Building a Real-Time Financial Analytics Dashboard: A Comprehensive Roadmap

I. Introduction

Purpose: This report provides a comprehensive roadmap, technical guidance, and curated resources for constructing a sophisticated real-time financial analytics dashboard. The objective is to offer a step-by-step plan suitable for a developer undertaking this complex full-stack project.

Project Vision: The goal is to create a web application inspired by the functionality and user interface of platforms like TradingView. This dashboard will feature live, interactive stock charts utilizing the Lightweight Charts library, a dynamic list displaying Nifty 50 stocks with real-time price updates, and a dedicated section for real-time sentiment analysis and market regime classification for selected stocks. The sentiment and regime insights will be derived from processing data streams originating from financial news sources and social media platforms.

Technology Stack Overview: The project leverages a modern, powerful technology stack designed for handling real-time data streams and complex analytics:

- **Frontend:** React.js (potentially with jQuery UI integration as specified), SCSS for styling, and the Lightweight Charts library for financial visualizations.
- **Backend:** Python with the FastAPI framework (or alternatively Java with Spring Boot), utilizing WebSockets for bidirectional real-time communication with the frontend.
- **Data Pipeline:** Apache Kafka for message queuing, Apache Flink for real-time stream processing (including Natural Language Processing for sentiment and regime logic), and TimescaleDB (an extension of PostgreSQL optimized for time-series data) for storing processed analytics.
- **Data Sources:** Dhan API for live and historical stock market data, News API (or alternatives) for financial news, and the Twitter (X) API and Reddit API for social media sentiment gathering.

This technology stack represents a robust, scalable architecture well-suited for the demands of real-time financial data processing and analysis. FastAPI, with its asynchronous capabilities [1] and strong support within the Python data science ecosystem, is an excellent choice for the backend, particularly for integrating NLP tasks. Apache Kafka and Apache Flink are industry-standard choices for building resilient, high-throughput streaming data pipelines.[3] TimescaleDB provides significant performance advantages over standard PostgreSQL for querying the time-series data that this application will generate and store.[7] While this combination offers substantial power and flexibility, it inherently introduces considerable complexity. Setting up, integrating, managing, and maintaining these distributed components requires careful planning and execution. Each piece (Kafka, Flink, TimescaleDB, APIs) has its own configuration nuances, scaling considerations, and potential failure modes that must be addressed throughout the development lifecycle.

Roadmap Structure: This report outlines a phased approach to building the application, starting with foundational setup and progressing through backend development, data pipeline construction, frontend implementation, integration, testing, and deployment. It also includes curated learning resources, alternative technology suggestions, and references to relevant open-source projects and articles.

II. Phase 0: Project Initiation and Environment Setup (Estimated Time: 1-2 Weeks)

This initial phase focuses on establishing the necessary development environments, configuring tools, and investigating the access requirements and limitations of the external APIs that will serve as data sources. Proper setup in this phase is crucial for a smooth development process.

A. Development Environment Configuration:

- **Local Setup Strategy:** Utilizing Docker and Docker Compose is highly recommended for managing the diverse set of services required (Kafka, Zookeeper, Flink, TimescaleDB/PostgreSQL, Backend API). This approach encapsulates dependencies, ensures environment consistency across development and potentially staging/production, and simplifies the setup process compared to installing each service natively.[3] A docker-compose.yml file will define and orchestrate these services.
- **Frontend (React/jQuery/SCSS):**
    - **Bootstrapping:** Use create-react-app (CRA) to initialize the React project structure. This provides a standardized setup with build configurations and development server ready.[17] Execute npx create-react-app frontend (or your preferred project name).
    - **SCSS Integration:** Modern versions of create-react-app (v2.0.0 and higher) include built-in support for Sass/SCSS.[18] Install the sass compiler via npm or yarn: npm install sass or yarn add sass.[18] Afterwards, .css files can be renamed to .scss, and imports updated accordingly (e.g., import './App.scss';).[20] No

ejection from CRA is necessary for basic SCSS usage.[18]

  - **jQuery UI Integration:** Integrating jQuery UI directly within a React application requires careful consideration. React manages the DOM via its virtual DOM, while jQuery and jQuery UI manipulate the DOM directly. This fundamental difference can lead to conflicts and unpredictable behavior if not managed properly. While technically possible by installing jQuery (npm install jquery) and initializing/destroying jQuery UI widgets within React's useEffect hook (to synchronize with the component lifecycle), this approach is generally discouraged in modern React development.[17] Evaluate if the specific jQuery UI components are essential or if native React components or React-specific UI libraries could provide similar functionality more harmoniously within the React ecosystem. If jQuery UI is non-negotiable, investigate React wrapper libraries for the specific widgets needed or prepare for careful lifecycle management within useEffect.
- **Backend (Python/FastAPI):**
  - **Virtual Environment:** Create and activate a Python virtual environment to isolate project dependencies (e.g., python -m venv venv followed by source venv/bin/activate on Unix-like systems).[22]
  - **Installation:** Install FastAPI and the Uvicorn ASGI server. Using the [standard] option installs helpful defaults, including Uvicorn: pip install "fastapi[standard]".[1]
  - **Basic Structure:** Create a main.py file. Import FastAPI, instantiate the app (app = FastAPI()), and define a root path operation (@app.get("/")) as a starting point.[1]
  - **Running the Server:** Use Uvicorn to run the application. The --reload flag enables auto-reloading during development: uvicorn main:app --reload.[23]
- **Backend (Java/Spring Boot - Alternative):**
  - **Bootstrapping:** Utilize Spring Initializr (start.spring.io or via IDE integration) to generate the project structure.[26] Select Maven or Gradle as the build tool and include "WebSocket" and "Spring Web" as dependencies.
  - **Basic Structure:** The generated project will include a main application class annotated with @SpringBootApplication.
  - **WebSocket Configuration:** Create a configuration class (e.g., WebSocketConfig) annotated with @Configuration and @EnableWebSocketMessageBroker. Implement the WebSocketMessageBrokerConfigurer interface.[26] Override configureMessageBroker to enable a message broker (e.g., simple in-memory broker) and define application destination prefixes (e.g., /app) and broker prefixes (e.g., /topic, /queue).[26] Override registerStompEndpoints to define the

WebSocket handshake endpoint (e.g., /ws or /greeting as seen in examples) and potentially enable SockJS fallbacks (withSockJS()).[26]

