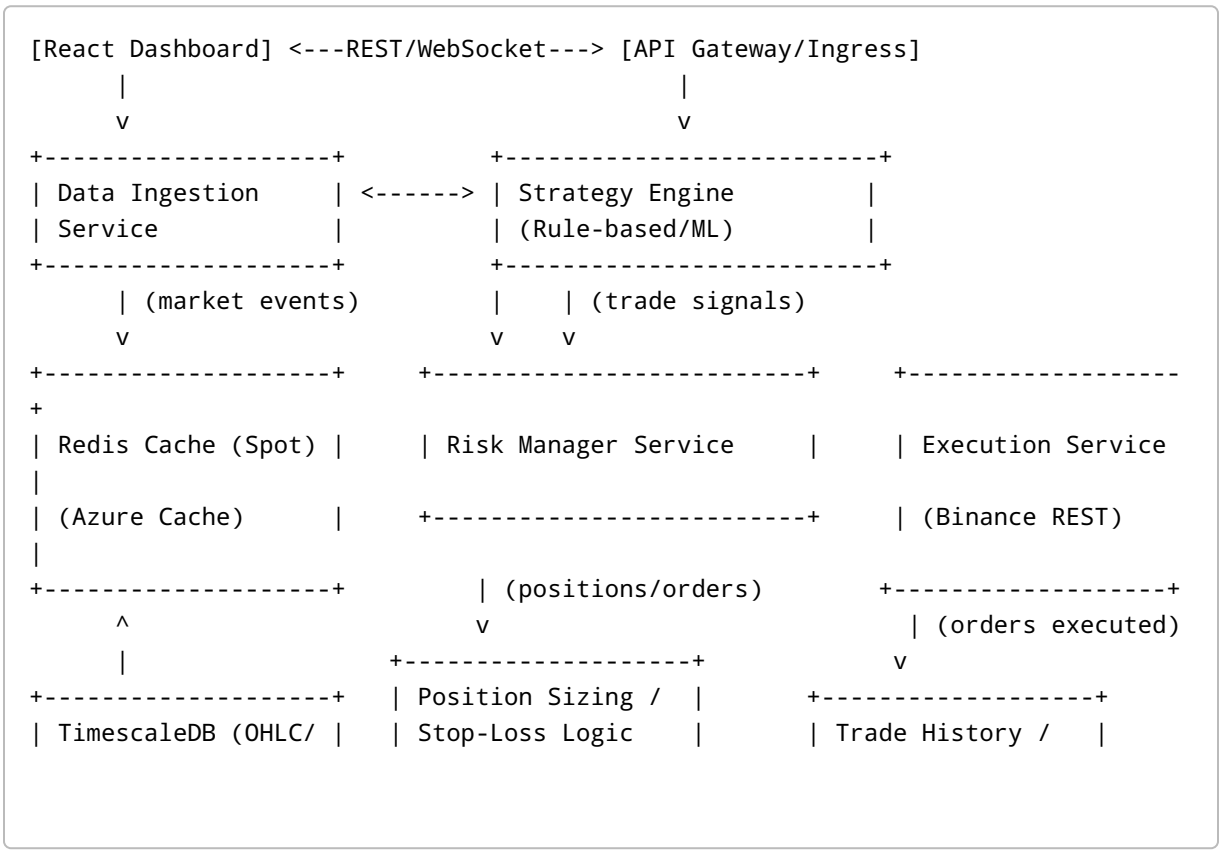
ChatGPT

Elite Crypto Trading Bot – Architecture & Design

This document details the architecture and logic of a high-performance, event-driven crypto trading bot built on **.NET Core 8 LTS**, Azure services, and a React frontend. It outlines each microservice, data flow, technology choice, and risk/control mechanisms. The system ingests real-time market data (via Binance WebSocket/REST and free crypto APIs), processes it in a modular microservices pipeline (data ingestion, strategy engine, risk manager, execution engine), and stores data in TimescaleDB (PostgreSQL) and Redis (Azure Cache). A secure React dashboard provides real-time monitoring, logs, and controls. Azure Kubernetes Service (AKS) and Docker ensure containerized scalability; CI/CD pipelines automate builds/deployments; Prometheus/Grafana and OpenTelemetry enable observability. Key security measures include Azure Key Vault for secrets, TLS encryption, and OAuth2-based login.

