# Data Platform Implementation Plan

Zhifeng Zhang

Oct. 13, 2024

## Contents



# Implementation Plan

We break the implementation into 6 steps: 1. Getting producer working, this includes getting both historical data and realtime data feed from cryptocompare, publish data to redpanda. 2. Getting consumer (for fix topics) working, this includes injest data into timescaledb. 3. Getting coordinator and consumer with dynamic topics working. 4. Getting consume with dynamic topics working. 5. Datastore implementation and API development. 6. Developing and deploying the DataStore API service.

## Step 1: Getting Producer Working

**Objective:** Fetch both historical and real-time data from **CryptoCompare** and publish it to **Redpanda**.

### Components Involved

- **Data Fetcher:** Retrieves data from CryptoCompare.

- **Producer Actor:** Publishes data to Redpanda topics.
- **State Machine:** Manages the producer's states using XState.

**Tasks**

**Historical Data Fetching**

- **Identify API Endpoints:** Understand CryptoCompare's API for historical data (e.g., OHLCV data).
- **Implement Fetch Logic:**
  - Use **Sequelize** to interface with TimescaleDB if we plan to store raw/historical data here before processing.
  - Alternatively, handle historical data within the producer and publish directly to Redpanda.
- **Handle Rate Limits and Pagination:**
  - Ensure we manage API rate limits.
  - Implement pagination to fetch large datasets without missing data.
- **Data Transformation:**
  - Convert fetched data into the required format before publishing.
- **Error Handling:**
  - Implement retries and backoff strategies for failed requests.
  - Log errors for monitoring and debugging.

**Real-Time Data Feed**

- **WebSockets or Streaming API:**
  - If CryptoCompare provides a WebSocket or streaming API, use it for real-time data.
  - If not, consider polling at reasonable intervals.
- **Data Processing:**
  - Process incoming tick data to prepare it for aggregation (e.g., appending timestamps).
- **Publishing to Redpanda:**
  - Define separate topics for real-time data (e.g., `tick-data`).
  - Ensure message serialization (e.g., JSON, Avro) aligns with the Consumers' expectations.

**Implementing the Producer Actor with XState  Producer State Machine Roles:** - **Initialize:** Set up connections to CryptoCompare and Redpanda. - **Fetch Historical Data:** Retrieve and publish historical data upon initialization or on-demand. - **Listen to Real-Time Feed:** Continuously ingest real-time data and publish it. - **Handle Dynamic Topic Instructions:** Respond to Coordinator's instructions to create or modify topics. - **Error States:** Manage any errors gracefully and attempt recovery.

**Example Producer State Machine:**

```
// src/stateMachines/ProducerActor.ts
import { createMachine, assign } from 'xstate';
import { ProducerEvent, ProducerContext } from '../types';
import logger from '../utils/logger';

export const ProducerActor = createMachine<ProducerContext, ProducerEvent>({
  id: 'producerActor',
  initial: 'idle',
  context: {
    currentTopic: 'tick-data',
    message: null,
    newTopicName: '',
    errorInfo: null,
  },
  states: {
    idle: {
      on: {
        FETCH_DATA: 'fetchingData',
        CREATE_DYNAMIC_TOPIC: 'creatingTopic',
      },
```

```
  },
  fetchingData: {
    invoke: {
      id: 'fetchHistoricalData',
      src: 'fetchHistoricalDataService',
      onDone: {
        target: 'publishingFixed',
        actions: assign({
          message: (context, event) => event.data,
        }),
      },
      onError: {
        target: 'error',
        actions: assign({
          errorInfo: (context, event) => event.data,
        }),
      },
    },
  },
  publishingFixed: {
    invoke: {
      id: 'publishToFixedTopics',
      src: 'publishToFixedTopicsService',
      onDone: {
        target: 'idle',
        actions: () => logger.info('Published historical data to fixed topics'),
      },
      onError: {
        target: 'error',
        actions: assign({
          errorInfo: (context, event) => event.data,
        }),
      },
    },
  },
  creatingTopic: {
    invoke: {
      id: 'createDynamicTopic',
      src: 'createDynamicTopicService',
      onDone: {
        target: 'idle',
        actions: assign({
          newTopicName: (context, event) => event.data.topicName,
        }),
      },
      onError: {
        target: 'error',
        actions: assign({
          errorInfo: (context, event) => event.data,
        }),
      },
    },
  },
  error: {
    entry: ['logError'],
    on: {
```

```
        RETRY: 'idle',
      },
    },
  },
},
{
  actions: {
    logError: (context, event) => {
      logger.error(`Producer Actor Error: ${context.errorInfo}`);
    },
  },
  services: {
    fetchHistoricalDataService: async (context, event) => {
      // Implement data fetching logic here
      // Example: Call CryptoCompare API for historical data
      return await fetchHistoricalData();
    },
    publishToFixedTopicsService: async (context, event) => {
      // Implement publishing logic here
      // Example: Publish historical data to fixed topics
      await publishToRedpanda(context.currentTopic, context.message);
    },
    createDynamicTopicService: async (context, event) => {
      // Implement topic creation logic here
      // Example: Instruct Coordinator or use Redpanda's Admin API
      await createNewTopic(event.topicName);
      return { topicName: event.topicName };
    },
  }
});
```

**Considerations:** - **Concurrency:** Ensure that historical data fetch and real-time ingestion don't block each other. - **Scalability:** Design the Producer to handle multiple data sources if needed. - **Monitoring:** Implement metrics and logging to monitor producer performance and health.

---

## Step 2: Get Fixed Topics Consumer Working

**Objective:** Consume data from fixed topics and ingest it into **TimescaleDB** using **Sequelize**.

### Components Involved

- **Consumer Fixed Actor:** Subscribes to fixed topics and processes incoming data.
- **Sequelize ORM:** Interfaces with TimescaleDB for data persistence.
- **TimescaleDB:** Stores aggregated OHLCV data.
- **State Machine:** Manages the consumer's states using XState.

### Tasks

#### Define Table Schemas with Sequelize

- **Define Models Corresponding to TimescaleDB Tables.**

  **Example OHLCV Model:**

```
// src/models/Ohlcv.ts
import { Model, DataTypes } from 'sequelize';
import { sequelize } from '../database';
```

```
class Ohlcv extends Model {
  public id!: number;
  public bucket!: Date;
  public symbol!: string;
  public open!: number;
  public high!: number;
  public low!: number;
  public close!: number;
  public volume!: number;
}

Ohlcv.init({
  id: {
    type: DataTypes.INTEGER.UNSIGNED,
    autoIncrement: true,
    primaryKey: true,
  },
  bucket: {
    type: DataTypes.DATE,
    allowNull: false,
  },
  symbol: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  open: {
    type: DataTypes.FLOAT,
    allowNull: false,
  },
  high: {
    type: DataTypes.FLOAT,
    allowNull: false,
  },
  low: {
    type: DataTypes.FLOAT,
    allowNull: false,
  },
  close: {
    type: DataTypes.FLOAT,
    allowNull: false,
  },
  volume: {
    type: DataTypes.FLOAT,
    allowNull: false,
  },
}, {
  sequelize,
  tableName: 'ohlcv',
  timestamps: false,
});

export default Ohlcv;
```

- **Ensure Proper Indexing and Time-Series Optimization:**
  - Use **TimescaleDB's hypertable** features for better performance.