- **Data Pipeline (Docker Compose):**
  - **docker-compose.yml:** Create this file at the project root to define and configure the pipeline services.[3]
  - **Services:**
    - **Zookeeper:** Required by Kafka for coordination.[3] Use an official image like confluentinc/cp-zookeeper:[tag].[14] Expose the client port (2181).
    - **Kafka:** Use an official image like confluentinc/cp-kafka:[tag].[14] Expose the broker port (e.g., map host port 19092 to container port 9092).[3] **Crucially**, configure environment variables like KAFKA_ZOOKEEPER_CONNECT, KAFKA_LISTENER_SECURITY_PROTOCOL_MAP, KAFKA_ADVERTISED_LISTENERS, and KAFKA_INTER_BROKER_LISTENER_NAME. KAFKA_ADVERTISED_LISTENERS is vital for connectivity; it must define addresses accessible by *all* clients (other containers within the Docker network, and potentially the host machine or backend service). For instance, INTERNAL://kafka:9092,EXTERNAL://localhost:19092 might be needed if both internal containers (like Flink) and external applications (like a producer running on the host) need to connect. Incorrectly configured listeners are a frequent cause of connection issues in Dockerized Kafka setups.[16]
    - **Flink:** Include separate services for jobmanager and taskmanager.[3] Use official Flink images (flink:[tag]). Expose the JobManager Web UI port (e.g., 8081 or 8083).[3] Ensure the Flink containers have access to necessary connector JARs (like flink-connector-kafka) by mounting them into the /opt/flink/lib directory within the containers, or building a custom Flink image.[14] Configure dependencies between Flink and Kafka/Zookeeper using depends_on.
    - **TimescaleDB/PostgreSQL:** Use an official TimescaleDB image, preferably the -ha variant which includes helpful extensions like PostGIS and Toolkit.[7] Example: timescale/timescaledb-ha:pg16-latest.[7] Expose the PostgreSQL port (5432).[7] Set the POSTGRES_PASSWORD environment variable.[7] Configure a persistent volume using Docker volumes or bind mounts (volumes: - timescale_data:/var/lib/postgresql/data) to prevent data loss when containers restart.[7] The TimescaleDB extension should be enabled by default in these images, but verification via psql (\dx) after startup is recommended.[7]
  - **Networking:** Define a custom Docker network (e.g., services_network) and

attach all services to it. This allows services to communicate using their service names as hostnames (e.g., kafka:9092 from within the Flink container).[8]

- **Version Control:** Initialize a Git repository at the project root. Create a .gitignore file to exclude environment files (.env), virtual environments (venv/), build artifacts (node_modules/, target/), IDE configurations, and sensitive data.

B. API Access and Investigation:

A critical early step is to understand the capabilities, limitations, and costs associated with each external API. This investigation will inform budget requirements, potential scope adjustments, and implementation strategies.

- **Dhan API:**
    - **Access & Cost:** Requires a Dhan trading account. The core Trading APIs (order placement, portfolio management) are free.[30] Data APIs, which include the necessary Live Market Feed (WebSocket) and Historical Data API, incur a monthly fee of ₹499 + taxes. This fee is waived if the account executes at least 25 trades in the Futures & Options (F&O) segment within the preceding 30 days.[30]
    - **Authentication:** Access is granted via an Access Token generated through the Dhan Web platform under the Profile section ("DhanHQ Trading APIs").[30]
    - **Real-time Data:** The primary mechanism for real-time data is WebSocket.[33] The v2 feed supports Ticker (LTP, LTT), Quote (includes OHLC, volume, total buy/sell quantity), and Full (includes Quote data plus market depth and Open Interest for derivatives) data packets.[34] Data is transmitted in a binary format for performance reasons.[33] A user can establish up to 5 concurrent WebSocket connections, and each connection can subscribe to up to 5000 instruments [34] (an increase from earlier limits mentioned in v1 docs [33]). Subscriptions are managed by sending JSON messages over the established WebSocket connection, with a limit of 100 instruments per subscription message.[34] The connection requires an initial binary authorization packet and utilizes a server-sent Ping/Pong mechanism for keep-alive.[33] Disconnection events are signaled with specific reason codes.[33]
    - **Nifty 50 Data:** To retrieve data specifically for the Nifty 50 index, the correct Security ID (13) must be used in conjunction with the Index (IDX) exchange segment code (0) when subscribing via WebSocket or querying REST endpoints.[35] Using incorrect IDs (like Security ID 2) may result in data for a different index (e.g., BSEDSI).[35]
    - **REST APIs:** Snapshot data (LTP, OHLC, Market Depth) can be fetched via

  POST requests to specific endpoints (/marketfeed/ltp, /marketfeed/ohlc, /marketfeed/quote).[36]
  - **Rate Limits:** Dhan provides relatively generous rate limits. Order placement APIs are limited to 25 requests/second, 250/minute, 1000/hour, and 7000/day.[37] REST-based Data APIs (like historical data) have a limit of 5 requests/second and 100,000/day.[37] REST Quote APIs (LTP, OHLC, Depth snapshots) are limited to 1 request/second.[37] Non-Trading APIs (e.g., fetching holdings, funds) allow 20 requests/second.[37] Per-minute and per-hour limits for non-trading APIs have been relaxed or removed.[39] Order modifications are capped at 25 per order.[32] These limits are generally sufficient for typical algorithmic trading and dashboard applications but should be kept in mind.
- **Twitter (X) API:**
  - **Access Tiers & Cost:** Access to the Twitter API has become significantly restricted and tiered.
    - **Free:** Extremely limited, primarily write-only (1,500 posts/month at the *app* level), unsuitable for reading real-time data.[42]
    - **Basic:** $100/month. Allows reading up to 10,000 tweets/month and posting 50,000/month (app level), or 3,000/month (user level).[42] Likely insufficient for comprehensive real-time monitoring.
    - **Pro:** $5,000/month. Allows reading 1 million tweets/month and posting 300,000/month.[42] Might be viable for a focused use case but represents a significant cost.
    - **Enterprise:** Custom pricing, potentially starting at $42,000/month or higher.[42] Offers the highest limits, full-archive search, and potentially better real-time streaming access.
  - **Rate Limits:** Vary drastically based on the chosen tier and specific API endpoint (v2 API is the current standard).[42] For example, searching recent tweets (/2/tweets/search/recent) might be limited to 180 requests per 15 minutes on the Free tier (if read access were available) versus 450/15-min on Basic.[43] Limits apply either per-user or per-app depending on the authentication context.[44] POST actions (tweeting) often have longer time windows (e.g., per 3 hours or 24 hours).[44] Exceeding limits results in errors (HTTP 429).[43]
  - **Real-time Access:** While streaming endpoints exist in the API, meaningful access for filtering real-time tweets based on keywords (like stock symbols) likely requires a Pro or Enterprise plan.
  - **Authentication:** OAuth 2.0 is mandatory for most interactions.[44]
  - **Limitations:** The API landscape shifted significantly after Elon Musk's acquisition, making broad data access much more expensive and restricted.[43]

Accessing historical data beyond recent tweets (full-archive search) is typically an Enterprise-only feature.[43]

- **News API (newsapi.org):**
    - **Access Tiers & Cost:**
        - **Developer:** Free, but strictly for development/testing, non-commercial use. Limited to 100 requests/day, data is delayed, and historical search is limited.[47] Not suitable for the production application.
        - **Business:** $449/month. Provides 250,000 requests/month, real-time access to new articles, search up to 5 years old, CORS support.[47] This is likely the minimum viable plan for this project.
        - **Enterprise:** Custom pricing for higher volumes and potentially SLAs.[47]
    - **Real-time:** Real-time news delivery is a feature of the paid plans (Business and above).[47]
    - **Data & Filtering:** Provides access to news articles and top headlines from numerous sources worldwide. Filtering by keywords (e.g., "AAPL", "Nifty 50", "stock market"), sources (e.g., "Bloomberg", "Reuters"), categories (e.g., "business", "technology"), language, and date ranges is supported.
    - **Rate Limits:** Governed by the monthly request quota of the chosen plan. Overage charges apply if the quota is exceeded (e.g., $0.0018 per extra request on the Business plan).[47]
    - **Finance Focus:** This is a general news API. Effective use for financial sentiment requires careful crafting of search queries (keywords, sources) to isolate relevant financial news.
- **Alternative News APIs (Finance Focused):** Several APIs specialize in financial news, potentially offering better filtering or relevance:
    - **Financelayer (apilayer.com):** Offers a free tier (100 requests/month, 60-minute delay) and paid plans ($15-$100/month) with higher limits and real-time data.[48] Allows filtering by keywords, sources, and stock tickers.[48]
    - **Real-Time Finance Data (RapidAPI/OpenWeb Ninja):** Provides a mix of market quotes, trends, and news. Has a free tier (200 requests/month, hard limit) and paid tiers ($25-$150/month).[49] Free plan has an additional hourly limit (1000 reqs/hour).[49] Rate limits (RPS and monthly quota) apply based on the plan.[49]
    - **EOD Historical Data (eodhd.com):** Uses a unique "API calls" credit system. News API requests consume 5 API calls each. Paid plans typically start with a daily limit of 100,000 API calls, which can be increased.[50] There's also a per-minute *request* limit of 1000.[50] Extra API calls can be purchased.[50]
    - **Alpaca News API:** Currently in beta, offered free with rate limits tied to Alpaca Market Data subscription plans (200 calls/minute for Free,

