ChatGPT

CryptoPulse AI: Architecture and Design of an LLM-Powered Crypto Trading Bot

Overview

We propose a modern **AI-driven cryptocurrency trading platform** that couples a React/TypeScript frontend with a Python FastAPI backend running LLM-based agents (via LangChain) to **analyze markets and execute trades**. The frontend dashboard (CryptoPulse AI) presents real-time charts and an AI chat assistant; the backend handles data aggregation, LLM reasoning (Google Gemini), and secure order execution. Key attributes:
- **Hybrid Python–TypeScript Stack:** React + Tailwind CSS on the client; Python (FastAPI) for AI and trading logic [1]. This separation lets us leverage Python's AI ecosystem while providing a responsive web UI.
- **LLM Integration:** We use Google's Gemini API (via Google AI Studio) for large-context market analysis, with fallbacks to open models (Ollama/OpenRouter) for resiliency. LangChain's AgentExecutor wraps market-data tools so the LLM can execute functions like `get_price()` or `place_order()` as needed [2] [3].
- **Secure Key Management:** All API keys (exchange keys, Gemini keys, etc.) are stored **only on the server side**, encrypted at rest or in a vault [4] [5]. The client never sees raw secrets. We inject secrets via environment variables or protected CI/CD secrets.
- **Cloud Deployment & CI/CD:** Services are containerized (Docker) and deployed on Fly.io for global low-latency delivery. GitHub Actions pipelines handle linting, testing (unit/integration), security scans (Snyk), and automated deployment on commits.
- **Observability & Security:** We integrate Sentry (React and Python SDKs) for error/performance monitoring [6] [7]. Snyk scans dependencies, and we follow best practices (rate limits, audit logs, AWS Secrets Manager or similar). Threat modeling (e.g. STRIDE) guides our risk mitigation.

High-Level Architecture

The system has two tiers: a **Client** (React/TS dashboard) and a **Server** (Python API + agents). Key components include:
- **AI Chat Agent (Backend):** A Python service uses LangChain's AgentExecutor. We define "tools" for market tasks, e.g.:

```
def get_price(symbol: str) -> str:
    price = exchange.fetch_ticker(symbol)['last']
    return f"The price of {symbol} is {price} USD"
def place_order(symbol: str, side: str, amount: float) -> str:
    order = exchange.create_order(symbol, 'market', side.lower(), amount)
    return f"Executed {side} order for {amount} {symbol}: ID {order['id']}"
tools = [Tool("GetPrice", get_price, ...), Tool("PlaceOrder",
place_order, ...)]
agent = initialize_agent(tools, llm, agent="conversational-react-
description", verbose=True)
```

The LLM (Gemini or fallback) uses these tools under controlled prompts, so a user's natural language ("Buy 0.01 BTC now") triggers the `place_order` function. [2] [3]
- **Market Data Services:** Python modules fetch cryptocurrency prices (e.g. CoinGecko API) and news (CryptoCompare or custom scrapers). We implement circuit breakers and caching to ensure robustness.
- **Exchange Interface:** A secure wrapper (e.g. CCXT) for Binance's API handles actual trades. User-supplied Binance API keys (spot-only, no withdrawals) are read from the backend and used server-side. We honor rate limits and use Binance's testnet for development.
- **Data & Auth (Supabase):** We use Supabase (Postgres) for user and session data. Supabase Auth (GoTrue) issues JWTs and enforces row-level security so each user's data is isolated [8] [9]. Real-time features (e.g. collaborative charts) could use Supabase's realtime engine if needed.
- **Infrastructure & CI/CD:** Components run in Docker. GitHub Actions automates: unit tests, type checks, Snyk scans, Docker image builds, and deployments to Fly.io. On deployment, Sentry is connected (via DSNs) and metrics/logs (container logs, Prometheus/StatsD) are enabled.

For reference, one published design used **AWS Lambda** with scheduled triggers (EventBridge) and DynamoDB as state store for a crypto bot [10]. It polled market data, updated context, executed trades, and pushed metrics to Grafana. We follow a similar pattern (periodic tasks or event triggers), but using our chosen stack.

The trading workflow is often modeled as a **loop of steps**: create a *watchlist*, note key *catalysts* (news/events), mark support/resistance *levels*, plan trades, track *order flow*, and conduct a *post-mortem* review [11]. Cointelegraph explicitly calls this a "repeatable loop" of trading steps [11]. LLMs like Gemini can assist at each stage by summarizing data (price feeds, news sentiment, charts) into actionable insights [11]. For instance, Gemini can rank your watchlist by 24h volatility or filter news headlines for impact on ETH/SOL, then help draft a disciplined trading plan [11] [12].