- Properly index on timestamp columns.

**Implement Consumer Fixed Actor with XState** **Consumer Fixed State Machine Roles:** - **Initialize:** Set up connections to Redpanda and TimescaleDB. - **Consume Messages:** Listen to fixed topics and process incoming tick data. - **Aggregate Data:** Convert tick data to OHLCV format. - **Persist Data:** Save aggregated OHLCV data to TimescaleDB using Sequelize. - **Error Handling:** Manage any errors during consumption or persistence.

**Example Consumer Fixed State Machine:**

```
// src/stateMachines/ConsumerFixedActor.ts
import { createMachine, assign } from 'xstate';
import { ConsumerFixedEvent, ConsumerFixedContext } from '../types';
import logger from '../utils/logger';
import { Ohlcv } from '../models/Ohlcv';

export const ConsumerFixedActor = createMachine<ConsumerFixedContext, ConsumerFixedEvent>({
  id: 'consumerFixedActor',
  initial: 'idle',
  context: {
    currentTopic: 'tick-data',
    message: null,
    dbRecord: null,
    errorInfo: null,
  },
  states: {
    idle: {
      on: {
        MESSAGE_RECEIVED: 'processingMessage',
      },
    },
    processingMessage: {
      invoke: {
        id: 'processMessage',
        src: 'processMessageService',
        onDone: {
          target: 'writingToDB',
          actions: assign({
            dbRecord: (context, event) => event.data,
          }),
        },
        onError: {
          target: 'error',
          actions: assign({
            errorInfo: (context, event) => event.data,
          }),
        },
      },
    },
    writingToDB: {
      invoke: {
        id: 'writeToDB',
        src: 'writeToDBService',
        onDone: {
          target: 'idle',
          actions: () => logger.info('OHLCV data written to TimescaleDB successfully'),
        },
        onError: {
```

```
          target: 'error',
          actions: assign({
            errorInfo: (context, event) => event.data,
          }),
        },
      },
    },
    error: {
      entry: ['logError'],
      on: {
        RESET: 'idle',
      },
    },
  },
},
{
  actions: {
    logError: (context, event) => {
      logger.error(`Consumer Fixed Actor Error: ${context.errorInfo}`);
    },
  },
  services: {
    processMessageService: async (context, event) => {
      // Implement OHLCV aggregation logic
      const tick = event.message;
      // Example aggregation: Convert tick to OHLCV
      // In reality, we might need to aggregate ticks over the 1-minute interval
      const ohlcvData = aggregateTickToOhlcv(tick);
      return ohlcvData;
    },
    writeToDBService: async (context, event) => {
      // Persist OHLCV data to TimescaleDB using Sequelize
      const ohlcv = context.dbRecord;
      await Ohlcv.create(ohlcv);
    },
  }
});
```

**Considerations:** - **Aggregation Mechanism:** - Implement a robust aggregation mechanism to accumulate tick data into OHLCV. - Consider using libraries or built-in frameworks for time-series data processing.

- **Concurrency and Batching:**
  - Handle high-throughput tick data by batching database writes to optimize performance.
- **Sequelize Integration:**
  - Ensure Sequelize is properly configured to connect to TimescaleDB.
  - Utilize transactions if multiple operations need atomicity.
- **Error Handling:**
  - Implement retry logic for transient errors.
  - Alert mechanisms for persistent failures to facilitate prompt resolution.

**Example Aggregation Logic** If we're handling aggregation within the Consumer Fixed Actor, here's a simplified example:

```
// src/utils/aggregation.ts

interface Tick {
  timestamp: Date;
```

```
  price: number;
  volume: number;
}

interface OhlcvData {
  bucket: Date;
  symbol: string;
  open: number;
  high: number;
  low: number;
  close: number;
  volume: number;
}

function aggregateTickToOhlcv(tick: Tick): OhlcvData {
  // Determine the 1-minute bucket
  const bucket = new Date(tick.timestamp);
  bucket.setSeconds(0, 0);

  // For simplification, assume this is the only tick in the bucket
  // In practice, we'd maintain state to accumulate ticks per bucket
  return {
    bucket,
    symbol: 'BTCUSD', // Replace with actual symbol from context
    open: tick.price,
    high: tick.price,
    low: tick.price,
    close: tick.price,
    volume: tick.volume,
  };
}
```

**Best Practice:** Use **TimescaleDB's continuous aggregates** as previously mentioned to offload aggregation logic from the application, enhancing scalability and reliability.

---

## Step 3: Get Coordinator and Consumer with Dynamic Topics Working

**Objective:** Implement dynamic topic creation and consumption to cater to application-specific data requests.

### Components Involved

- **Coordinator Actor:** Manages creation of dynamic topics based on application needs.
- **Producer Actor:** Publishes data to newly created dynamic topics.
- **Consumer Dynamic Topics Actor:** Subscribes to dynamic topics and serves data to applications.
- **State Machines:** Manage each actor's states using XState.

### Tasks

#### Implementing Dynamic Topic Creation in Coordinator Actor

- **Handle Requests:** Listen for requests to create new dynamic topics (e.g., `CREATE_DYNAMIC_TOPIC` events).
- **Communicate with Producer:** Instruct the Producer Actor to start publishing to the new topic.
- **Spawn Consumers:** Initialize Consumer Dynamic Topics Actors to subscribe to the new topics.
- **Error Handling:** Manage failures in topic creation or consumer spawning.

**Example Coordinator State Machine:**

```
// src/stateMachines/CoordinatorActor.ts
import { createMachine, assign } from 'xstate';
import { CoordinatorEvent, CoordinatorContext } from '../types';
import logger from '../utils/logger';

export const CoordinatorActor = createMachine<CoordinatorContext, CoordinatorEvent>({
  id: 'coordinatorActor',
  initial: 'active',
  context: {
    topics: [],
    pendingTopic: null,
    errorInfo: null,
  },
  states: {
    active: {
      on: {
        REQUEST_CREATE_TOPIC: 'processingCreateTopic',
      },
    },
    processingCreateTopic: {
      invoke: {
        id: 'createTopicService',
        src: 'createTopicService',
        onDone: {
          target: 'notifyingActors',
          actions: assign({
            pendingTopic: (context, event) => event.data,
            topics: (context, event) => [...context.topics, event.data],
          }),
        },
        onError: {
          target: 'error',
          actions: assign({
            errorInfo: (context, event) => event.data,
          }),
        },
      },
    },
    notifyingActors: {
      invoke: {
        id: 'notifyProducersAndConsumers',
        src: 'notifyActorsService',
        onDone: {
          target: 'active',
          actions: () => logger.info('Coordinator: Notified Producer and Consumers'),
        },
        onError: {
          target: 'error',
          actions: assign({
            errorInfo: (context, event) => event.data,
          }),
        },
      },
    },
    error: {
      entry: ['logError'],
```

```
      on: {
        RETRY: 'active',
      },
    },
  },
},
{
  actions: {
    logError: (context, event) => {
      logger.error(`Coordinator Actor Error: ${context.errorInfo}`);
    },
  },
  services: {
    createTopicService: async (context, event) => {
      // Implement topic creation logic using Redpanda's Admin API
      const { topicName, partitions } = event;
      await createDynamicTopic(topicName, partitions);
      return { topicName, partitions };
    },
    notifyActorsService: async (context, event) => {
      const { topicName } = context.pendingTopic;
      // Send messages or trigger events to Producer and Consumer Dynamic Actors
      // Example: Send event to Producer to start publishing to the new topic
      producerActor.send({ type: 'CREATE_DYNAMIC_TOPIC', topicName, partitions: 1 });

      // Spawn or notify Consumer Dynamic Actor to subscribe to the new topic
      // This could involve interfacing with a higher-level manager or actor system
      const consumerDynamic = spawn(ConsumerDynamicActor.withContext({
        // Provide necessary context
        topicName,
      }));

      return consumerDynamic;
    },
  }
});
```

