Oracle System

[Technical Approach + Design]

Nattapat Iammelap

Executive Summary

Objective

Develop a versatile oracle system to fetch real-time cryptocurrency prices from various sources to support multiple clients with diverse asset requirements. The system should achieve the following goals:

- Retrieve and transmit real-time cryptocurrency prices from Decentralized Exchanges (e.g., Uniswap), Centralized Exchanges (e.g., Binance), and Price Aggregator Services (e.g., CoinMarketCap, CoinGecko).
- Provide services for multiple clients to fetch and transmit real-time cryptocurrency prices tailored to distinct needs.

Key System Requirements

To achieve the stated objectives, the system must be capable of the following:

- Support Multiple Protocols for Data Retrieval
- Offer Flexible Query Connections for Clients
- Maintain Source, System and Client Independence
- Guarantee Data Accuracy and System Resilience

System Design

The overall system consists of the following components

Data Sources

The **Data Sources** component represents the external providers from which the system fetches cryptocurrency price data. These include:

- **Decentralized Exchanges (DEXs):** Examples include Uniswap and PancakeSwap, which provide data via blockchain smart contracts or APIs.
- **Centralized Exchanges (CEXs):** Examples include Binance and Coinbase, which use REST APIs or WebSocket streams.
- **Price Aggregator Services:** Examples include CoinMarketCap and CoinGecko, which consolidate data from multiple exchanges into accessible APIs.
- **Specialized Sources:** This may include custom APIs, web scraping, or other external data pipelines.

The Data Sources component plays a critical role in supplying cryptocurrency pricing data to the system. Its key responsibilities include:

- Serve as the primary input for cryptocurrency pricing data.
- Provide diverse data formats and protocols that require normalization and standardization downstream.

Oracle Core

The **Oracle Core** is the central processing and business logic layer responsible for managing, aggregating, and distributing data. Key functionalities include:

- **Data Aggregation:** Merge data from multiple sources to create a unified and consistent output.
- **Real-Time Processing:** Process data to ensure clients receive accurate, low-latency results.
- **Source Management:** Monitor the health of data sources and address failures by switching to alternatives or applying methods like data weighting to reconcile discrepancies.
- **Transformation and Enrichment:** Scale, normalize, or enrich the data to meet client-specific requirements.

Database and Cache

The **Database and Cache** ensures efficient storage and quick access to real-time and historical data.

- **Database:** A durable backend storage solution (e.g., Timescale or MongoDB) for maintaining historical data and logs.
- **Cache:** In-memory databases (e.g., Redis) to store frequently accessed real-time data, ensuring low latency for client queries.

This components are essential for efficient data storage and retrieval, with the following key functions:

- Support both real-time data retrieval and longer-term historical analysis.
- Provide redundancy and resilience through replication and backup mechanisms.

API Gateway

The **API Gateway** manages all client interactions with the system. Key responsibilities include:

- **Request Management:** Balance loads across system components and handle a high volume of concurrent requests.
- **Routing:** Direct requests to the appropriate internal services (e.g., public query or custom client endpoints).
- **Public Query Interface:** Provide flexible APIs to cater to general clients with diverse needs.
- **Custom Client Endpoints:** Offer specialized query mechanisms for clients with unique requirements.
- **Security:** Ensure secure access through authentication, authorization, and rate-limiting.

- **High Availability:** Implement retry mechanisms and circuit breakers for resilient client interactions.

Data Retriever

The **Data Retriever** is a dedicated service to fetch data from external sources.