Frontend (TypeScript/React)

The user interface is a single-page React app (TypeScript) with Tailwind CSS. Features include:
- **Live Charts & Data:** Real-time crypto price charts (e.g. recharts) for multiple timeframes. Data is fetched from CoinGecko or streaming exchange feeds. We show price changes, order book depth, etc.
- **AI Chat Panel:** A conversational UI where users ask market questions or request actions. Chat history is stored (e.g. in IndexedDB or backend DB) and can be managed (create/select sessions). Queries are sent to a FastAPI chat endpoint that invokes the LangChain agent.
- **Alerts & Wallets:** UI to set price alerts (e.g. notify when BTC>60k). Simulated wallet modal (MetaMask, Binance Connect) shows how orders connect to user accounts. Actual wallet integration (for futures or crypto margin) could be added, but initially it's a visual placeholder.
- **Responsive Design & Themes:** Layout adapts to desktop/mobile. We include a dark/light mode toggle and custom toasts for feedback.
- **Error Monitoring:** We include Sentry's React SDK to capture JS errors, exceptions, and performance metrics [6]. This yields stack traces and user context (browser, screen) to debug front-end issues.
- **No Secrets on Client:** The frontend has **no secret logic**. All API calls requiring keys go to our backend. Environment variables prefixed (e.g. `VITE_`) expose only public keys (e.g. analytics).

Backend (Python, FastAPI + LangChain)

The server uses Python for its rich ML and crypto libraries [1]:
- **LangChain Agent Executor:** We set up LangChain's `AgentExecutor` with our tool functions. This lets the LLM decide which tool to call at each step. For example, if the user says "What's a good entry for BTC today?", the agent might call `get_price("BTC/USDT")` and then respond with the result.

- **Google Gemini API:** We call Gemini's API for language understanding. Prompts include system/context instructions and the dynamic user query. We follow best practices: give GPT function descriptions, limit token usage, and include relevant data in the prompt. We also guard the LLM: it should **never receive raw secrets or trade without confirmation** [4] [13]. Keys and sensitive info stay on the backend.
- **Trade Execution:** When the agent decides to trade, it invokes our secure `place_order()` tool. That tool calls Binance's REST API. We log all trade events and results. Initially, we operate in paper-trading mode (Binance Testnet) until strategies are vetted.
- **REST API Endpoints:** FastAPI exposes endpoints (e.g. `/api/prices`, `/api/order`, `/api/chat`) for the frontend. These endpoints authenticate via Supabase JWT and validate inputs. They return JSON with data or success/failure.
- **Model Fine-Tuning & Retraining:** (Long-term) we may fine-tune a private LLM on logged trading interactions or research data to improve responses. All training data would be sanitized for privacy.

Data & API Integration

We aggregate from multiple sources:
- **Price Data:** CoinGecko and exchange APIs for live and historical prices. We fetch candlesticks (OHLCV) for charts and on-demand tickers for the agent. Data calls use retries, caching, and respect rate limits.
- **News & Social Feeds:** CryptoCompare News API (or RSS feeds, Twitter scraping) supply market news. We feed recent headlines to Gemini as context or let the agent query them (via a `get_news()` tool).
- **Exchange (Binance):** The Binance spot API powers our order execution and account queries. We restrict keys to trading only. We also monitor Binance's websockets for real-time fills/status updates (for portfolio sync).
- **Database (Supabase/Postgres):** All persistent data lives in a Postgres DB (via Supabase). This includes user profiles, saved chat transcripts, trade history, and configurations. Supabase automatically provides a RESTful API (PostgREST) and Realtime channels if needed [8] [9]. We leverage Supabase Auth for users and Row-Level Security (RLS) so each user sees only their own data.