  10,000/minute for Unlimited).[51] Provides real-time news via WebSocket and historical data (6+ years) initially sourced from Benzinga.[51] Pricing may change after the beta period.[51]
  - **Finnhub.io:** Offers a Market News endpoint. Access is tiered (Free to various paid levels). Subject to overall API rate limits (e.g., 30 calls/second across all endpoints) plus plan-specific limits.[52] Real-time news likely delivered via WebSocket.[52]
- **Reddit API:**
  - **Access Tiers & Cost:**
    - **Free Tier:** Intended for non-commercial use, moderation tools, and academic research. Subject to rate limits.[54]
    - **Paid Tier:** For commercial applications or those exceeding free tier limits. Priced at $0.24 per 1,000 API calls.[55]
    - **Enterprise Tier:** For very large-scale applications, custom pricing.[56]
  - **Rate Limits:** The free tier is limited to 100 Queries Per Minute (QPM) per OAuth client ID when using OAuth authentication.[46] Without OAuth, the limit drops to 10 QPM.[55] These limits are averaged over a 10-minute window to allow for bursts.[54] Different OAuth grant types might have different effective limits (e.g., app-only OAuth might be lower).[58] API responses include headers (X-Ratelimit-Used, X-Ratelimit-Remaining, X-Ratelimit-Reset) to monitor usage.[54]
  - **Authentication:** OAuth 2.0 authentication is required for any meaningful access and to benefit from the higher free tier rate limit.[46] A unique and descriptive User-Agent string must also be provided in requests.[54]
  - **Data Access:** The API allows programmatic access to posts (submissions), comments, user profiles, subreddit information, etc..[46] Filtering by subreddit, searching for keywords, and retrieving comment trees are common operations. Access to NSFW (mature) content via the API is restricted.[55] Adherence to data privacy and deletion policies is mandatory (e.g., removing content if deleted on Reddit).[54]
  - **Pushshift:** Pushshift.io, previously a valuable resource for accessing historical and near real-time Reddit data archives, is **no longer publicly accessible**.[59] Following Reddit API changes, Pushshift access is now restricted solely to approved Reddit moderators for moderation purposes only.[59] This requires moderators to register, obtain approval from Reddit, and use temporary (24-hour) OAuth tokens provided by Pushshift.[61] Therefore, Pushshift is **not a viable option** for this project's goal of general sentiment analysis. While historical data dumps from Pushshift exist (e.g., via Academic Torrents [64]), they are not suitable for the required real-time analysis.

- **API Comparison Summary:** The feasibility and cost of acquiring real-time data vary significantly across the required sources. Dhan's API appears the most accessible for core stock data, with free trading APIs and a reasonably priced data API subscription.[30] However, obtaining real-time, high-volume data from Twitter/X and Reddit for sentiment analysis will likely require substantial investment in their paid API tiers due to the severe limitations of the free tiers.[42] Similarly, real-time news requires a paid subscription to a service like NewsAPI.org or a finance-focused alternative.[47] This necessitates a careful budget allocation for data acquisition or a potential reduction in the scope of the real-time sentiment features, perhaps focusing initially only on news-based sentiment or a very limited set of stocks/social media sources.
Table 1: API Data Source Comparison

API Provider	Relevant Data	Access Tiers/Pricing (Monthly Est.)	Key Rate Limits (Real-time/ Relevant Tier)	Real-time Method	Key Limitations/ Notes
Dhan	Stock Quotes, Depth, Trades	Trading: Free; Data: ₹499+tax (or free w/ 25 F&O trades) [30]	WebSocket: 5 connections, 5k instruments/ conn. REST: 1-5 req/sec (Quote/Data) [34]	WebSocket (Binary)	Data API subscription needed for WebSocket feed. Binary format requires parsing.
Twitter (X) API	Tweets	Free: N/A for reading; Basic: $100; Pro: $5000; Enterprise: $42k+ [42]	Basic: 10k reads/month total, 450 req/15min (recent search).[42] Pro: 1M reads/month .	Streaming API (Paid Tiers)	Free tier unusable for reading. Significant cost for meaningful real-time volume. Volatile API landscape.
NewsAPI.or	News	Developer:	Business:	Polling (Paid	Paid plan

g	Articles	Free (Dev only, delayed); Business: $449; Enterprise: Custom [47]	250k req/month included.[47]	Tiers)	needed for real-time & commercial use. General news, requires filtering.
Financelayer	Financial News	Free: 100 req/mo (delayed); Paid: $15-$100 [48]	Varies by plan.	Polling (Paid Tiers)	Paid plan needed for real-time. Focused on finance.
Alpaca News API	Financial News (Benzinga)	Free (Beta, tied to Market Data plan) [51]	200-10k calls/min (based on data plan) [51]	WebSocket	Currently in Beta, pricing may change. Limited sources initially.
Reddit API	Posts, Comments	Free: Non-commercial; Paid: $0.24/1k calls [55]	Free (OAuth): 100 QPM.[54]	Polling	100 QPM likely insufficient for broad real-time monitoring. Paid tier cost scales with usage. Pushshift unavailable.
Pushshift.io	Historical Reddit Data Archive	N/A (Restricted Access) [61]	N/A	N/A	Access restricted to approved Reddit moderators for moderation only. Not usable for this project's real-time

					goals.

III. Phase 1: Backend Development - Core Setup & Stock Data (Estimated Time: 2-3 Weeks)

With the environment configured and API access understood, this phase focuses on building the foundational backend components and integrating the primary stock data source.

A. FastAPI Server Setup:

- **Project Structure:** Organize the backend code logically. A common pattern involves directories for routes (API endpoint definitions), models (Pydantic data models), services (business logic, API interactions), core (configuration, database connections), and tests.
- **Configuration Management:** Implement a way to manage sensitive information like API keys and database credentials securely. Environment variables (loaded using libraries like python-dotenv in development) or dedicated configuration files are standard practices. Avoid hardcoding secrets directly in the source code.
- **Initial API Routes:** Define the basic API structure using FastAPI's decorators. Start with essential endpoints, such as a health check (/health) and placeholders for data needed by the frontend (e.g., /api/nifty50-list).[1]
- **Data Validation:** Utilize Pydantic models to define the expected structure and data types for request bodies and response payloads.[22] FastAPI automatically uses these models for data validation and serialization, improving robustness and generating interactive API documentation (Swagger UI/ReDoc).[25]

B. Dhan API Integration (Stock Data):

- **Connectivity:** Implement the logic to authenticate and interact with the Dhan API. This involves using the generated Access Token in the request headers (access-token) along with the client-id.[36] Consider using the official DhanHQ-py Python library provided by Dhan, as it likely encapsulates authentication and request formatting details, simplifying integration.[30] If not using the library, carefully follow the API documentation for header requirements and endpoint specifications.
- **Nifty 50 List:** Develop a service function that calls the appropriate Dhan REST API endpoint (e.g., an instrument discovery or search endpoint, filtered for the Nifty 50 index constituents) to fetch the list of stocks belonging to the Nifty 50 index.[35] This list might include security IDs, trading symbols, and other static details needed by the frontend.