High-Level System Architecture

The bot follows a **microservices architecture** deployed on AKS, with each component in a Docker container. Components communicate via an API gateway/ingress and asynchronous messaging (e.g. Azure Service Bus) for an event-driven design [1] [2]. An example flow is illustrated below (arrows indicate data/event flow):



```
| Historical data)   |   +--------------------+       | Analytics Storage |
+--------------------+                                 +-------------------+
```

- **Data Ingestion Service:** Connects to Binance (testnet) via WebSocket and REST, as well as crypto data APIs (CoinGecko, CoinMarketCap, CryptoPanic). It processes incoming ticks and news, writing time-series data into TimescaleDB and hot data into Redis [3] [4], and publishes events to downstream services. This ensures real-time market updates and historical archiving.
- **Strategy Engine:** Implements rule-based (e.g. moving averages, momentum) and optional ML-based strategies. It subscribes to market events, evaluates entry/exit conditions, and emits trade signals. (Optional: an ML component can be trained offline on TimescaleDB data to refine signals.)
- **Risk Manager:** Applies strict risk controls on signals: it computes position sizing (e.g. limiting risk per trade to 1–2% of capital) and enforces stop-loss/drawdown rules [5]. Signals breaching limits are blocked. Approved signals pass through with computed order sizes.
- **Execution Service:** Receives approved trade instructions, connects to Binance REST API (HMAC-authenticated) to place or cancel orders. It monitors order status and logs fills, cancellations, or errors. Each trade and updated portfolio state is stored in TimescaleDB.
- **Data Stores:** TimescaleDB (PostgreSQL) holds all tick data and completed trade history. TimescaleDB is optimized for time-series analytics – it can ingest ~1–2M ticks/sec with 10–100ms query latency, making it ideal for backtesting and metrics [3] [4]. Redis (via Azure Cache) provides in-memory caching for the latest quotes, orderbooks, and quick lookups, with sub-millisecond latency [6] [7].
- **Monitoring & Logging:** All services emit metrics and health info (e.g. counters, latencies, errors) via OpenTelemetry. Prometheus scrapes these metrics (exposed on `/metrics` endpoints) and Grafana dashboards visualize them [8] [9]. Distributed traces (instrumented via ActivitySource) are sent to Jaeger (for example) to diagnose cross-service latency [10] [8]. Real-time alerts (e.g. excessive error rates or failed health checks) can be configured.
- **React Dashboard:** A secure single-page app that shows live trade logs, performance charts, and strategy insights. It uses OAuth2 (e.g. Azure AD B2C via MSAL) for user authentication [11]. The dashboard polls or subscribes to backend events (via REST or WebSocket) to update charts and tables in real time. Admin users can adjust strategy parameters or reset components through this UI.

This architecture is horizontally scalable and resilient. AKS hosts the containerized microservices, providing managed Kubernetes orchestration. AKS "provides the runtime platform for deploying, scaling, and managing … microservices" [2]. Each service is stateless and can scale independently; state (like market data or positions) resides in external stores (TimescaleDB, Redis, Azure Service Bus) [12]. Azure-managed services (Key Vault, Redis, PostgreSQL) ensure high availability and reduce operational burden. A CI/CD pipeline automates building Docker images and deploying to AKS (see below) [13] [14].

Module Logic and Data Flows

Data Ingestion Service

- **Function:** Maintain a persistent connection to Binance's WebSocket (testnet endpoint `wss://testnet.binance.com:9443/ws`) to receive real-time price/tick updates [15]. Periodically fetch account snapshots or missing data via Binance REST. Concurrently poll free crypto APIs: CoinGecko/CoinMarketCap for reference prices and market cap, CryptoPanic for news/sentiment, and optionally Glassnode for on-chain metrics.