Secure Key Management & Security Rules

We enforce strong security:
- **Backend-Only Secrets: Never send private keys to the client.** Binance and Gemini keys reside on the server and are fetched from secure storage (env vars, vault). The frontend requests an action, the backend executes it with the key. This follows best practices [4] [5].
- **Environment & Vaults:** Use `.env` files (not committed) or cloud secret managers. In Docker, pass secrets via Fly.io secrets. Implement CI/CD secret injection so no keys end up in git.
- **Least Privilege:** We create separate API keys per environment (dev, staging, prod). Binance keys have only trading permissions (withdrawals disabled). GenAI keys have minimal scopes.
- **Network Security:** All traffic is over HTTPS. We configure CORS to only allow our frontend origin. We use rate limiting and input validation on our FastAPI endpoints to prevent abuse.
- **Threat Modeling (STRIDE):** We applied STRIDE to identify attacks. For example, a stolen private key could allow **Spoofing** or **Privilege Escalation** by injecting fraudulent orders [14]. To mitigate, we store keys encrypted and require 2FA for admin actions. We log all API accesses for audit trails.
- **LLM Safety:** The assistant is guided to analysis only – it "reasons" and suggests trades but requires explicit user sign-off. We strictly sanitize user inputs in prompts and disable open-ended code execution. The function-calling interface itself prevents the LLM from inventing unauthorized actions [4] [13].
- **Monitoring & Scanning:** Sentry tracks runtime errors; we set up alerts for exceptions. We run Snyk or similar SAST tools in CI to catch vulnerable dependencies before deployment.

- **Compliance:** If deployed commercially, we'd add KYC/AML controls. PAAL's design for compliance (audit logs, fine-grained permissions) [15] [16] influences our approach.

Competitive Analysis

Our design draws on emerging industry trends:
- **Bybit TradeGPT:** Bybit's *TradeGPT* integrates OpenAI GPT-4 with proprietary data tools to answer trader questions [17]. It provides real-time market insights (patterns, sentiment, price forecasts) but *does not* autonomously execute trades [18]. It's an embedded exchange chatbot for decision support.
- **PAAL AI:** A blockchain-focused AI ecosystem. PAAL allows users to train custom bots (personalized on their data) and supports multi-modal inputs (text, image, audio, video) [15] [16]. Its emphasis is on user customization and compliance (KYC/AML logs). PAAL's assistants perform research and analysis (chart summaries, sentiment) as well as limited execution. Our system similarly uses multi-modal LLMs but currently focuses on core trading tasks.
- **ChainGPT:** Marketed as an "AI for blockchain," ChainGPT offers a Web3 AI assistant and tools like smart contract auditors. Its trading assistant answers price queries and emphasizes on-chain analytics (token metrics, NFT info) [19] [20]. It's powered by a native token and open APIs.
- **Other LLM Agents:** Solutions like Manus AI and AltFINS Copilot show multi-agent trading assistants are feasible [1]. These integrate data feeds and LLMs to automate tasks. Unlike exchange-specific bots, our architecture is platform-agnostic, designed for easy integration with any exchange (via CCXT) and extensible to new data sources.

Differentiators: Our stack uses open-source building blocks (Supabase, FastAPI, Fly.io) for transparency and cost-effectiveness. We emphasize **security and observability**, using threat modeling (STRIDE), Sentry, and Snyk to reassure stakeholders. Unlike a simple one-off script, this is a production-ready microservices architecture with CI/CD, monitoring, and strong compliance considerations – crucial for investor confidence and real-world deployment.

Implementation Guide & Roadmap

1. **Design & Prototyping:** Begin by coding tool functions (`get_price`, `place_order`, etc.) and a simple LangChain agent. Test each tool in isolation (e.g. fetch a real Binance ticker using the sandbox, execute a mock order). This verifies the agent's function-calling loop [2] [3].
2. **Backend Setup:** Scaffold a FastAPI project. Create endpoints for authentication (via Supabase), chat (`/api/chat`), market data (`/api/prices`), and orders. Wire in Supabase SDK for user management. Configure environment variables for secrets (Gemini key, Binance keys, Supabase URL/keys).
3. **Frontend Development:** Use Vite (or Create React App) to set up the React/TS project (see CryptoPulse README). Build core components: Chart (price), ChatPanel (AI assistant), NewsFeed, AlertModal, WalletModal, etc. Implement state stores or Context for chat sessions and portfolio. Connect components to backend APIs.
4. **Data Integration:** Implement a data service layer in Python. Use [CoinGecko API] for price history; add circuit-breaker logic to fall back to cached data if the API fails. Similarly, call CryptoCompare for news (with caching). Write unit tests for these data-fetching functions.
5. **Trading Logic:** Start with a simple strategy (e.g. RSI-based). The LLM can generate trade signals, but we wrap them with our own risk checks (max trade size, stop-loss). Initially require user confirmation before an API order. Use Binance's testnet to simulate.
6. **CI/CD Pipeline:** Configure GitHub Actions:
7. Lint and type-check (ESLint for TS, mypy/flake8 for Python).
8. Run unit tests (Jest for frontend, pytest for backend).