**Considerations:** - **Concurrency Control:** Ensure that multiple topic creation requests are handled appropriately without conflicts. - **Resource Management:** Monitor the number of dynamic topics and associated consumers to avoid resource exhaustion. - **Naming Conventions:** Establish clear naming conventions for dynamic topics to prevent collisions and ensure consistency.

**Implementing Consumer Dynamic Topics Actor with XState Consumer Dynamic State Machine Roles:** - **Initialize:** Set up subscriptions to specified dynamic topics. - **Consume Messages:** Listen to dynamic topics and process incoming data. - **Serve Data:** Interface with applications to provide consumed data. - **Error Handling:** Manage any errors during consumption or data processing.

**Example Consumer Dynamic State Machine:**

```
// src/stateMachines/ConsumerDynamicActor.ts
import { createMachine, assign } from 'xstate';
import { ConsumerDynamicEvent, ConsumerDynamicContext } from '../types';
import logger from '../utils/logger';
import { subscribeToTopic } from '../utils/kafka'; // Mock function

export const ConsumerDynamicActor = createMachine<ConsumerDynamicContext,
 ↪  ConsumerDynamicEvent>({
```

```
id: 'consumerDynamicActor',
initial: 'idle',
context: {
  dynamicTopics: [],
  currentSubscription: null,
  message: null,
  errorInfo: null,
},
states: {
  idle: {
    on: {
      SUBSCRIBE_TO_TOPIC: 'subscribing',
    },
  },
  subscribing: {
    invoke: {
      id: 'subscribeToTopicService',
      src: 'subscribeToTopicService',
      onDone: {
        target: 'consuming',
        actions: assign({
          dynamicTopics: (context, event) => [...context.dynamicTopics, event.data.topicName],
        }),
      },
      onError: {
        target: 'error',
        actions: assign({
          errorInfo: (context, event) => event.data,
        }),
      },
    },
  },
  consuming: {
    invoke: {
      id: 'consumeMessages',
      src: 'consumeMessagesService',
      onDone: {
        target: 'idle',
        actions: () => logger.info('Consumer Dynamic: Finished consuming messages'),
      },
      onError: {
        target: 'error',
        actions: assign({
          errorInfo: (context, event) => event.data,
        }),
      },
    },
    // we can handle incoming messages here or trigger further actions/events
  },
  error: {
    entry: ['logError'],
    on: {
      RETRY: 'idle',
    },
  },
},
```

```
},
{
  actions: {
    logError: (context, event) => {
      logger.error(`Consumer Dynamic Actor Error: ${context.errorInfo}`);
    },
  },
  services: {
    subscribeToTopicService: async (context, event) => {
      const { topicName } = event;
      await subscribeToTopic(topicName); // Implement subscription logic
      return { topicName };
    },
    consumeMessagesService: async (context, event) => {
      // Implement message consumption logic
      // Example: Listen to Kafka/Redpanda for incoming messages on dynamicTopics
      await consumeFromTopics(context.dynamicTopics, (message) => {
        logger.info(`Dynamic Consumer: Received message on ${message.topic}`);
        // Process message or trigger actions
      });
    },
  }
});
```

**Considerations:** - **Scalability:** Ensure that each dynamic consumer is lightweight and capable of handling high-throughput data if necessary. - **Subscription Management:** Implement mechanisms to unsubscribe or terminate consumers gracefully when they are no longer needed. - **Security:** Secure dynamic subscriptions to prevent unauthorized data access.

#### Producer and Coordinator Communication

- **Event-Driven Communication:** Utilize XState's event mechanism to have the Coordinator and Producer communicate seamlessly.
- **Supervisor Pattern:** Consider using a higher-level supervisor actor or manager to oversee interactions and recover from failures.

#### Handling Data Consistency and Synchronization

- **Idempotency:** Ensure that creating dynamic topics and subscribing consumers are idempotent to prevent duplicate processing.
- **Ordering Guarantees:** Kafka/Redpanda inherently provide ordering within partitions, but ensure the Consumers respect this in processing logic.

---

## Step 4: Datastore Implementation

**Objective:** Develop an API service (**Datastore**) that allows applications to request and consume OHLCV data without needing to interact directly with the underlying data platform components.

#### Components Involved

- **Datastore (API Service):** Interfaces with **TimescaleDB** and **Redis** to serve data to applications.
- **Sequelize ORM:** Facilitates database interactions with TimescaleDB.
- **Redis Client:** Interfaces with Redis for caching.
- **State Machine:** Manages the Datastore's states using XState.
- **Integration with Existing Actors:** Coordinates with Producer and Coordinator for data fetching.

**Tasks**

**Design the API Endpoints**

1. **Data Request Endpoint:**
   - **GET /ohlcv**
   - **Parameters:**
     - `symbol` (e.g., BTCUSD)
     - `startDate` (ISO 8601 format)
     - `endDate` (optional, defaults to current time)
     - `interval` (e.g., 1m, 5m)
2. **Health Check Endpoint:**
   - **GET /health**
   - Returns the status of the Datastore service.

**Implementing the Datastore State Machine with XState  Datastore State Machine Roles:** - **Initialize:** Set up connections to TimescaleDB, Redis, and communicate with other actors. - **Handle API Requests:** Process incoming data requests from applications. - **Fetch Data:** - **From Cache (Redis):** Attempt to retrieve data from Redis. - **From Database (TimescaleDB):** If not in cache, query TimescaleDB. - **Initiate Data Fetch:** If data missing, instruct the Coordinator to fetch it. - **Return Data:** Serve the data back to the requesting application. - **Error Handling:** Manage any errors during the process.