- **Backend Endpoint:** Create a specific API endpoint in FastAPI (e.g., GET /api/nifty50-list) that utilizes the service function to retrieve the Nifty 50 list and returns it to the frontend, likely as a JSON array.

C. Initial WebSocket Setup (Foundation):

- **Endpoint Definition:** Define a basic WebSocket endpoint within the FastAPI application using the @app.websocket decorator. Choose a relevant path, for example, /ws/stocks.[2]
- **Connection Handling:** Implement the initial handshake logic within the WebSocket endpoint function. This involves accepting the incoming connection using await websocket.accept().[68] Optionally, send an initial confirmation message to the client upon successful connection.
- **Connection Management:** Create a simple Python class (e.g., ConnectionManager) to keep track of active WebSocket connections. This class should have methods like connect(websocket: WebSocket) to add a new connection to a set or list, and disconnect(websocket: WebSocket) to remove it when the connection closes.[68] Instantiate this manager globally or pass it via dependency injection. Even if broadcasting isn't implemented yet, tracking connections is fundamental.

Establishing the basic REST endpoints for static data like the Nifty 50 list serves as an important first step. It allows verification of the core backend setup, configuration management, and fundamental connectivity with the Dhan API before tackling the more complex real-time streaming aspects. Concurrently laying the groundwork for WebSocket communication by defining the endpoint and a connection manager prepares the architecture for the subsequent phase focused on live data integration. This incremental approach reduces risk and allows for iterative testing.

IV. Phase 2: Backend Development - Real-time Data Streaming (Estimated Time: 3-4 Weeks)

This phase focuses on integrating the real-time stock data feed from Dhan and broadcasting these updates to connected frontend clients via the backend's WebSocket connection.

A. Dhan WebSocket Integration:

- **Client Connection:** Within the backend application (potentially in a background task or a dedicated service), implement the client logic to establish a WebSocket connection to Dhan's Market Feed endpoint (e.g., wss://api.dhan.co/market-feed/v2/ - check docs for the exact v2 URL).[33] Use a suitable Python WebSocket client library (like websockets).

- **Authorization:** Handle the specific binary authorization packet that Dhan requires immediately after connection establishment.[33] This involves sending a precisely structured binary message containing authentication details.
- **Subscription Management:** Implement logic to send subscription requests over the Dhan WebSocket. For v2, these are JSON messages.[34] Subscribe to the required instruments: the Nifty 50 constituents for the list view and potentially the currently selected stock for more detailed data (Quote or Full packets). Remember the limit of 100 instruments per subscription message; multiple messages might be needed for the full Nifty 50.[34]
- **Binary Data Parsing:** This is a critical and potentially complex part. Implement the logic to receive and parse the incoming binary data packets from Dhan.[33] Based on the Response Header bytes (specifically the message code), determine the packet type (Ticker, Quote, Full, Prev Close, etc.). Then, parse the subsequent bytes according to the documented structure for that packet type, extracting fields like LTP (float32), LTT (int32), quantity (int16/int32), depth levels, etc., paying close attention to data types and byte offsets.[34] Libraries like Python's struct module can be helpful for unpacking binary data.
- **Keep-Alive:** Ensure the client responds correctly to the server's Ping messages to maintain the connection (the websockets library often handles this automatically, but verify).[33]
- **Disconnection Handling:** Implement robust error handling and reconnection logic. Listen for disconnection packets from Dhan, log the reason code, and attempt to re-establish the connection and re-subscribe to instruments after a delay.[33]

B. Backend WebSocket Broadcasting:

- **Connection Manager Enhancement:** Extend the ConnectionManager class created in Phase 1. Add methods like broadcast(message: str) or send_personal_message(message: str, websocket: WebSocket).[68] The broadcast method will iterate through the set of active_connections and send the message to each connected frontend client.
- **Data Forwarding:** As the backend receives and successfully parses binary data from the Dhan WebSocket feed, transform this data into a suitable format for frontend consumption (likely JSON). Use the ConnectionManager's broadcast method to send these updates (e.g., LTP changes for the Nifty 50 list, or detailed Quote/Depth updates for a selected stock) to all relevant connected frontend clients via the backend's own WebSocket endpoint (/ws/stocks).

- **Filtering/Routing:** Decide on the broadcasting strategy. Should all clients receive all Nifty 50 ticker updates? Or should detailed Quote/Depth updates only be sent to clients who have explicitly selected that stock? Implement filtering logic within the backend before broadcasting if necessary. Define a clear JSON message structure for communication with the frontend (e.g., specifying message type, stock symbol, and payload).

C. API Endpoints for Sentiment/Regime Data:

- **Placeholder Endpoints:** Define the API endpoints that the frontend will eventually use to fetch sentiment and market regime data. Use FastAPI decorators:
  - @app.get("/api/sentiment/{stock_symbol}")
  - @app.get("/api/regime/{stock_symbol}")
- **Initial Implementation:** For now, these endpoints can return static mock data (e.g., {"sentiment": "neutral", "score": 0.5}) or a simple {"status": "not implemented"} response. They will be fully implemented after the data pipeline (Phase 4) is operational and populating TimescaleDB or relevant Kafka topics.

The requirement to handle a binary WebSocket protocol from Dhan introduces a layer of complexity not present with typical JSON-based WebSockets.[33] Careful implementation of byte-level parsing according to the Dhan documentation is essential. Furthermore, the connection to the external Dhan feed is a potential point of failure. Building resilient logic to handle disconnections, manage errors, and automatically attempt reconnection is critical for the stability and reliability of the entire real-time data flow to the end-users.

Simultaneously, efficiently managing the backend's own WebSocket connections to frontend clients is important for performance. As the number of connected clients grows, the simple broadcasting loop shown in basic examples [68] might become a bottleneck. Consider asynchronous broadcasting techniques or internal queuing within the backend if performance issues arise during testing with multiple clients. The ConnectionManager [69] design should anticipate potential scaling needs.

V. Phase 3: Data Engineering Pipeline - Setup & Ingestion (Estimated Time: 3-4 Weeks)

This phase focuses on setting up the core infrastructure for the data pipeline (Kafka, TimescaleDB) and building the producer applications that will fetch data from external sources (News, Twitter, Reddit) and ingest it into Kafka.

