQLAlchemy or Django ORM) to store users, keys, trades, logs. For scalability, consider TimescaleDB (Postgres extension) for time series of price data.
- **Frontend:** React + TypeScript (optional) with Axios for REST calls; UI libs (Material-UI); Recharts or TradingView widget for charts.
- **Authentication:** PyJWT for token management; use an OAuth2 library or implement email/2FA flows.
- **Security:** Let's Encrypt for SSL; OWASP-recommended headers; rotating logs with Logrotate or ELK stack.
- **Monitoring:** Prometheus/Grafana for metrics, Sentry for error tracking.

By following these components and best practices, one can build a robust Python-based backend trading bot and a React dashboard for an "AI trading bot" service. The system design ensures account security, automated market analysis, and clear trade execution logic, while keeping sensitive operations (API keys, withdrawals) tightly controlled [5] [1].

Sources: This roadmap is based on current practices and documentation for crypto trading bots and APIs [6] [4] [8] [7] [12], and draws on published guides for building algorithmic trading systems [14] [9].

1 JWT Security Best Practices | Curity
https://curity.io/resources/learn/jwt-best-practices/

2 4 5 6 Binance Python API – A Step-by-Step Guide - AlgoTrading101 Blog
https://algotrading101.com/learn/binance-python-api-guide/

3 Get Deposit Addresses | Kraken API Center
https://docs.kraken.com/api/docs/rest-api/get-deposit-addresses/

7 Trading Pairs in the Cryptocurrency Market: A Complete Guide for Beginner Traders | Robinhood101 on Binance Square
https://www.binance.com/en/square/post/860659

8 10 14 Step-by-Step Guide to Building a Custom AI Trading Bot | Python-bloggers
https://python-bloggers.com/2025/01/step-by-step-guide-to-building-a-custom-ai-trading-bot/

9 TA-Lib/ta-lib-python: Python wrapper for TA-Lib (http://ta-lib ... - GitHub
https://github.com/TA-Lib/ta-lib-python

11 How to Short Crypto | IG International
https://www.ig.com/en/trading-strategies/how-to-short-cryptocurrencies-230612

12 MI Airobot Review: Quantitative trading "click a button" Ponzi
https://behindmlm.com/companies/click-a-button-app-ponzis/mi-airobot-review-quantitative-trading-click-a-button-ponzi/

13 Customer Advisory: AI Won’t Turn Trading Bots into Money Machines | CFTC
https://www.cftc.gov/LearnAndProtect/AdvisoriesAndArticles/AITradingBots.html