- **Logic:** Parse incoming messages (JSON), normalize data, and write to stores. For each tick or relevant news item, the service writes the raw event to TimescaleDB (hypertables partitioned by symbol/time) and updates any relevant cached state in Redis (e.g. current best bid/ask, 24h high/low). It also publishes structured events to an internal event queue (e.g. Azure Service Bus) so that downstream services (strategy engine, monitors) can consume them asynchronously [1]. The service handles reconnection logic and backoff for APIs.
- **Outputs:** Persisted raw market events (TimescaleDB, Redis), and published events (via Service Bus or in-memory queue) for the Strategy Engine. Example output: a "BTCUSDT tick" event containing price, volume, timestamp.

Strategy Engine

- **Function:** Analyze incoming market events against trading strategies. Supports multiple independent strategies (e.g. trend-following, mean-reversion). Optionally integrate ML models (e.g. TensorFlow or ML.NET) to refine or generate signals.
- **Logic:** The engine subscribes to the event stream from the ingestion service. It maintains any needed state (e.g. moving average windows) or runs ML inference on recent data. When a strategy's conditions are met, it emits a **trade signal** (e.g. "Buy 0.1 BTC at market"). Signals include symbol, side, quantity, and any stop-loss/take-profit levels. Signals are published to the Risk Manager service. The engine logs decision metrics (e.g. indicators values) to TimescaleDB for analysis.
- **Outputs:** Trade signals (sent to Risk Manager). For example: `{ symbol: "BTCUSDT", action: "BUY", quantity: 0.1, strategy: "MA_Crossover" }`.

Risk Manager

- **Function:** Enforce risk controls on signals. Implements stop-loss, max drawdown, and position-sizing rules to protect capital.
- **Logic:** On receiving a signal, the Risk Manager checks current P&L and exposure (from Redis or DB). It calculates an appropriate position size (e.g. risk 1% of equity, using distance to a trailing stop) [5]. It then applies drawdown limits (e.g. if cumulative loss >5%, halt trading) and verifies no position exceeds max leverage. If all checks pass, the manager forwards the order to the Execution service; otherwise, it rejects or modifies the order. Alerts (e.g. "drawdown limit reached") can be logged or sent to the dashboard.
- **Outputs:** Approved or modified order instructions. Example: `{ symbol: "BTCUSDT", side: "BUY", quantity: 0.08, stop_loss: 55000 }`. It also updates Redis/TimescaleDB with new position sizes and risk metrics.

Execution Service

- **Function:** Place and manage orders on Binance.
- **Logic:** Listens for approved orders from the Risk Manager. For each, it calls the Binance REST API to create, cancel, or modify orders. Requests are signed with API key/secret (securely loaded from Azure Key Vault [16]). The service handles API responses: on fill or rejection, it records order status in TimescaleDB. If a stop-loss or take-profit is hit, it sends a cancellation or reverse order as needed. Execution is asynchronous: placed orders are tracked until complete.
- **Outputs:** Actual trade executions and status. For example, on a successful buy: a DB record in TimescaleDB: `{ orderId, symbol, side, price, quantity, timestamp }`. It also emits fill events to update positions and notify the React dashboard (showing trade logs and updated equity).

Monitoring & Logging

- **Function:** Provide observability for all services.
- **Logic:** Each microservice is instrumented with OpenTelemetry: metrics (request counts, latencies, custom counters) and traces are recorded. A Prometheus exporter exposes these metrics (e.g. via an HTTP `/metrics` endpoint) [8] [10]. Prometheus (running in AKS or on Azure) scrapes these endpoints at intervals. Grafana dashboards (connected to Prometheus) display key metrics (e.g. trades/sec, latencies, resource usage) and health checks [9]. For example, health endpoints (`/health/live`, `/health/ready`) integrate with Kubernetes probes and with Prometheus alerts. Jaeger (or Zipkin) collects traces so one can trace a request through ingestion → strategy → risk → execution. This aids debugging and performance tuning.
- **Outputs:** Time-series metrics and alerts in Prometheus/Grafana; traces in Jaeger; log files. For example, Grafana panels might show "24h P&L", "API call success rate", or "Kafka lag" (if using event streaming).