A. Kafka Cluster Setup & Topic Creation:

- **Verification:** Ensure the Kafka and Zookeeper services defined in the docker-compose.yml (from Phase 0) are running correctly. Use docker-compose ps to check status and docker-compose logs <service_name> to inspect logs.
- **Topic Creation:** Utilize the Kafka command-line tools, executed within the running Kafka container, to create the necessary topics for the pipeline.[12] Connect to the container using docker-compose exec kafka bash (or similar) and then run the kafka-topics.sh script:
    - kafka-topics.sh --create --topic raw_news --bootstrap-server kafka:9092 --partitions 1 --replication-factor 1
    - kafka-topics.sh --create --topic raw_tweets --bootstrap-server kafka:9092 --partitions 1 --replication-factor 1
    - kafka-topics.sh --create --topic raw_reddit_posts --bootstrap-server kafka:9092 --partitions 1 --replication-factor 1
    - kafka-topics.sh --create --topic processed_sentiment --bootstrap-server kafka:9092 --partitions 1 --replication-factor 1
    - kafka-topics.sh --create --topic market_regimes --bootstrap-server kafka:9092 --partitions 1 --replication-factor 1
    - *(Note: bootstrap-server uses the internal Docker network hostname kafka:9092. Partitions and replication factor are set to 1 for simplicity in a local Docker setup).*[12] List topics using kafka-topics.sh --list --bootstrap-server kafka:9092 to verify creation.[12]

B. Data Source Producers (Python Scripts):

Develop individual Python scripts responsible for fetching data from each source and publishing it to the corresponding Kafka topic. These can run as separate Docker containers defined in the docker-compose.yml or be executed locally (ensuring they can connect to the Kafka broker exposed by Docker, e.g., localhost:19092). Use a Python Kafka client library like kafka-python.[12]

- **News Producer:**
    - Select a News API provider (e.g., NewsAPI.org Business plan [47], Financelayer [48], Alpaca News API [51]) based on the investigation in Phase 0.
    - Use the provider's recommended client library or the standard requests library to fetch news articles. Implement logic to periodically query the API based on relevant financial keywords (Nifty 50 stock symbols/names, "stock market", "earnings", etc.) and potentially filter by reputable financial sources.
    - Handle API key authentication securely. Implement robust error handling and respect the API's rate limits.[47]

    - Format the fetched articles (e.g., title, content snippet, source, timestamp, URL) into a consistent JSON structure.
    - Initialize a KafkaProducer instance connected to the Kafka broker (e.g., localhost:19092 if running locally and port is mapped, or kafka:9092 if running within the Docker network).
    - Publish the JSON-formatted news articles to the raw_news Kafka topic.
- **Twitter Producer:**
    - **Prerequisite:** A paid Twitter API subscription (Pro or Enterprise) is almost certainly required for gathering sufficient real-time data for this project's goals.[42] The Free and Basic tiers have prohibitive read limitations.
    - Use a reliable Twitter API v2 client library (e.g., tweepy [70] or others).
    - Implement logic to connect to the Twitter API's streaming endpoint (if available under the chosen plan) or periodically poll search endpoints. Filter tweets based on relevant keywords (e.g., cashtags like $RELIANCE, $INFY), mentions of Nifty 50 company names, or potentially track specific financial news accounts.
    - Handle OAuth 2.0 authentication flow securely.
    - Implement strict adherence to the rate limits associated with the chosen API tier and endpoints.[42] Use strategies like backoff and retry for rate limit errors (HTTP 429).
    - Format relevant tweet data (text, timestamp, user info, potentially engagement metrics) into JSON.
    - Publish the JSON data to the raw_tweets Kafka topic.
- **Reddit Producer:**
    - Utilize the official Reddit API, not Pushshift.[60]
    - Use a Python Reddit API wrapper like PRAW (praw).[46]
    - Implement OAuth 2.0 authentication (script-based flow might be suitable). Remember the User-Agent requirement.[54]
    - Develop logic to monitor specific relevant subreddits (e.g., r/IndianStockMarket, r/stocks, company-specific subreddits) for new posts and comments. Filter content based on keywords (stock symbols/names).
    - Carefully manage API calls to stay within the free tier rate limit (100 QPM with OAuth) or budget for the paid tier ($0.24/1k calls) if higher volume is needed.[54] Use the rate limit headers in responses to monitor usage.[54]
    - Extract relevant information (post title, body, comment text, author, timestamp, subreddit) and format as JSON.
    - Publish the JSON data to the raw_reddit_posts Kafka topic.

C. TimescaleDB/PostgreSQL Setup:

- **Verification:** Ensure the TimescaleDB container is running and accessible (Phase 0).
- **Schema Definition:** Connect to the database using psql or a database client tool.[7] Execute SQL commands to create the necessary tables for storing the processed data.
    - **Sentiment Table:**
      SQL
      ```sql
      CREATE TABLE stock_sentiment (
        time TIMESTAMPTZ NOT NULL,
        stock_symbol VARCHAR(10) NOT NULL,
        source VARCHAR(50), -- e.g., 'twitter', 'newsapi', 'reddit'
        sentiment_score DOUBLE PRECISION, -- e.g., VADER compound score -1 to 1
        sentiment_label VARCHAR(10), -- e.g., 'positive', 'negative', 'neutral'
        raw_text TEXT,
        source_id VARCHAR(255) -- Unique ID from the source (tweet ID, article URL hash, etc.)
      );
      -- Add unique constraint to prevent duplicates if needed
      -- ALTER TABLE stock_sentiment ADD CONSTRAINT unique_sentiment_entry UNIQUE (time, stock_symbol, source, source_id);
      ```
    - **Market Regime Table:**
      SQL
      ```sql
      CREATE TABLE market_regimes (
        time TIMESTAMPTZ NOT NULL,
        stock_symbol VARCHAR(10) NOT NULL,
        regime_type VARCHAR(50) NOT NULL, -- e.g., 'Trend', 'Volatility', 'Liquidity'
        regime_label VARCHAR(50) NOT NULL, -- e.g., 'Trending Up', 'Low Volatility', 'Risk-On'
        score DOUBLE PRECISION, -- Optional confidence score
        PRIMARY KEY (time, stock_symbol, regime_type) -- Composite key example
      );
      ```
- **Hypertable Conversion:** Convert the created tables into TimescaleDB hypertables. This is essential for leveraging TimescaleDB's time-series optimizations.[8]
  SQL
  ```sql
  -- Enable the TimescaleDB extension if not already done (should be automatic in HA images)
  CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;

  -- Convert tables to hypertables, partitioning by the 'time' column
  SELECT create_hypertable('stock_sentiment', 'time');
  ```

```sql
SELECT create_hypertable('market_regimes', 'time');
```

- **Indexing:** Create indexes on frequently queried columns, especially stock_symbol and time, to improve query performance. TimescaleDB automatically creates an index on the time column for hypertables.
  SQL
  ```sql
  CREATE INDEX idx_sentiment_symbol_time ON stock_sentiment (stock_symbol, time DESC);
  CREATE INDEX idx_regime_symbol_time ON market_regimes (stock_symbol, time DESC);
  ```

The success of the data ingestion phase, particularly for social media, hinges significantly on the available budget for paid APIs and the ability to navigate the respective platforms' restrictions and rate limits effectively. It is pragmatic to start by implementing the News API producer, as it presents fewer technical hurdles compared to the real-time complexities and stringent limitations of the Twitter and Reddit APIs. This allows for establishing a functional data flow into Kafka before tackling the more challenging sources.

Similarly, defining the TimescaleDB schema and converting tables to hypertables early using create_hypertable is crucial.[8] This ensures that the database is optimized for the time-series nature of the incoming sentiment and regime data from the outset. Hypertables automatically handle time-based partitioning, which is fundamental for achieving efficient query performance as data volume grows.[9] Neglecting this step and using standard PostgreSQL tables would undermine the primary reason for choosing TimescaleDB and likely lead to performance degradation later.

VI. Phase 4: Data Engineering Pipeline - Real-time Processing (Flink) (Estimated Time: 4-6 Weeks)

This phase involves building the Apache Flink application(s) to process the raw data streams from Kafka, perform sentiment analysis and market regime detection, and sink the results into TimescaleDB and/or other Kafka topics.