**Example Datastore State Machine:**

```
// src/stateMachines/DatastoreActor.ts
import { createMachine, assign, spawn } from 'xstate';
import { DatastoreEvent, DatastoreContext } from '../types';
import logger from '../utils/logger';
import { CoordinatorActor } from './CoordinatorActor';
import { ProducerActor } from './ProducerActor';
import { Ohlcv } from '../models/Ohlcv';
import { redisClient } from '../utils/redis'; // Assume a Redis client is set up

export const DatastoreActor = createMachine<DatastoreContext, DatastoreEvent>({
  id: 'datastoreActor',
  initial: 'idle',
  context: {
    requestedTopic: '',
    startDate: null,
    endDate: null,
    interval: '1m',
    data: null,
    errorInfo: null,
    coordinatorRef: undefined,
    producerRef: undefined,
  },
  states: {
    idle: {
      on: {
        REQUEST_DATA: 'processingRequest',
      },
    },
    processingRequest: {
      entry: assign({
        requestedTopic: (context, event) => event.symbol,
        startDate: (context, event) => new Date(event.startDate),
        endDate: (context, event) => event.endDate ? new Date(event.endDate) : new Date(),
        interval: (context, event) => event.interval || '1m',
      }),
```

```
    invoke: {
      id: 'fetchFromCache',
      src: 'fetchFromCacheService',
      onDone: {
        target: 'dataServed',
        cond: (context, event) => event.data !== null,
        actions: assign({
          data: (context, event) => event.data,
        }),
      },
      onError: {
        target: 'fetchFromDB',
        actions: assign({
          errorInfo: (context, event) => event.data,
        }),
      },
    },
  },
  fetchFromDB: {
    invoke: {
      id: 'queryDatabase',
      src: 'queryDatabaseService',
      onDone: [
        {
          target: 'dataServed',
          cond: (context, event) => event.data.length > 0,
          actions: assign({
            data: (context, event) => event.data,
          }),
        },
        {
          target: 'requestDataFetch',
          cond: (context, event) => event.data.length === 0,
        },
      ],
      onError: {
        target: 'error',
        actions: assign({
          errorInfo: (context, event) => event.data,
        }),
      },
    },
  },
  requestDataFetch: {
    invoke: {
      id: 'initiateDataFetch',
      src: 'initiateDataFetchService',
      onDone: 'waitingForData',
      onError: {
        target: 'error',
        actions: assign({
          errorInfo: (context, event) => event.data,
        }),
      },
    },
  },
```

```
    waitingForData: {
      // Implement a mechanism to wait for data to be available
      // This could be event-driven or polling-based
      invoke: {
        id: 'waitForData',
        src: 'waitForDataService',
        onDone: {
          target: 'dataServed',
          actions: assign({
            data: (context, event) => event.data,
          }),
        },
        onError: {
          target: 'error',
          actions: assign({
            errorInfo: (context, event) => event.data,
          }),
        },
      },
    },
    dataServed: {
      type: 'final',
      entry: 'serveData',
    },
    error: {
      entry: ['logError'],
      on: {
        RESET: 'idle',
      },
    },
  },
},
{
  actions: {
    serveData: (context, event) => {
      // Implement logic to send data back to the requesting app
      logger.info(`Datastore: Serving data for ${context.requestedTopic}`);
    },
    logError: (context, event) => {
      logger.error(`Datastore Actor Error: ${context.errorInfo}`);
    },
  },
  services: {
    fetchFromCacheService: async (context, event) => {
      const { symbol, interval, startDate, endDate } = context;
      const key = `ohlcv:${symbol}:${interval}:${startDate.toISOString()}`;
      const cachedData = await redisClient.get(key);
      return cachedData ? JSON.parse(cachedData) : null;
    },
    queryDatabaseService: async (context, event) => {
      const { symbol, interval, startDate, endDate } = context;
      const records = await Ohlcv.findAll({
        where: {
          symbol,
          bucket: {
            [Op.gte]: startDate,
```

```
            [Op.lte]: endDate,
          },
        },
        order: [['bucket', 'ASC']],
      });
      return records.map(record => record.toJSON());
    },
    initiateDataFetchService: (context, event) => {
      // Instruct Coordinator to fetch data
      context.coordinatorRef?.send({
        type: 'REQUEST_CREATE_TOPIC',
        topicName: `${context.symbol}-${context.interval}`,
        partitions: 1,
      });
      return Promise.resolve();
    },
    waitForDataService: async (context, event) => {
      // Implement a mechanism to wait for data to be fetched and available
      // This could utilize event listeners or a callback system
      return new Promise((resolve, reject) => {
        const onData = (newData) => {
          // When data is available, resolve the promise
          if (newData.symbol === context.requestedTopic) {
            resolve(newData);
          }
        };

        // Subscribe to events or set up a listener
        // Example: EventEmitter or XState actor events
        datastoreActor.on('DATA_FETCHED', onData);

        // Optionally, set a timeout to reject if data isn't fetched in time
        setTimeout(() => {
          reject('Data fetch timed out');
        }, 60000); // 1 minute timeout
      });
    },
  }
});
```

**Considerations:** - **Asynchronous Operations:** Ensure that long-running operations like data fetching and waiting are handled without blocking the event loop. - **Data Consistency:** Validate and sanitize data before serving it to applications. - **Caching Strategy:** Determine TTL (Time-To-Live) for cached data in Redis to balance freshness and performance. - **Scalability:** Design the Datastore to handle multiple concurrent requests efficiently. - **Security:** Implement authentication and authorization to protect data access.

### Integrate Redis for Caching

- **Set Up Redis:**
  - Deploy Redis as a separate service or use a managed Redis service.
  - Ensure secure connections (e.g., use TLS, secure access credentials).
- **Implement Caching Logic:**
  - **Read-Through Cache:** Attempt to read from Redis first; if data is missing, fetch from TimescaleDB and populate Redis.
  - **Write-Behind Cache:** Update Redis when new data is written to TimescaleDB.

- **Optimize Cache Keys:**
  - Use descriptive and consistent key patterns (e.g., `ohlcv:{symbol}:{interval}:{bucket}`).
- **Example Usage in Datastore:**

```
// src/utils/cache.ts
import Redis from 'ioredis';

export const redisClient = new Redis({
  host: 'localhost',
  port: 6379,
  password: 'your_redis_password', // if applicable
});

// Function to set cache
export const setCache = async (key: string, value: any, ttl: number = 3600) => {
  await redisClient.set(key, JSON.stringify(value), 'EX', ttl);
};

// Function to get cache
export const getCache = async (key: string) => {
  const data = await redisClient.get(key);
  return data ? JSON.parse(data) : null;
};
```