React Dashboard (UI)

- **Function:** Provide a web UI for monitoring and control.
- **Logic:** The React app (SPA) fetches data from the backend (via REST or GraphQL) and subscribes to real-time updates (e.g. via WebSocket or server-sent events) for live price charts and orderbooks. It displays trade logs, performance stats (P&L, win-rate), and key indicators per strategy. Admin pages allow tuning parameters (risk limits, strategy toggles). All API calls are secured by OAuth2; we use Azure AD B2C for user authentication with MSAL in React [11]. Sensitive operations (changing settings) require admin role. Communication uses HTTPS/TLS for security.
- **Outputs:** Rich UI elements (charts, tables, alerts). For example, a real-time chart showing historical price vs executed trades, a table of current positions, and logs of recent signals/trades with color-coded P&L.

Infrastructure Components (Azure Cloud)

- **Azure Kubernetes Service (AKS):** Hosts all microservices as containers. AKS is a *fully managed Kubernetes service* that automatically provisions nodes, handles upgrades, and scales pods. It "provides the runtime platform for deploying, scaling, and managing … microservices" [2]. We use AKS because it supports auto-scaling, rolling updates, and integrates with Azure networking (Cilium or Flannel CNI) for performance.
- **Azure Container Registry (ACR):** Stores private Docker images. CI pipelines push built images to ACR, and AKS pulls these for deployments.
- **Azure Cache for Redis:** A managed Redis instance for caching. Azure's offering "is backed by open-source Redis and natively supports Redis data structures [6]." It provides sub-millisecond response and supports TLS/SSL for secure client connections [7]. We use Redis to store ephemeral data (e.g. latest prices, orderbook snapshots, session state) and to implement a fast pub/sub bus if needed.
- **TimescaleDB (PostgreSQL on Azure):** A time-series optimized database. We plan to use Azure Database for PostgreSQL with the Timescale extension. TimescaleDB excels at storing time-series (market ticks, candles) and performing analytics. It can ingest millions of data points per second and serve queries in milliseconds [4] [3]. Timescale's SQL interface allows complex queries (e.g. "average volume in last hour" or backtesting signals) while maintaining relational capabilities.
- **Azure Service Bus / Event Hubs:** For event-driven messaging, we employ Azure Service Bus queues or Azure Event Hubs. For example, ingestion publishes market events into Service Bus queues that

strategy and monitoring services subscribe to. This decouples components and provides reliable delivery. (The Microsoft sample architecture uses Service Bus in a similar way [1].)

- **Azure Key Vault:** All secrets (API keys for Binance, tokens for third-party APIs, certificates) are stored in Key Vault. Vault "is a cloud service for securely storing and accessing secrets" [16]. Microservices authenticate to Key Vault using managed identities, pulling credentials at runtime rather than hardcoding them. This ensures keys are never checked into code.
- **Identity & Access:** We use Azure Active Directory (Entra) for service identities. AKS nodes use managed identities to pull container images and access other Azure resources. The React frontend uses Azure AD B2C for user login (OAuth2) as described above [11].

CI/CD Pipelines

We implement automated CI/CD using GitHub Actions or Azure DevOps pipelines. Each commit triggers a build: it compiles .NET services, runs unit/integration tests, builds Docker images, and pushes them to ACR. A deployment workflow then applies Kubernetes manifests (or Helm charts) to update AKS. GitHub Actions have ready-made Azure AKS deployment templates [13]. For example, a GitHub Action can `docker build`, `docker push`, then use `kubectl` to update deployments in AKS. This continuous deployment ensures rapid, consistent updates; teams can "independently build and deploy microservices to AKS by using Azure Pipelines" [14]. Rollbacks are supported via versioned images.