A. Flink Job Setup:

- **Development:** Choose the Flink API for development. The Python DataStream API offers integration with Python's rich NLP ecosystem [14], while the Java/Scala DataStream API might offer performance benefits and more mature connector options.[71]

- **Connectors:**
  - **Kafka Source:** Configure Flink Kafka Consumers within the job to read from the raw_news, raw_tweets, and raw_reddit_posts topics created in Phase 3.[3] Specify the bootstrap servers (e.g., kafka:9092), topic names, consumer group ID, and a deserialization schema (e.g., JsonDeserializationSchema or a custom deserializer) to parse the incoming JSON messages.
  - **Kafka Sink:** Configure Flink Kafka Producers to write processed results (sentiment, regimes) to the processed_sentiment and market_regimes topics if downstream consumers (like the backend) need to react to these events via Kafka. Specify bootstrap servers, topic names, and a serialization schema (e.g., JsonSerializationSchema).
  - **TimescaleDB/PostgreSQL Sink:** Configure a sink to write the final processed data directly into the TimescaleDB hypertables created in Phase 3. Flink's JdbcSink is a viable option.[10] Configure the JDBC URL (jdbc:postgresql://timescaledb:5432/postgres or your specific DB name), username, password, and the SQL insert statement. Parameterize the insert statement to map fields from the Flink data stream to the table columns. Consider configuring batching for better performance. For exactly-once semantics, investigate if the JDBC sink supports two-phase commits or design the sink operation to be idempotent.
- **Checkpointing:** Enable Flink checkpointing within the job configuration. Checkpoints periodically save the state of the application (including Kafka consumer offsets) to a durable storage (configurable, can be filesystem in Docker for local dev). This is crucial for fault tolerance and achieving exactly-once or at-least-once processing guarantees.[4] Set an appropriate checkpoint interval (e.g., every 1-5 minutes).

B. Sentiment Analysis Implementation:

- **NLP Integration:** Within a Flink operator (e.g., a MapFunction or ProcessFunction), integrate a suitable Python NLP library.
  - **Lexicon-based:** NLTK's VADER (SentimentIntensityAnalyzer) is a good starting point, specifically tuned for social media text and providing positive, negative, neutral, and compound scores.[74] TextBlob also offers polarity and subjectivity scores, potentially using pattern-based or Naive Bayes analyzers.[74] These are generally faster and require less setup.
  - **Machine Learning:** For potentially higher accuracy (at the cost of complexity and resource usage), consider libraries like spaCy [74] or Hugging Face Transformers [74] to load pre-trained sentiment analysis models. This might require managing model files within the Flink environment.

    - **External AI Services:** If advanced analysis (like explanation generation [75] or complex classification [71]) is needed, the Flink job could make API calls to services like OpenAI.[71] However, this introduces external dependencies, potential latency bottlenecks, and significant costs.
- **Processing Logic:**
    1. Receive raw messages (news, tweets, posts) from Kafka sources.
    2. Extract the text content. Perform necessary text cleaning (e.g., remove URLs, special characters, lowercase).
    3. Apply the chosen sentiment analysis tool/model to the cleaned text to get a sentiment score/label.
    4. Implement logic (e.g., using regular expressions or keyword matching) to identify and extract relevant stock symbols (e.g., "$AAPL", "Reliance Industries") mentioned in the text. A single message might relate to multiple symbols.
    5. For each identified symbol, create an output record containing the timestamp, stock symbol, data source (news/twitter/reddit), calculated sentiment score/label, and potentially the original text or source ID.[75]
    6. Send the structured output records to the processed_sentiment Kafka topic and/or the TimescaleDB sink.

C. Market Regime Detection Implementation:

This is arguably the most complex part of the data processing logic, requiring both financial domain knowledge and proficiency in Flink's stateful stream processing capabilities.

- **Data Requirements:** Regime detection often requires more than just sentiment. Price and volume data are essential for trend and volatility analysis. This data can be sourced directly from the Dhan WebSocket feed (Phase 2). This might necessitate:
    - The backend service (Phase 2) also producing parsed Dhan data (LTP, volume) to a dedicated Kafka topic (e.g., stock_ticks).
    - A separate Flink job consuming stock_ticks to perform price-based regime calculations.
    - Alternatively, integrating price/volume lookups or streams directly into the main Flink processing job if feasible.
    - Liquidity analysis might require Market Depth data from Dhan [33], adding further complexity to data ingestion and processing.
- **Algorithm Definition & Implementation:**
    - **Trending Up/Down:** Requires stateful processing over time windows. Use

  Flink's windowing capabilities (e.g., TumblingEventTimeWindows, SlidingEventTimeWindows) combined with aggregation functions (AggregateFunction) or ProcessWindowFunction to calculate indicators like moving averages, MACD, or the slope of a regression line on price data within the window.[5] Compare short-term vs. long-term moving averages to determine trend direction.
  - **Volatility (Volatile / Low Volatility):** Calculate statistical measures like the standard deviation or variance of price returns within a time window. Again, Flink's windowing and aggregation functions are required. Compare the calculated volatility against predefined thresholds or historical norms.
  - **Risk-On / Risk-Off:** This is more abstract and harder to quantify reliably. It might involve analyzing correlations between the stock and market indices, sector performance, market breadth indicators (e.g., advance/decline line), or potentially aggregating sentiment across a wide range of assets. Implementation in Flink would be complex, likely requiring multiple input streams and sophisticated state management.
  - **Liquidity Crunch Early Warning:** Could involve analyzing the bid-ask spread, the depth of the order book (requires Dhan's Market Depth feed [33]), or sudden anomalies in trading volume compared to historical averages. Requires access to granular order book data and potentially complex pattern detection logic within Flink.
  - **Stock Mood Index (Extreme Fear to Extreme Greed):** This could be a composite index derived from other calculated metrics. For example, combining aggregated sentiment scores (from step B) with volatility measures (like historical vs. implied volatility, if available). The VIX index is a common market-wide example, but creating a stock-specific equivalent requires careful definition and calibration.
- **Flink Implementation:** Use appropriate Flink operators: keyBy() (to partition streams by stock symbol), window() (to define time boundaries), apply() or process() (to implement custom logic within windows), AggregateFunction (for standard aggregations like average, standard deviation), ProcessFunction (for more complex stateful logic and timer-based operations).
- **Output:** Structure the detected regime information (timestamp, stock symbol, regime type, regime label, potentially a confidence score) and send it to the market_regimes Kafka topic and/or the TimescaleDB sink.

D. Flink Deployment and Monitoring:

- **Packaging:** If using Java/Scala, build the application into a fat JAR containing all dependencies. If using Python, ensure the script and all required libraries (including NLP models if used locally) are accessible within the Flink cluster's Python environment (e.g., by including them in a custom Docker image or mounting them).

- **Job Submission:** Use the Flink command-line interface within the jobmanager container to submit the packaged job to the cluster: flink run [options] <jar-file-or-python-script> [arguments].[12]
- **Monitoring:** Access the Flink Web UI through the port exposed in Docker Compose (e.g., http://localhost:8081).[12] Monitor running jobs, check logs for errors, inspect task managers, and crucially, monitor for signs of backpressure (indicating the pipeline cannot process data as fast as it arrives) and checkpoint failures. These metrics are vital for ensuring the pipeline is stable and performing correctly.