### Handling Historical Data Requests

- **User-Specified Start Date:**
  - Allow users to specify a start date for historical data.
  - Query TimescaleDB starting from the user-specified date up to the latest available data.
- **Ensuring Seamless Integration with Real-Time Data:**
  - **Fetch Last Timestamp:**
    - Query TimescaleDB to find the latest timestamp for the requested symbol and interval.
  - **Align Real-Time Data:**
    - Start real-time data collection from the timestamp following the last historical data point to avoid overlaps.
- **Example Implementation:**

```
async function handleOhlcvRequest(symbol: string, startDate: Date): Promise<OhlcvData[]> {
  // Step 1: Check Redis
  const cacheKey = `ohlcv:${symbol}:1m:${startDate.toISOString()}`;
  let data = await getCache(cacheKey);

  if (data) {
    return data;
  }

  // Step 2: Query TimescaleDB
  const lastRecord = await Ohlcv.findOne({
    where: { symbol },
    order: [['bucket', 'DESC']],
  });

  const queryStartDate = startDate;
  const queryEndDate = lastRecord ? new Date(lastRecord.bucket.getTime() + 60000) : new
   ↪  Date();
```

```
  const historicalData = await Ohlcv.findAll({
    where: {
      symbol,
      bucket: {
        [Op.gte]: queryStartDate,
        [Op.lte]: queryEndDate,
      },
    },
    order: [['bucket', 'ASC']],
  });

  if (historicalData.length === 0) {
    // Step 3: Initiate Data Fetch
    await coordinatorActor.send({
      type: 'REQUEST_CREATE_TOPIC',
      topicName: `${symbol}-1m`,
      partitions: 1,
    });

    // Step 4: Wait for Data to be Available
    data = await waitForData(symbol, queryEndDate);
    await setCache(cacheKey, data);
    return data;
  }

  // Step 5: Fetch Real-Time Data Starting Point
  const realTimeStart = new Date(queryEndDate.getTime() + 60000); // Next minute
  // Fetch real-time data from Datastore
  const realTimeData = await fetchRealTimeData(symbol, realTimeStart);

  // Step 6: Merge and Cache Data
  const mergedData = [...historicalData, ...realTimeData];
  await setCache(cacheKey, mergedData);
  return mergedData;
}
```

**Considerations:** - **Data Freshness:** Ensure that cached data in Redis is updated promptly after ingestion to serve the latest data to applications. - **Thread Safety:** If multiple requests trigger data fetch simultaneously, implement mechanisms to prevent duplicate requests. - **Timeouts and Fallbacks:** Implement reasonable timeouts when waiting for data fetches and provide fallback responses or partial data if necessary.

---

## Step 5: Datastore Integration and API Development

**Objective:** Develop an API service that applications can interact with to request and consume OHLCV data seamlessly, without needing direct knowledge of the underlying data platform.

### Components Involved

- **Datastore API Service:** Handles HTTP/HTTPS requests from applications.
- **Datastore Actor:** Manages API request processing using XState.
- **Sequelize ORM:** Interfaces with TimescaleDB.
- **Redis Client:** Interfaces with Redis for caching.
- **Consumer Dynamic Topics Actors:** Provide real-time data streams to applications.
- **State Machines:** Manage Datastore's states using XState.

**Tasks**

**Develop the API Service**

1. **Choose a Framework:** Utilize frameworks like **Express.js**, **Fastify**, or **Koa** for building the API service.
2. **Define API Endpoints:**
   - **GET /ohlcv**
     - **Description:** Retrieves OHLCV data for a specified symbol and timeframe.
     - **Parameters:**
       * `symbol` (e.g., BTCUSD)
       * `startDate` (ISO 8601 format)
       * `endDate` (optional, defaults to current time)
       * `interval` (e.g., 1m, 5m)
   - **GET /health**
     - **Description:** Checks the health status of the Datastore service.
3. **Implement Rate Limiting and Security:**
   - **Rate Limiting:** Prevent abuse and ensure fair usage.
   - **Authentication and Authorization:** Secure API endpoints to restrict access to authorized applications.
   - **Input Validation:** Validate and sanitize all incoming parameters to prevent injection attacks and ensure data integrity.

**Implement the Datastore Actor with Comprehensive State Management  Datastore State Machine Roles:** - **Receive Request:** Await incoming data requests from applications. - **Check Cache:** Attempt to retrieve data from Redis. - **Query Database:** If not in cache, query TimescaleDB. - **Initiate Data Fetch:** If data is missing, signal the Coordinator to fetch it. - **Wait for Data:** Await completion of data fetching. - **Serve Data:** Return the requested data to the application via the API response. - **Handle Errors:** Manage any errors encountered during the process.

**Example Datastore State Machine:**

```
// src/stateMachines/DatastoreActor.ts
import { createMachine, assign, send, spawn } from 'xstate';
import { DatastoreEvent, DatastoreContext } from '../types';
import logger from '../utils/logger';
import { CoordinatorActor, ProducerActor, ConsumerDynamicActor } from './actors';

export const DatastoreActor = createMachine<DatastoreContext, DatastoreEvent>({
  id: 'datastoreActor',
  initial: 'idle',
  context: {
    requestedTopic: '',
    startDate: null,
    endDate: null,
    interval: '1m',
    data: null,
    errorInfo: null,
    coordinatorRef: spawn(CoordinatorActor),
    producerRef: spawn(ProducerActor),
    consumerDynamicRefs: [],
  },
  states: {
    idle: {
      on: {
        REQUEST_DATA: 'processingRequest',
      },
    },
    processingRequest: {
```

```
    entry: assign({
      requestedTopic: (context, event) => event.symbol,
      startDate: (context, event) => new Date(event.startDate),
      endDate: (context, event) => event.endDate ? new Date(event.endDate) : new Date(),
      interval: (context, event) => event.interval || '1m',
    }),
    invoke: {
      id: 'fetchFromCache',
      src: 'fetchFromCacheService',
      onDone: [
        {
          target: 'dataServed',
          cond: (context, event) => event.data !== null,
          actions: assign({
            data: (context, event) => event.data,
          }),
        },
      ],
      onError: 'fetchFromDB',
    },
  },
  fetchFromDB: {
    invoke: {
      id: 'queryDatabase',
      src: 'queryDatabaseService',
      onDone: [
        {
          target: 'dataServed',
          cond: (context, event) => event.data.length > 0,
          actions: assign({
            data: (context, event) => event.data,
          }),
        },
        {
          target: 'requestDataFetch',
          cond: (context, event) => event.data.length === 0,
        },
      ],
      onError: {
        target: 'error',
        actions: assign({
          errorInfo: (context, event) => event.data,
        }),
      },
    },
  },
  requestDataFetch: {
    invoke: {
      id: 'initiateDataFetch',
      src: 'initiateDataFetchService',
      onDone: 'waitingForData',
      onError: 'error',
    },
  },
  waitingForData: {
    invoke: {
```

```
        id: 'waitForData',
        src: 'waitForDataService',
        onDone: {
          target: 'dataServed',
          actions: assign({
            data: (context, event) => event.data,
          }),
        },
        onError: 'error',
      },
    },
    dataServed: {
      type: 'final',
      entry: 'serveData',
    },
    error: {
      entry: ['logError'],
      on: {
        RESET: 'idle',
      },
    },
  },
},
{
  actions: {
    serveData: (context, event) => {
      // Implement logic to send data back via API response
      logger.info(`Datastore: Serving data for ${context.requestedTopic}`);
      // Example: Emit an event or interface with the API handler
    },
    logError: (context, event) => {
      logger.error(`Datastore Actor Error: ${context.errorInfo}`);
    },
  },
  services: {
    fetchFromCacheService: async (context, event) => {
      const { symbol, interval, startDate } = context;
      const cacheKey = `ohlcv:${symbol}:${interval}:${startDate.toISOString()}`;
      const cachedData = await redisClient.get(cacheKey);
      if (cachedData) {
        return JSON.parse(cachedData);
      }
      throw new Error('Cache miss');
    },
    queryDatabaseService: async (context, event) => {
      const { symbol, interval, startDate, endDate } = context;
      const records = await Ohlcv.findAll({
        where: {
          symbol,
          bucket: {
            [Op.gte]: startDate,
            [Op.lte]: endDate,
          },
        },
        order: [['bucket', 'ASC']],
      });
```

```
      return records.map(record => record.toJSON());
    },
    initiateDataFetchService: (context, event) => {
      const { symbol, interval } = context;
      // Instruct Coordinator to create a new dynamic topic
      context.coordinatorRef.send({
        type: 'REQUEST_CREATE_TOPIC',
        topicName: `${symbol}-${interval}`,
        partitions: 1,
      });
      return Promise.resolve();
    },
    waitForDataService: async (context, event) => {
      const { symbol, interval } = context;
      // Implement a mechanism to wait for data to be published to the new dynamic topic
      // This could involve setting up a listener or polling for new data

      // Example using EventEmitter or callbacks (pseudo-code)
      return new Promise((resolve, reject) => {
        const onData = (data) => {
          if (data.symbol === symbol && data.interval === interval) {
            resolve(data);
          }
        };

        // Subscribe to a data fetched event
        dataFetcher.on('dataFetched', onData);

        // Optionally, set a timeout
        setTimeout(() => {
          reject('Data fetch timed out');
        }, 60000); // 1 minute timeout
      });
    },
  }
});
```