Component Flows (Technical Diagrams)

Below is an ASCII flow diagram illustrating the key data paths. Each arrow denotes data flow or control messages:

```
[React Dashboard (User)]
       |  (HTTPS/WebSocket)
       v
[Ingress/API Gateway]
   |       |       |
   |       |       +--> [Data Ingestion Service]
   |       |             (Subscribes Binance WS, polls APIs, writes to DB/Redis,
publishes events)
   |       |                      |
   |       |                      v
   |       +--> [Strategy Engine] <---+
   |       |       (Consumes market events,
   |       |        generates trade signals)
   |       |             |
   |       |             v
   |       +--> [Risk Manager Service]
   |       |       (Applies size/stop/drawdown rules,
   |       |        approves/rejects signals)
   |       |             |
   |       |             v
   |       +--> [Execution Service]
```

```
|               (Places orders on Binance via REST,
|                logs fills/cancels)
|
+--> [Prometheus/Grafana]
        (Scrapes metrics, serves dashboards)
```

Each service may also access: - **TimescaleDB** (for historical data and trade logs). - **Azure Cache (Redis)** (for current prices, state, pub/sub). - **Azure Key Vault** (for secrets like API keys).

In summary, market data flows from Binance (and other APIs) into the system, moves through strategy → risk → execution, and results (trades, logs) go back to storage and UI. Prometheus periodically scrapes metrics from each service (at `/metrics`), while the dashboard receives live trade updates via REST or WebSocket pushed from services.

Design Justifications

- **Microservices & AKS:** This modular approach allows independent development, deployment, and scaling of each component (ingestion, strategy, risk, execution, etc.). AKS's managed Kubernetes provides auto-scaling and high availability. For example, if market data volume spikes, we can scale out the ingestion pods without affecting others. AKS "hosts and orchestrates the microservices containers" [2], and stateless design means pods are easily replaced. Kubernetes service discovery and ingress controllers simplify routing. This design supports extreme throughput and resilience. ASP.NET Core itself is highly performant: independent benchmarks show it is **one of the fastest web frameworks** for API servers [17]. Its async model (async/await) handles thousands of concurrent connections with minimal overhead [18]. Using .NET also means seamless integration with Azure and full control over infrastructure.
- **TimescaleDB + Redis:** Splitting data storage by use-case improves performance. TimescaleDB is optimized for historical time-series queries; as a PostgreSQL extension, it supports complex SQL analytics [3] [4]. Redis handles real-time caching and pub/sub: its in-memory speed ensures the latest prices and positions are accessible with low latency. Azure's managed Redis "improves throughput with sub-millisecond latency" [6] and supports TLS [7] for secure transport. Decoupling reads/writes via cache and database prevents bottlenecks.
- **Event-Driven Messaging:** Using a message bus (Azure Service Bus or Event Hubs) decouples services. The ingestion service can publish tick events even if the strategy engine is busy; events are buffered in the queue [1]. This also aids reliability (messages survive transient faults) and allows multiple consumers (e.g. a logging service or parallel strategies) to subscribe. It simplifies scaling: multiple strategy-engine instances can consume from the same queue if needed.
- **Risk Controls:** Built-in risk modules enforce trading discipline, crucial for capital preservation. Automated stop-loss and max-drawdown are industry best practices [19]. For instance, setting a hard drawdown limit (e.g. 5% of account) ensures catastrophic losses are avoided. Position sizing (e.g. 1% per trade) prevents oversized bets [20]. These controls run independently of strategy logic, making the system robust against strategy failures.
- **CI/CD and DevOps:** Automated pipelines ensure reproducibility and quick recovery. Each microservice is built, tested, and deployed through the pipeline [13]. This aligns with cloud-native best practices: any commit can be swiftly promoted to production with zero-downtime rolling updates. Versioning and containerization make maintenance easier.

- **Observability:** Continuous monitoring is essential for a live trading system. By integrating OpenTelemetry, Prometheus, and Grafana, we get real-time visibility into system health and performance [9] [10]. For example, Grafana dashboards can display latency of Binance API calls or memory usage of each pod. Alerting (e.g. via Prometheus rules) can notify operators of anomalies (network errors, slow database queries, etc.) before they impact trading. .NET tracing helps quickly pinpoint where delays occur in the service chain.