9. Run Snyk (or OWASP ZAP) scans for known vulnerabilities.
10. Build Docker images for `frontend` and `backend`. Tag with commit SHA.
11. Deploy to Fly.io on merge to `main`, and to a `staging` app on pull request.
12. **Monitoring & Logging:** Register Sentry for both apps. Initialize Sentry SDK in React and FastAPI to capture uncaught exceptions and performance traces [7] [6]. Set up Fly.io alerts or external logging (Datadog, Prometheus/Grafana) for infrastructure metrics.
13. **Testing:** Write integration tests. For trading, mock Binance responses or use testnet endpoints. Include end-to-end tests: e.g. simulate a user chat request and verify the backend calls the correct tool. Ensure at least 80% code coverage on critical modules.
14. **Documentation:** Maintain developer docs (e.g. in Notion or Markdown) describing architecture diagrams, data flow, API contracts (Swagger). Provide example API contract for a smart contract tool if supporting on-chain (e.g. a Solidity ABI schema). Outline user stories for investor demos.
15. **Security Review:** Before launch, conduct a threat model walkthrough (using STRIDE or Smithery tool). Verify no secrets in source. Ensure all dependencies are up-to-date and scanned. Perform a penetration test on authentication endpoints.

Checklist Before Launch

- [ ] **Code & PRs:** All features merged and tested. Linting/type-check passing.
- [ ] **Secrets:** No keys in repo. Production keys stored in vault or environment only.
- [ ] **CI/CD:** Pipelines green; deployment process verified.
- [ ] **Deployment:** Backend and frontend running on staging (Fly.io).
- [ ] **Security:** Sentry and security scans active. RLS policies in Supabase tested.
- [ ] **Risk Controls:** Max trade sizes, stop-loss rules configured. Emergency “panic button” endpoint implemented.
- [ ] **Monitoring:** Sentry dashboards set; log aggregation checked. Alerts configured.
- [ ] **Documentation:** Whitepaper-style docs (this report), API reference, architecture diagrams, go-to-market notes.
- [ ] **Demo:** Prepare investor demo (show UI, discuss architecture merits and competitive advantage).

Conclusion

This design delivers a **robust, secure, LLM-powered crypto trading assistant**. By combining proven cloud components (FastAPI, Supabase, Fly.io) with Google Gemini and LangChain, we can offer advanced market analysis and automated trades, while ensuring safety (encrypted keys, limited scopes) and observability (Sentry, logs). Our architecture is informed by industry use cases (Bybit’s TradeGPT, PAAL AI, ChainGPT) [17] [15] but is tailored for adaptability and enterprise readiness. The modularity allows swapping exchanges or models, and the focus on CI/CD and security will reassure stakeholders. All elements – from blockchain-specific tools to compliance logging – are documented and justified, making this system investor-ready and future-proof.

Sources: We drew on recent engineering guides and articles (e.g. on LLM trading agents and system design) [4] [21] [10] [17] [8], combining best practices for secure key storage, event-driven trading loops, and cloud-native deployment. Each design choice above is grounded in these industry references.

---

1 2 3 4 15 16 17 18 19 20 Building a Multimodal AI Trading Assistant for Cryptocurrency — A Manus AI-Inspired Framework | by Jung-Hua Liu | Medium
https://medium.com/@gwrx2005/building-a-multimodal-ai-trading-assistant-for-cryptocurrency-a-manus-ai-inspired-framework-6e098fc9b3f9

5 How Should You Store and Access API Keys in Your Application?
https://www.tokenmetrics.com/blog/best-practices-storing-accessing-api-keys-applications?0fad35da_page=5&74e29fd5_page=5

6 Error and Performance Monitoring for React | Sentry
https://sentry.io/for/react/

7 Python Error Tracking and Performance Monitoring | Sentry
https://sentry.io/for/python/

8 9 Architecture | Supabase Docs
https://supabase.com/docs/guides/getting-started/architecture

10 Crypto Trading Bot: Architecture and Roadmap | by Vitalii Honchar | Medium
https://vitalii-honchar.medium.com/crypto-trading-bot-architecture-and-roadmap-f3e26cf9956a

11 12 13 21 How to Day Trade Crypto With Google Gemini AI
https://cointelegraph.com/news/how-to-day-trade-crypto-using-google-s-gemini-ai

14 Cryptocurrency Security Threat Modeling: Beyond Vulnerabilities | NopSec
https://www.nopsec.com/blog/cryptocurrency-security-threat-modeling-beyond-vulnerabilities/