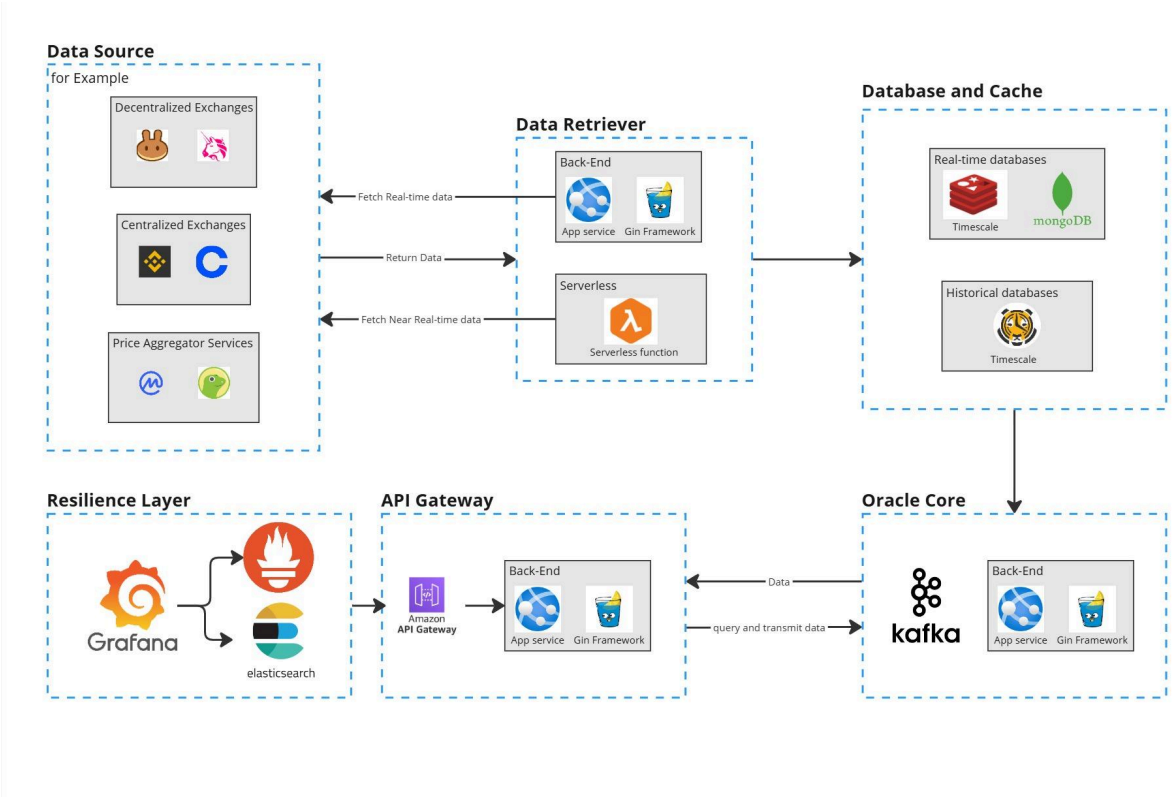
- **Protocol Support:** Handle diverse protocols, including REST APIs, WebSocket streams, and blockchain interactions.
- **Data Normalization:** Standardize data from different sources into a uniform format for the Oracle Core.
- **Specialized Data Retrieval:** Develop custom solutions for non-standard sources, including web scraping if necessary.
- **Resilience:** Implement retries, fallback mechanisms, and source prioritization to ensure continuous data availability.

Error Handling/Resilience Layer

The **Error Handling/Resilience Layer** ensure system observability and robust operations.

- **Health Checks:** Monitor the status of all components, including Data Sources, API Gateway, and Oracle Core.
- **Error Handling:** Detect failures and trigger appropriate fallback mechanisms, such as switching to alternate sources or notifying administrators.
- **Event Logging:** Record all system events, including requests, errors, and data discrepancies, for debugging and compliance.
- **Real-Time Monitoring and Alerts:** Notify administrators of anomalies or failures through integrated alerting systems.
- **Resilience Layer:** Incorporate circuit breakers, failover strategies, and automatic recovery mechanisms.

System Architecture



Client Interaction Mechanisms

Mechanism for Client Queries

The **Mechanism for Client Queries** is divided into two parts: **Public Query Interface** and **Custom Client Endpoint**. The Public Query Interface will offer both RESTful APIs and GraphQL endpoints to cater to different client needs.

1. Public Query Interface

The **Public Query Interface** will provide flexible and scalable methods for clients to retrieve cryptocurrency pricing data from various sources. To meet a wide range of requirements, the system will offer two options: **REST API** and **GraphQL API**.

REST API

The REST API will allow clients to make HTTP requests and retrieve real-time cryptocurrency data in a simple and structured JSON format. A client can query data from specific sources or aggregate data from multiple sources by providing parameters in the query string.