**Considerations:** - **Asynchronous Handling:** Ensure that waiting for data fetches does not block the main event loop. - **Event Coordination:** Utilize EventEmitter or similar patterns to handle inter-component communication. - **Timeouts and Retries:** Implement reasonable timeouts and retry mechanisms to handle failed data fetches.

**Implement API Endpoints with Integration to the Datastore Actor  Example with Express.js:**

```
// src/api/server.ts
import express from 'express';
import { interpret } from 'xstate';
import { DatastoreActor } from '../stateMachines/DatastoreActor';
import logger from '../utils/logger';

const app = express();
app.use(express.json());

const datastoreService = interpret(DatastoreActor)
  .onTransition(state => {
    if (state.changed) {
      logger.info(`Datastore Actor transitioned to state: ${state.value}`);
    }
```

```
  })
  .start();

app.get('/ohlcv', async (req, res) => {
  const { symbol, startDate, endDate, interval } = req.query;

  if (!symbol || !startDate) {
    return res.status(400).json({ error: 'symbol and startDate are required' });
  }

  try {
    // Send REQUEST_DATA event to Datastore Actor
    datastoreService.send({
      type: 'REQUEST_DATA',
      symbol: symbol as string,
      startDate: startDate as string,
      endDate: endDate as string,
      interval: interval as string,
    });

    // Implement a mechanism to await dataServed state or use callbacks/events
    datastoreService.onDone(() => {
      // Serve the data
      res.json(datastoreService.state.context.data);
      datastoreService.offDone();
    });

  } catch (error) {
    logger.error(`API Error: ${error.message}`);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});

app.get('/health', (req, res) => {
  res.json({ status: 'OK' });
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  logger.info(`Datastore API service running on port ${PORT}`);
});
```

**Considerations:** - **Concurrency:** Handle multiple simultaneous API requests efficiently. - **Response Time:** Optimize the Datastore Actor's state machine to minimize API response latency. - **Security:** Implement authentication (e.g., JWT) and authorization checks to secure the API.

---

## Additional Technical Considerations

### Real-Time Data Aggregation and Synchronization

To handle real-time data aggregation and synchronize it with historical data:

- **Continuous Aggregates:** Utilize TimescaleDB's continuous aggregates to automatically compute OHLCV data from incoming tick data.

```sql
-- Create a continuous aggregate for 1-minute OHLCV
CREATE MATERIALIZED VIEW ohlcv_1m
WITH (timescaledb.continuous) AS
SELECT
  time_bucket('1 minute', timestamp) AS bucket,
  symbol,
  first(price, timestamp) AS open,
  max(price) AS high,
  min(price) AS low,
  last(price, timestamp) AS close,
  sum(volume) AS volume
FROM
  tick_data
GROUP BY
  bucket, symbol;

-- Add a policy to refresh the view periodically
SELECT add_continuous_aggregate_policy('ohlcv_1m',
  start_offset => INTERVAL '2 minutes',
  end_offset => INTERVAL '0 minutes',
  schedule_interval => INTERVAL '1 minute');
```

- **Timestamp Alignment:** Ensure that real-time data starts fetching from the timestamp immediately after the last available historical data to prevent overlaps or gaps.

```typescript
async function getLastTimestamp(symbol: string): Promise<Date | null> {
  const lastRecord = await Ohlcv.findOne({
    where: { symbol },
    order: [['bucket', 'DESC']],
  });
  return lastRecord ? new Date(lastRecord.bucket.getTime() + 60000) : null; // Next minute
}
```

- **Seamless Integration:**
  - When a user requests data, fetch historical data up to the last recorded timestamp.
  - Start streaming real-time data from the next timestamp onwards.
  - Ensure that the data presented to the user is a continuous and gapless timeline.

### Implementing Sequalize with TimescaleDB

**Connection Setup:**

```typescript
// src/database/index.ts
import { Sequelize } from 'sequelize';

export const sequelize = new Sequelize('database', 'username', 'password', {
  host: 'timescaledb_host',
  dialect: 'postgres',
  logging: false, // Disable or enable as needed
  pool: {
    max: 10,
    min: 0,
    acquire: 30000,
    idle: 10000,
  },
});
```

**Model Synchronization:**

Ensure models are synchronized with TimescaleDB's hypertables.

```
// src/models/index.ts
import { sequelize } from '../database';
import Ohlcv from './Ohlcv';

const initializeModels = async () => {
  await sequelize.authenticate();
  await sequelize.sync(); // Consider using migrations for production
  logger.info('Sequelize connected and models synchronized');
};

initializeModels();
```

**Considerations:** - **Migrations:** Use **Sequelize Migrations** for version-controlled schema changes, especially important for production environments. - **Hypertable Creation:** Ensure that the tables are converted to hypertables after model synchronization.

```
// src/utils/timescaledb.ts
import { sequelize } from '../database';
import logger from '../utils/logger';

export const createHypertables = async () => {
  try {
    await sequelize.query(`
      SELECT create_hypertable('ohlcv', 'bucket', if_not_exists => TRUE);
    `);
    logger.info('Hypertables created or already exist');
  } catch (error) {
    logger.error(`Failed to create hypertables: ${error.message}`);
  }
};

createHypertables();
```

**Redis Integration for Caching**

**Connection Setup:**

```
// src/utils/redis.ts
import Redis from 'ioredis';

export const redisClient = new Redis({
  host: 'redis_host',
  port: 6379,
  password: 'your_redis_password', // if applicable
});
```

**Implement Cache Middleware:**

- **Middleware for API Requests:** Check Redis before querying the database.

```
// src/middleware/cacheMiddleware.ts
import { Request, Response, NextFunction } from 'express';
import { redisClient } from '../utils/redis';

export const cacheMiddleware = async (req: Request, res: Response, next: NextFunction) => {
  const { symbol, interval, startDate } = req.query;
```

```
  const key = `ohlcv:${symbol}:${interval}:${startDate}`;

  try {
    const cachedData = await redisClient.get(key as string);
    if (cachedData) {
      return res.json(JSON.parse(cachedData));
    }
    next();
  } catch (error) {
    next(); // On error, proceed without cache
  }
};
```

**Considerations:** - **Cache Consistency:** Ensure that updates in TimescaleDB are reflected in Redis. - **Expiration Policies:** Set appropriate TTLs based on data freshness requirements. - **Cache Invalidation:** Implement mechanisms to invalidate or update cache entries when underlying data changes.