Integration with Data Providers (Free → Premium)

Initially, the bot uses **free data sources** for development and basic operation: - **Binance Testnet:** For safe testing of order execution without real funds (WebSocket and REST) [15]. - **CoinGecko API:** Wide coverage of coins, generous free tier [21]. - **CoinMarketCap API:** Industry-standard market data, free tier available [22]. - **CryptoPanic API:** Aggregated crypto news and sentiment. (CryptoPanic "delivers real-time crypto news, sentiment analysis, and market-moving alerts" [23].) - **Glassnode API (optional):** For on-chain analytics; its free tier provides some metrics and the professional tier unlocks ~7,500 on-chain indicators [24].

As the system matures, we plan to migrate to **premium data feeds** for reliability and depth. Paid APIs offer higher rate limits and advanced data (full order books, historical ticks, institutional-grade metrics) [25]. For example, upgrading to CoinMarketCap Pro or a dedicated feed (e.g. Kaiko) would eliminate rate-limit constraints. Premium subscriptions (like Glassnode Studio Enterprise) grant richer on-chain metrics and SLAs [25] [24]. In practice, we would abstract data-provider interfaces so we can swap sources. The architecture supports parallel data streams, so premium feeds can be introduced without downtime.

Security Considerations

- **Secrets Management:** All API keys, credentials, and TLS certificates are stored in **Azure Key Vault**, not in code. Key Vault is "a cloud service for securely storing and accessing secrets" (like API keys) [16]. Microservices access these secrets at runtime using managed identities, ensuring keys remain protected.
- **Encryption (TLS):** All communication is encrypted. The dashboard and APIs use HTTPS/TLS. Internal service calls (e.g. to Redis, to Binance, between pods) also use TLS or secure channels. Azure Cache for Redis enforces TLS for in-transit data [7]. Certificates (for TLS termination at ingress) are managed by Azure (Application Gateway or Ingress controllers).
- **Authentication/OAuth2:** The React frontend uses OAuth2 flows for user login (via Azure AD B2C or ADFS). MSAL in React handles token acquisition [11]. This provides standard security (no handling of passwords in our code). API endpoints validate JWT tokens and enforce RBAC (regular user vs admin). Binance API keys (client ID/secret) are never exposed to the client side; trading orders originate from the trusted backend only.
- **Network Security:** Services run in a secured Azure VNet. Pod networking is isolated via Azure CNI. Ingress rules restrict API access to necessary ports. Azure private endpoints can be used to connect to managed services (Redis, DB) without public internet exposure.
- **Logging & Auditing:** All trades and configuration changes are logged (with immutable storage or append-only logs), enabling audit trails. Azure Application Insights or Log Analytics can store logs centrally. Failed logins or anomalous API usage trigger alerts.

Expected Outputs per Component

- **Data Ingestion:** Continuously outputs raw market data (ticks, orderbooks) to TimescaleDB and Redis; emits parsed events (e.g. new ticker) to the message bus. Expected output example: a new row in TimescaleDB every tick, and a Redis key update for the latest price.
- **Strategy Engine:** Emits trade signals based on configured rules. Expected outputs include structured messages like *"BUY 0.5 BTC at market"* or *"SELL ETH if price > $4,000"*. Additionally, it may output analytic metrics (e.g. indicator values) to TimescaleDB for backtesting analysis.
- **Risk Manager:** Produces approved, sized orders or rejection notices. For each signal, the Risk Manager outputs either an executable order (e.g. `{symbol: "BTCUSDT", side: "BUY", quantity: 0.07}`) or a logged warning (e.g. *"Signal halted: drawdown limit reached."*).
- **Execution Service:** Generates trade execution records. For each order placed, the system records an output trade with details (filled quantity, price, timestamp) in TimescaleDB. It also updates on-chain logs or portfolio snapshots. These outputs feed into the dashboard's trade log.
- **Monitoring:** Continuously emits metric data points (CPU/memory usage, request rates, error counts) to Prometheus, and health statuses. Expected outputs include Prometheus time-series and Grafana alerts (e.g. "HTTP 500 error rate >5%").
- **React Dashboard:** Renders tables and charts based on data from the backend. Outputs are visual: e.g. a candlestick chart of BTC price with overlaid trade markers, a table of executed trades with realized P&L, and a panel showing current total equity and strategy performance metrics. The dashboard also outputs user-initiated commands (like *"update risk limit to 3%"*) back to the backend via secure API calls.