Example Endpoint (REST API):

```
GET
/api/cryptocurrency-price?asset={asset_name}&source={source_name}
```

Response (JSON format):

```
{
  "asset": "bitcoin",
  "price": 90000,
  "unit": "USD",,
  "source": "Binance",
  "timestamp": "2024-11-18T12:00:00Z"
}
```

Clients can specify which source to retrieve data from using the query string. If no source is specified, the system may aggregate data from multiple sources and provide a weighted average or fallback data from a default source.

GraphQL API

For more complex use cases, the **GraphQL API** will provide flexibility in querying. Clients can request exactly the data they need and even combine information from multiple sources in a single query. The simplicity and adaptability of GraphQL make it ideal for clients who need customized data structures.

Example GraphQL Query:

```
query {
  getPrice(asset: "bitcoin", sources: ["Binance", "CoinGecko"]) {
    asset
    price
    unit
    source
    timestamp
  }
}
```

Response (JSON format):

```
{
  "data": {
    "getPrice": [
      {
        "asset": "bitcoin",
        "price": 90000,
        "unit": "USD",
```

```
        "source": "Binance",
        "timestamp": "2024-11-18T12:00:00Z"
      },
      {
        "asset": "bitcoin",
        "price": 91000,
        "unit": "USD",
        "source": "CoinGecko",
        "timestamp": "2024-11-18T12:00:00Z"
      }
    ]
  }
}
```

In this approach, clients can select multiple sources and assets in a single query. This flexibility allows clients to tailor their requests according to their needs, reducing the complexity of multiple API calls.

asset: The name or symbol of the cryptocurrency (e.g., Bitcoin, Ethereum).
price: The real-time price of the asset.
unit: The unit of currency used for the price (e.g., USD, EUR, etc.).
source: The source from which the data was fetched (e.g., Binance, CoinGecko).
timestamp: The time when the data was retrieved.

For real-time updates, we can use **WebSocket** with **REST API** and **GraphQL subscriptions** for **GraphQL** to receive updates whenever the price changes.

2. Custom Client Endpoint

The **Custom Client Endpoint** will provide tailored functionality for clients with specific needs not covered by the public query interface. The details of these endpoints will be flexible, depending on the unique requirements of each client. These endpoints may include custom data formats, source preferences, or other specialized features based on the client's use case.

Transmitting Cryptocurrency Prices to the Oracle

The mechanism for transmitting cryptocurrency prices to the oracle will allow clients to submit real-time price data through **RESTful APIs**, **GraphQL mutations**, and **WebSocket connections**. The oracle will store this data in both a **real-time database** and a **historical database** for future analysis. **Authentication** will be required for all data submissions to ensure secure and authorized data transmission.

1. Public Data Transmission Interface

Clients will be able to transmit cryptocurrency prices to the oracle system via REST API or GraphQL mutations. These methods will allow for easy and structured submission of data to the oracle, which will then be stored in the appropriate databases.

REST API

The **REST API** will allow clients to send HTTP requests to submit cryptocurrency price data to the oracle. The request will include the asset name, price, source, unit, and timestamp as part of the payload. Authentication is required using API keys or OAuth tokens.

Example Endpoint (REST API):

```
POST /api/cryptocurrency-price
```

Request Payload (JSON format):

```
{
    "asset": "bitcoin",
    "price": 90000,
    "unit": "USD",
    "source": "ClientA",
    "timestamp": "2024-11-18T12:00:00Z"
}
```

GraphQL Mutations

Clients can also submit price data using **GraphQL mutations**, which allow clients to send data in a structured query format. Similar to the REST API, authentication will be required.

Example GraphQL Mutation:

```
mutation {
  transmitPrice(
    asset: "bitcoin",
    price: 90000,
    unit: "USD",
    source: "ClientA",
    timestamp: "2024-11-18T12:00:00Z"
  ) {
    asset
    price
    unit
    source
    timestamp
  }
}
```

Response (JSON format):

```
{
  "data": {
    "transmitPrice": {
      "asset": "bitcoin",
      "price": 90000,
      "unit": "USD",
      "source": "ClientA",
      "timestamp": "2024-11-18T12:00:00Z"
    }
```

```
  }
}
```