Implementing market regime detection presents a significant challenge compared to standard sentiment analysis. It demands a clear understanding of the underlying financial concepts and how to translate them into stateful stream processing logic using Flink's APIs.[5] Given this complexity, it is advisable to start with simpler, price-based regimes like trend and volatility before attempting more intricate ones like risk-on/off or liquidity analysis, which require more sophisticated data inputs and algorithms.

Furthermore, maintaining data integrity across the pipeline (Kafka -> Flink -> TimescaleDB/Kafka) is paramount, especially if the output influences trading decisions. Rigorously configuring Flink's checkpointing [4] and ensuring the sinks (particularly the JDBC sink to TimescaleDB) are configured for at-least-once or exactly-once semantics (through idempotency or transactions) is essential to prevent data loss or duplication during failures and restarts.

VII. Phase 5: Frontend Development - UI and Integration (Estimated Time: 4-5 Weeks)

This phase involves building the user interface, integrating the charting library, connecting to the backend for data, and establishing the real-time WebSocket communication.

A. UI Layout Implementation:

- **Component Structure:** Break down the UI into reusable React components: a main layout component (DashboardLayout), a chart container (StockChartContainer), a stock list component (Nifty50List), a sentiment/regime display (SentimentPanel), and potentially sub-components for specific elements (e.g., list items, gauge displays).

- **Styling (SCSS):** Use SCSS for styling the components. Organize SCSS files logically (e.g., per component, or using a modular structure like BEM). Implement the TradingView-inspired layout using CSS Grid or Flexbox for the main sections (chart left, list top-right, sentiment bottom-right). Ensure the layout is responsive across different screen sizes using media queries.

B. Lightweight Charts Integration:

- **Installation:** Add the lightweight-charts library to the frontend project: npm install lightweight-charts or yarn add lightweight-charts.[76]
- **Chart Component (StockChart.js):**
  - Create a dedicated component to encapsulate the chart logic.[76]
  - Use the useRef hook to obtain a reference to the DOM element that will contain the chart.[76]
  - Utilize the useEffect hook for setup and teardown:
    - Inside useEffect, call createChart(chartContainerRef.current, chartOptions) to initialize the chart instance when the component mounts. Pass configuration options for appearance (layout colors, background, grid lines), time scale, price scale, etc..[76]
    - Add the required series to the chart instance, for example, chart.addCandlestickSeries() or chart.addLineSeries() for price data, and potentially chart.addHistogramSeries() for volume.[76] Store references to these series.
    - Implement the cleanup function within useEffect to call chart.remove() when the component unmounts. This prevents memory leaks.[76]
- **Data Loading:**
  - **Historical Data:** Implement logic (initially within useEffect, later triggered by stock selection) to fetch historical price data (OHLCV) for the selected stock from a backend API endpoint (to be created or enhanced). Use the series.setData(historicalDataArray) method to populate the chart initially.[76] The data format should match the library's requirements (e.g., { time: unix_timestamp, open:..., high:..., low:..., close:... }).
  - **Real-time Updates:** Prepare the chart component to receive real-time updates (ticks or new bars) via WebSocket messages (handled in step E). Use the series.update(newDataPoint) method to efficiently append or update the latest data point on the chart. Using update is significantly more performant for streaming data than repeatedly calling setData.[76]

C. Nifty 50 List Implementation:

- **Component (Nifty50List.js):** Create a component to render the list.
- **Data Fetching:** Use useEffect to fetch the list of Nifty 50 stocks (symbols, names, potentially initial prices) from the backend API endpoint (/api/nifty50-list) created in Phase 1. Store the list in React state.
- **Rendering:** Map over the stock list data in state and render list items, displaying the stock symbol and possibly the latest price/change (which will be updated via WebSocket).
- **Selection Handling:** Implement an onClick handler for each list item. When a stock is clicked, update a shared state variable (e.g., in a parent component or context) with the selected stock symbol. This selection change should trigger data fetching/updates in the StockChart and SentimentPanel components.

D. Sentiment Display Implementation:

- **Component (SentimentPanel.js):** Create a component for this section.
- **Data Fetching:** When the selected stock symbol changes (monitored via useEffect dependency array), fetch the corresponding sentiment and market regime data from the backend API endpoints (/api/sentiment/{selected_stock} and /api/regime/{selected_stock}). Store this data in local state.
- **Rendering:** Display the fetched sentiment score (e.g., using a text label "Bullish", a numerical score, a simple gauge component, or color coding). Display the fetched market regime labels (e.g., "Trending Up", "Low Volatility"). Optionally, display related news headlines or social media snippets if the backend provides them. This section will also need to be updated by real-time WebSocket messages if the backend pushes live sentiment/regime changes.

E. WebSocket Client Integration:

- **Connection Management:** Establish the WebSocket connection to the backend endpoint (ws://<backend_host>:<port>/ws/stocks) when the main application component mounts. The native browser WebSocket API is sufficient [69], although libraries or custom hooks (useWebSocket) can simplify state management and reconnection logic. Store the WebSocket instance reference (e.g., using useRef or state).

  JavaScript
  ```javascript
  // Example using native WebSocket API in a React component
  const socket = useRef(null);
  useEffect(() => {
   socket.current = new WebSocket('ws://localhost:8000/ws/stocks'); // Replace with actual backend URL
  ```

```
 socket.current.onopen = () => console.log('WebSocket Connected');
 socket.current.onclose = () => console.log('WebSocket Disconnected');
 socket.current.onerror = (error) => console.error('WebSocket Error:', error);

 // Message handler defined below
 socket.current.onmessage = handleWebSocketMessage;

 // Cleanup on unmount
 return () => {
  socket.current.close();
 };
},); // Empty dependency array ensures this runs once on mount
```

- **Message Handling (onmessage):** Implement the onmessage event handler.[69] This function will receive data pushed from the backend.
  1. Parse the incoming message event data (event.data), which should be JSON strings sent by the backend.
  2. Determine the type of update based on the message structure (e.g., Nifty 50 tick, selected stock quote update, sentiment update, regime update).
  3. Update the relevant React state based on the received data. This state update will propagate through the component tree and trigger re-renders.

JavaScript
```
const handleWebSocketMessage = (event) => {
 try {
  const message = JSON.parse(event.data);
  // Example: Update chart if it's a price update for the selected stock
  if (message.type === 'stock_update' && message.symbol === selectedStock) {
   // Assuming chartSeriesRef holds the Lightweight Charts series instance
   chartSeriesRef.current?.update({
    time: message.timestamp,
    open: message.open, // Or just price if it's a tick
    high: message.high,
    low: message.low,
    close: message.close,
    // volume: message.volume // if available
   });
  }
  // Example: Update Nifty 50 list state
  if (message.type === 'nifty50_tick') {
   setNifty50Data(prevData => ({
    ...prevData,
      [message.symbol]: { price: message.ltp, change: message.change }
    }));
  }
  // Example: Update sentiment state
  if (message.type === 'sentiment_update' && message.symbol === selectedStock) {
    setSentimentData({ score: message.score, label: message.label });
  }
  //... handle other message types (regime, etc.)
} catch (error) {
  console.error('Failed to parse WebSocket message:', error);
}
};
```

- **State Management:** For managing the WebSocket connection instance and distributing the received real-time data across multiple components (Chart, List, Sentiment), consider using React Context API or a dedicated state management library (like Zustand, Redux, or Jotai). This avoids prop drilling and provides a cleaner way for components to subscribe to the live data they need. Create a context provider near the top of the application tree that manages the WebSocket connection and exposes the latest received data and connection status. Components can then consume this context to access the data.
- **Disconnection/Error Handling:** Implement logic in the onclose and onerror handlers.[69] Attempt reconnection after a delay if the connection closes unexpectedly. Provide visual feedback to the user if the real-time connection is lost.