Conclusion

This design leverages Azure's managed services and proven architectural patterns to build a robust, high-throughput crypto trading bot. The **microservices** separation ensures each module can focus on its logic (data ingestion, strategy, risk, execution, UI) while scaling independently. Using **.NET Core** provides high performance and seamless cloud integration [17]. **TimescaleDB** and **Redis** handle the data layers for analytics and real-time state [3] [6]. As needs grow, modularity allows upgrading data feeds or strategies without rearchitecting the system. Security is built-in with Azure Key Vault and OAuth2 [16] [11]. Finally, continuous monitoring (Prometheus/Grafana, OpenTelemetry) and automated CI/CD ensure the system remains reliable and maintainable in production.

Sources: Official Azure and technology documentation and industry best practices (see citations). All architecture decisions prioritize **scalability, performance, and maintainability** while enforcing strict risk controls and security.

---

[1] [2] [12] [14] Microservices Architecture on Azure Kubernetes Service - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/containers/aks-microservices/aks-microservices

[3] GitHub - timescale/timescaledb: A time-series database for high-performance real-time analytics packaged as a Postgres extension
https://github.com/timescale/timescaledb

4 kdb+ vs TimescaleDB for Market Data: 2025 Performance & Cost Guide | sanj.dev
https://sanj.dev/post/kdb-vs-timescaledb-market-data-comparison

5 19 20 7 Risk Management Strategies for Algorithmic Trading
https://nurp.com/algorithmic-trading-blog/7-risk-management-strategies-for-algorithmic-trading/

6 Azure Cache for Redis | Microsoft Azure
https://azure.microsoft.com/en-us/products/cache

7 Azure Cache for Redis: Beginner's Guide | Redwerk
https://redwerk.com/blog/azure-cache-for-redis-beginners-guide/

8 9 ASP.NET Core Health Checks and Monitoring with Prometheus + Grafana
https://www.c-sharpcorner.com/article/asp-net-core-health-checks-and-monitoring-with-prometheus-grafana/

10 Example: Use OpenTelemetry with Prometheus, Grafana, and Jaeger - .NET | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/core/diagnostics/observability-prgrja-example

11 Enable authentication in a React application by using Azure Active Directory B2C building blocks | Microsoft Learn
https://learn.microsoft.com/en-us/azure/active-directory-b2c/enable-authentication-react-spa-app

13 Deploy a Cloud-Native .NET Microservice Automatically with GitHub Actions and Azure Pipelines - Training | Microsoft Learn
https://learn.microsoft.com/en-us/training/modules/microservices-devops-aspnet-core/

15 Websocket API General Info | Binance Open Platform
https://developers.binance.com/docs/derivatives/usds-margined-futures/websocket-api-general-info

16 Key Vault | Microsoft Azure
https://azure.microsoft.com/en-us/products/key-vault

17 18 Why .NET Is Ideal for High-Performance API Development
https://metadesignsolutions.com/why-net-is-ideal-for-high-performance-api-development/

21 22 Best Crypto APIs in 2026: Comprehensive Guide
https://nownodes.io/blog/best-crypto-apis-in-2025-comprehensive-guide/

23 CryptoPanic | Real-Time Crypto News & Market Sentiment
https://app.coinpedia.org/company/cryptopanic/

24 Comparative Analysis of Messari API and Glassnode API | by Jung-Hua Liu | Jan, 2026 | Medium
https://medium.com/@gwrx2005/comparative-analysis-of-messari-api-and-glassnode-api-d5e600f4038d

25 Are Crypto APIs Free or Paid? Cost, Features, and Comparisons
https://www.tokenmetrics.com/blog/crypto-apis-free-vs-paid-options?74e29fd5_page=123