### Real-Time Feed and Tick Aggregation

**a. Real-Time Data Consumption:** - Utilize the **Consumer Fixed Topics Actor** to subscribe to `tick-data` and perform real-time aggregation.

**b. Aggregation Logic:** - Decide between in-memory aggregation or using **TimescaleDB's continuous aggregates**.

**c. Handling Partial Buckets:** - **Finalizing Buckets:** Once the current minute is complete, confirm that all tick data has been aggregated.

**Example Real-Time Aggregation with In-Memory Aggregator:**

```
// src/utils/aggregator.ts
interface Tick {
  timestamp: Date;
  price: number;
  volume: number;
}

interface OhlcvData {
  bucket: Date;
  symbol: string;
  open: number;
  high: number;
  low: number;
  close: number;
  volume: number;
}

class Aggregator {
  private data: Map<string, OhlcvData> = new Map();

  aggregate(tick: Tick, symbol: string): OhlcvData {
    const bucket = new Date(tick.timestamp);
    bucket.setSeconds(0, 0);
    const key = `${symbol}-${bucket.toISOString()}`;

    if (!this.data.has(key)) {
      this.data.set(key, {
        bucket,
        symbol,
```

```
        open: tick.price,
        high: tick.price,
        low: tick.price,
        close: tick.price,
        volume: tick.volume,
      });
    } else {
      const existing = this.data.get(key)!;
      existing.high = Math.max(existing.high, tick.price);
      existing.low = Math.min(existing.low, tick.price);
      existing.close = tick.price;
      existing.volume += tick.volume;
    }

    return this.data.get(key)!;
  }

  finalizeBucket(symbol: string, bucket: Date): OhlcvData | null {
    const key = `${symbol}-${bucket.toISOString()}`;
    if (this.data.has(key)) {
      const data = this.data.get(key)!;
      this.data.delete(key);
      return data;
    }
    return null;
  }
}

export default Aggregator;
```

**Considerations:** - **Partial Data Handling:** Ensure that data is not lost during in-memory aggregation by persisting periodically. - **Synchronization with Database:** When using continuous aggregates, ensure consistency between real-time aggregation and database records.

---

## Step 6: Develop and Deploy the Datastore API Service

**Objective:** Create a robust API layer that interfaces with applications, handles data requests, and communicates with underlying data platform components.

### Components Involved

- **Express.js (or similar framework):** For building RESTful APIs.
- **Datastore Actor:** Manages API logic using XState.
- **Sequelize ORM:** Interfaces with TimescaleDB.
- **Redis Client:** For caching responses.
- **Security Middleware:** For authentication and authorization.
- **Monitoring Tools:** For logging and performance tracking.

### Tasks

**Implementing the API Service  Example API Service with Express.js:**

```
// src/api/server.ts
import express from 'express';
import { interpret } from 'xstate';
import { DatastoreActor } from '../stateMachines/DatastoreActor';
```

```
import { REQUEST_DATA, RESET } from '../types/events';
import logger from '../utils/logger';
import { cacheMiddleware } from '../middleware/cacheMiddleware';

const app = express();
app.use(express.json());

const datastoreService = interpret(DatastoreActor)
  .onTransition(state => {
    if (state.changed) {
      logger.info(`Datastore Actor transitioned to state: ${state.value}`);
    }
  })
  .start();

app.get('/ohlcv', cacheMiddleware, async (req, res) => {
  const { symbol, startDate, endDate, interval } = req.query;

  if (!symbol || !startDate) {
    return res.status(400).json({ error: 'symbol and startDate are required' });
  }

  try {
    // Send REQUEST_DATA event to Datastore Actor
    datastoreService.send({
      type: 'REQUEST_DATA',
      symbol: symbol as string,
      startDate: startDate as string,
      endDate: endDate as string,
      interval: interval as string,
    });

    // Await response (Implement event listener or callback mechanism)
    // Example using a promise or callback (Pseudo-code)
    const data = await new Promise((resolve, reject) => {
      datastoreService.onDone(() => {
        resolve(datastoreService.state.context.data);
      });

      datastoreService.onError(() => {
        reject(datastoreService.state.context.errorInfo);
      });
    });

    // Set cache
    const cacheKey = `ohlcv:${symbol}:${interval}:${startDate}`;
    await setCache(cacheKey, data);

    res.json(data);
  } catch (error) {
    logger.error(`API Error: ${error.message}`);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});

app.get('/health', (req, res) => {
```

```
  res.json({ status: 'OK' });
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  logger.info(`Datastore API service running on port ${PORT}`);
});
```

**Considerations:** - **Asynchronous Handling:** Ensure that API responses wait appropriately for the Datastore Actor's state transitions without causing request timeouts. - **Error Responses:** Provide meaningful error messages and appropriate HTTP status codes. - **Concurrency:** Handle multiple simultaneous requests efficiently, possibly by scaling the API service horizontally. - **Documentation:** Use tools like **Swagger** or **OpenAPI** to document API endpoints for clear integration with applications.

### Implementing Security Measures

- **Authentication:** Use JWT or OAuth2 to secure API endpoints.

```
// src/middleware/authMiddleware.ts
import { Request, Response, NextFunction } from 'express';
import jwt from 'jsonwebtoken';

export const authMiddleware = (req: Request, res: Response, next: NextFunction) => {
  const token = req.headers.authorization?.split(' ')[1];

  if (!token) return res.status(401).json({ error: 'Unauthorized' });

  jwt.verify(token, process.env.JWT_SECRET!, (err, decoded) => {
    if (err) return res.status(403).json({ error: 'Forbidden' });
    // Optionally attach decoded data to request
    (req as any).user = decoded;
    next();
  });
};
```

- **Authorization:** Ensure that only authorized applications can access specific data.
- **Input Validation:** Sanitize and validate all incoming parameters using libraries like **Joi** or **Yup**.

```
// src/middleware/validateRequest.ts
import { Request, Response, NextFunction } from 'express';
import Joi from 'joi';

export const validateOhlcvRequest = (req: Request, res: Response, next: NextFunction) => {
  const schema = Joi.object({
    symbol: Joi.string().required(),
    startDate: Joi.date().iso().required(),
    endDate: Joi.date().iso(),
    interval: Joi.string().valid('1m', '5m', '15m', '30m', '1h').default('1m'),
  });

  const { error } = schema.validate(req.query);

  if (error) {
    return res.status(400).json({ error: error.details[0].message });
  }

  next();
```

```
};
```

**Integration with Routes:**

```
// src/api/server.ts
import { authMiddleware } from '../middleware/authMiddleware';
import { validateOhlcvRequest } from '../middleware/validateRequest';

app.get('/ohlcv', authMiddleware, validateOhlcvRequest, cacheMiddleware, async (req, res) => {
  // Existing handler logic
});
```

### Monitoring and Logging

- **Structured Logging:** Use libraries like **Winston** or **Pino** for structured logging.
- **Monitoring Metrics:** Integrate with monitoring tools like **Prometheus**, **Grafana**, or **Datadog** to track metrics such as request rates, error rates, and response times.
- **Distributed Tracing:** Implement tracing (e.g., using **Jaeger** or **Zipkin**) to trace requests across different components.