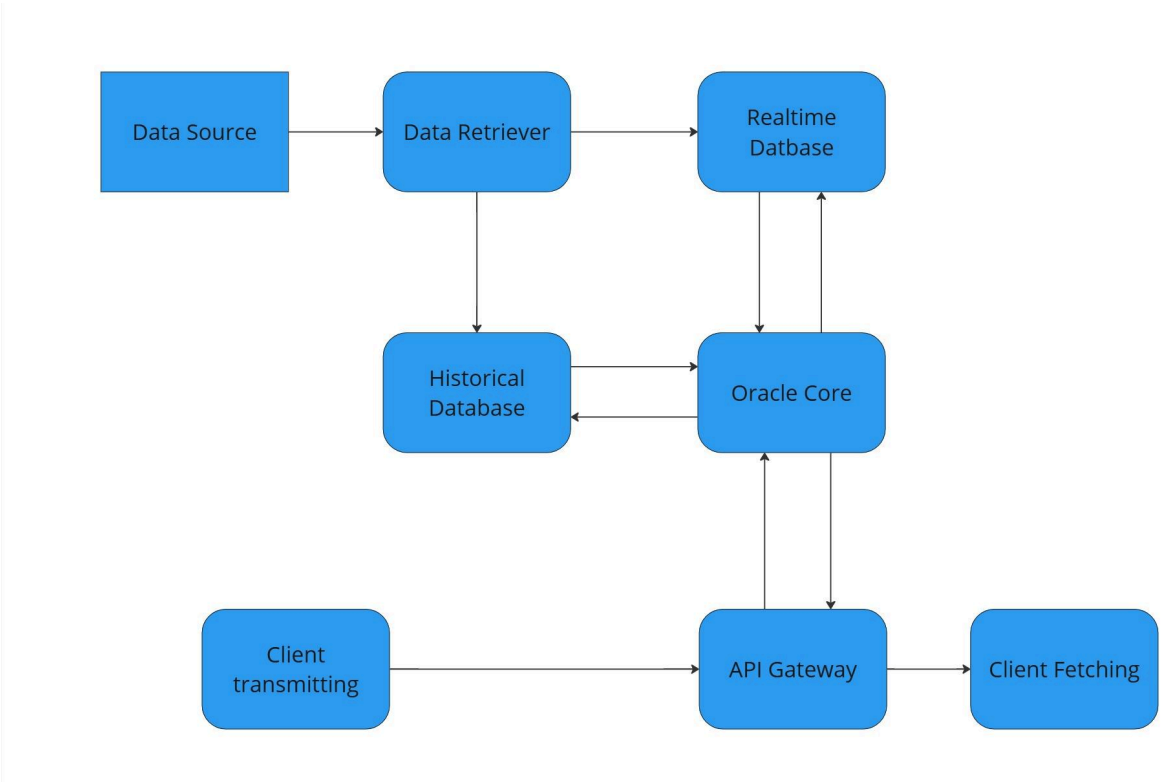
Authentication:
The request must include an `Authorization` header with a valid token:
`Authorization: Bearer <API_KEY_OR_OAUTH_TOKEN>`

The oracle will store the transmitted data in both the **real-time database** and **historical database**.

2. Custom Client Data Transmission Endpoint

The **Custom Client Data Transmission Endpoint** will allow clients with unique needs to transmit cryptocurrency prices. These endpoints will be flexible and customizable, with specific features and requirements tailored to each client's use case. Authentication will also be required for these endpoints.

Data Flow Diagram



Ensuring Robustness and Data Integrity

Data Accuracy and Consistency

Data Normalization: Ensure that data from different sources follows the same format

Consistency Strategy: Cross-check prices from multiple sources and use the most reliable or aggregated value.

Latency Considerations: Implement caching and set TTL (Time To Live) for prices to avoid outdated data and reduce load on APIs.

Error Handling and Resilience

Backup Sources: Implement a failover mechanism where the system queries alternative data sources if a source fails.

Redis Caching: If a primary source fails, fallback to previously cached prices from **Redis** or a **local cache** to minimize service disruption.

Circuit Breaker Pattern: If a data source is failing continuously, the system will temporarily disable requests to that source and route to healthy sources, preventing cascading failures.

Retry Strategies: Manage retry attempts in a controlled manner.

Logging: Use structured logging with **Prometheus** for logging data flow and errors, providing insight into system performance and errors.

Monitoring & Alerts: Use monitoring tools like **Prometheus** and **Grafana** to track the health of data sources. Set up alerting with **Alertmanager** or **PagerDuty** to notify the team in case of source outages or failures.