### Testing

- **Unit Testing:** Write unit tests for individual components, especially state machines and services.
- **Integration Testing:** Ensure that components interact correctly (e.g., Datastore Actor correctly communicates with Producers).
- **End-to-End Testing:** Simulate real user interactions to validate the entire data flow from data fetch to API response.

### Deployment Considerations

- **Containerization:** Use **Docker** to containerize the services for consistency across environments.
- **Orchestration:** Utilize **Kubernetes** or similar platforms for managing deployments, scaling, and failover.
- **CI/CD:** Implement continuous integration and deployment pipelines to automate testing and deployment processes.

---

## Technical Solution to Historical and Real-Time Data Synchronization

**Problem Statement:** When users request complete historical data with a specified start date, the system needs to integrate this data with real-time data feeds. The last timestamp in the historical data may not align perfectly with the current timestamp, especially for 1-minute OHLCV data.

**Solution Strategy:**

1. **Determine the Last Timestamp in TimescaleDB:**
   - Query TimescaleDB to find the latest timestamp available for the requested symbol and interval.

```
async function getLastTimestamp(symbol: string): Promise<Date | null> {
  const lastRecord = await Ohlcv.findOne({
    where: { symbol },
    order: [['bucket', 'DESC']],
  });
  return lastRecord ? new Date(lastRecord.bucket.getTime() + 60000) : null; // Next minute
}
```

2. **Fetch Historical Data:**
   - Retrieve OHLCV data from TimescaleDB starting from the user-specified start date up to the last available timestamp.

```
async function fetchHistoricalData(symbol: string, startDate: Date, endDate: Date):
↪ Promise<OhlcvData[]> {
  const records = await Ohlcv.findAll({
    where: {
      symbol,
      bucket: {
        [Op.gte]: startDate,
        [Op.lte]: endDate,
      },
    },
    order: [['bucket', 'ASC']],
  });
  return records.map(record => record.toJSON());
}
```

3. **Initiate Real-Time Data Fetching:**
   - Instruct the **Coordinator Actor** to create a dynamic topic starting from the next minute after the last timestamp.

```
async function initiateRealTimeFetch(symbol: string, lastTimestamp: Date) {
  const dynamicTopic = `${symbol}-rt-${lastTimestamp.toISOString()}`;
  coordinatorActor.send({
    type: 'REQUEST_CREATE_TOPIC',
    topicName: dynamicTopic,
    partitions: 1,
  });
}
```

4. **Consume Real-Time Data:**
   - Once the dynamic topic is created, a **Consumer Dynamic Topics Actor** subscribes to it and begins consuming real-time data starting from `lastTimestamp + 1 minute`.

```
async function subscribeToRealTimeData(symbol: string, dynamicTopic: string) {
  consumerDynamicActor.send({
    type: 'SUBSCRIBE_TO_TOPIC',
    topicName: dynamicTopic,
  });
}
```

5. **Merge Historical and Real-Time Data:**
   - Combine the fetched historical data with the real-time data stream, ensuring continuity without overlaps or gaps.

```
async function fetchOhlcvData(symbol: string, startDate: string, endDate?: string,
↪ interval: string = '1m'): Promise<OhlcvData[]> {
  const start = new Date(startDate);
  const end = endDate ? new Date(endDate) : new Date();

  // Step 1: Fetch Historical Data
  const lastTimestamp = await getLastTimestamp(symbol);
  const queryEndDate = lastTimestamp ? lastTimestamp : end;
  const historicalData = await fetchHistoricalData(symbol, start, queryEndDate);

  // Step 2: Initiate Real-Time Fetching if necessary
  if (!lastTimestamp || lastTimestamp < end) {
    await initiateRealTimeFetch(symbol, lastTimestamp || start);
```

```
    const dynamicTopic = `${symbol}-rt-${(lastTimestamp || start).toISOString()}`;
    await subscribeToRealTimeData(symbol, dynamicTopic);
  }

  // Step 3: Await Real-Time Data Fetch
  const realTimeData = await waitForRealTimeData(symbol, end);

  // Step 4: Merge Data
  return [...historicalData, ...realTimeData];
}
```

6. **Handling Data Retrieval in Real-Time Data Fetching:**
   - Implement mechanisms (like event listeners or message acknowledgment in XState) to collect real-time data once available.

**Considerations:** - **Synchronization Precision:** Ensure that `lastTimestamp + 1 minute` accurately aligns with real-time data ingestion to prevent overlaps or gaps. - **Concurrency:** Manage concurrent requests and data fetches without race conditions. - **Scalability:** Ensure that the system can handle multiple dynamic topics and corresponding consumers efficiently. - **Fault Tolerance:** Implement retries and failover mechanisms for dynamic topic creation and data fetching. - **Data Consistency:** Validate that merged data maintains temporal consistency and integrity.

---

## Further Enhancement

1. **Modular Development:**
   - Develop and test each component/modules individually before integrating.
   - Use isolation during testing to ensure each actor behaves as expected.
2. **Comprehensive Logging and Monitoring:**
   - Implement detailed logging within each actor to track state transitions and actions.
   - Utilize monitoring tools to observe system health, performance metrics, and handle alerts.
3. **Robust Error Handling:**
   - Ensure that all actors gracefully handle failures and implement retry mechanisms where appropriate.
   - Design provide fallback strategies or default states to maintain system stability.
4. **Scalability and Performance:**
   - Optimize database queries and cache usage to handle high-throughput scenarios.
   - Consider horizontal scaling for actors and services to manage increased load.
5. **Documentation and Maintenance:**
   - Maintain clear documentation for all components, state machines, and interactions.
   - Use tools like **Graphviz** diagrams to visualize and communicate system architecture.
6. **Security Considerations:**
   - Secure all data streams and API endpoints using encryption, authentication, and authorization mechanisms.
   - Regularly audit and update security protocols to safeguard data integrity and privacy.
7. **Testing Strategy:**
   - Implement unit tests for state machines and services.
   - Conduct integration tests to validate interactions between actors and services.
   - Perform end-to-end tests to simulate real-world data flows and API interactions.
8. **Continuous Integration and Deployment (CI/CD):**
   - Automate testing and deployment processes to ensure rapid and reliable delivery of updates.
   - Use containerization (e.g., Docker) and orchestration (e.g., Kubernetes) for consistent deployment environments.

---