Updating the Lightweight Charts instance efficiently is crucial for a smooth user experience. Using series.update() [76] for each incoming tick or bar is the recommended approach for real-time data streams, preventing performance degradation associated with replacing the entire dataset frequently. Managing the shared WebSocket connection and the flow of real-time data updates across different parts of the React application is a key architectural consideration. Employing a state management solution or React Context can significantly simplify this process, making the application more maintainable and scalable.

VIII. Phase 6: Integration, Testing, and Deployment (Estimated Time: 3-4 Weeks + Ongoing)

This final active development phase involves connecting all the pieces, establishing robust testing practices, and deploying the application. Testing and deployment are ongoing activities throughout the project lifecycle.

A. Component Integration:

- **Flink Pipeline to Backend:** Determine how the processed data (sentiment, market regimes) from Flink reaches the backend API to be served to the frontend. Two primary approaches exist:
    1. **Database-centric:** Flink jobs sink the processed stock_sentiment and market_regimes data directly into the TimescaleDB hypertables (using the JDBC Sink configured in Phase 4).[10] The backend API endpoints (/api/sentiment/{symbol}, /api/regime/{symbol}) then query TimescaleDB to retrieve the latest data for the requested stock symbol when called by the frontend. This is simpler for request-response patterns but might not be fully real-time unless the backend also polls the DB or uses database triggers (which adds complexity).
    2. **Kafka-centric:** Flink jobs publish results to the dedicated Kafka topics (processed_sentiment, market_regimes). The backend service then needs to run Kafka consumers (using a library like kafka-python integrated within FastAPI, possibly in background tasks) to listen to these topics. Upon receiving a new message (e.g., updated sentiment for AAPL), the backend can proactively push this update via its WebSocket connection (/ws/stocks) to relevant frontend clients. This enables more immediate real-time updates for sentiment/regimes on the frontend but adds Kafka consumer logic to the backend service.
    - A hybrid approach is also possible (Flink writes to both DB and Kafka). The choice depends on the desired real-time granularity for sentiment/regime updates versus implementation complexity.
- **Frontend to Backend:** Replace all mock data fetching and WebSocket interactions in the React frontend with calls to the actual backend API endpoints and WebSocket connection established in previous phases. Ensure data formats align between frontend expectations and backend responses/pushes.

B. Testing Methodologies:

A comprehensive testing strategy is essential for ensuring the quality and reliability of this complex, distributed system.

- **Unit Testing:** Focus on testing individual functions and components in isolation.
    - **Frontend (React):** Use Jest and React Testing Library (or Enzyme) to test React components. Mock API calls (fetch, axios), WebSocket interactions (mock WebSocket object or use testing utilities), and props to verify

component rendering and behavior based on different inputs and states.
    - **Backend (FastAPI):** Use pytest and FastAPI's TestClient to test API endpoint handlers. Test service layer functions, data transformation logic, and the parsing logic for Dhan's binary WebSocket messages independently. Mock external dependencies like the Dhan API client, Kafka producers/consumers, and database connections using libraries like pytest-mock or unittest.mock.
    - **Data Pipeline (Flink):** Flink provides utilities for testing individual operators (like MapFunction, ProcessFunction, WindowFunction). Test the logic within these operators with sample input data. Test Kafka producer and consumer scripts separately to ensure they correctly serialize/deserialize messages and handle connection errors. Test NLP functions for sentiment analysis with sample text inputs.
- **Integration Testing:** Verify the interactions *between* different components of the system.
    - **Backend <-> TimescaleDB:** Write tests that start the backend service and a test database (e.g., using testcontainers or a dedicated test DB instance), make API calls that trigger database operations, and assert the database state is correct.
    - **Backend <-> Kafka:** Test that the backend can correctly produce messages to Kafka topics and/or consume messages from topics (depending on the integration strategy chosen in VIII.A). Requires a running Kafka instance for the test environment.
    - **Flink <-> Kafka:** Run Flink jobs in a test environment connected to a test Kafka instance. Produce sample messages to input topics and verify that the job consumes them and produces the expected output messages to sink topics.
    - **Flink <-> TimescaleDB:** Test the Flink JDBC sink by running a job that processes sample data and verifies that the correct records are inserted into the TimescaleDB hypertable.
    - **Frontend <-> Backend:** Use end-to-end testing tools like Cypress or Playwright, but focus on specific interactions. Start the backend server (potentially with mocked external dependencies). Have the frontend test suite make API calls to the running backend and establish WebSocket connections. Assert that the frontend receives the expected data and updates its state/UI correctly based on API responses and WebSocket messages.
- **End-to-End (E2E) Testing:** Simulate complete user workflows across the entire deployed stack.
    - **Scenario Definition:** Define realistic user scenarios, e.g., "User loads dashboard, selects INFY from Nifty 50 list, chart displays historical data, live

    ticks start updating the chart via WebSocket, a news event triggers a sentiment change in Flink, the sentiment panel on the frontend updates via WebSocket push from the backend."
  - **Environment:** Requires a fully integrated environment, ideally mirroring production. Docker Compose can be used to spin up the entire stack (Frontend, Backend, Kafka, Flink, TimescaleDB) for E2E tests.
  - **Tooling:** Use browser automation tools like Cypress or Playwright to drive the frontend UI.
  - **Challenges & Strategies:** Testing asynchronous, real-time flows is complex. Tests need robust waiting mechanisms (e.g., wait for a specific WebSocket message, wait for a UI element to update, wait for a database record) rather than fixed time delays (sleep). Assertions may need to span across different system boundaries (e.g., check UI state, query backend API, inspect database record, potentially check Kafka topic messages). This might require custom commands or helper functions within the E2E testing framework. Simulating external events (like Dhan ticks or news arrival) might involve injecting messages directly into Kafka topics or mocking the producer components during the test run.

Validating the asynchronous data flows inherent in this real-time system is a primary challenge for E2E testing. Tests must be carefully designed to handle the non-deterministic timing of events propagating through Kafka, Flink, the backend, and WebSockets to the frontend. Asserting data consistency across the UI, backend state, and database after an event requires a well-orchestrated test setup capable of interacting with and observing multiple parts of the distributed system.

C. Deployment Strategies:

- **Containerization:** Package each service into a Docker image for consistent deployment.
  - **Frontend:** Use a multi-stage Dockerfile. Stage 1 builds the React application (npm run build). Stage 2 copies the static build artifacts into a lightweight web server image like Nginx to serve the frontend.
  - **Backend:** Create a Dockerfile for the FastAPI (or Spring Boot) application, installing dependencies and copying the application code. Ensure the container starts the ASGI server (Uvicorn) or the Java application.
  - **Producers:** If the Python data producers are complex, package them into their own Docker images as well.
- **Orchestration:**
  - **Docker Compose:** Suitable for single-server deployments or

    development/testing environments. The docker-compose.yml file defines how to run all the containers together.
  - **Kubernetes (K8s):** The standard for scalable, resilient deployments in cloud environments. Requires defining K8s deployment configurations (Deployments, Services, StatefulSets for Kafka/DB, ConfigMaps, Secrets). Offers auto-scaling, rolling updates, and self-healing capabilities.
- **Cloud Options:** Leverage cloud platforms for hosting and managed services:
  - **Compute:** Deploy containers to AWS ECS/EKS, Google Cloud Run/GKE, Azure Container Apps/AKS.
  - **Managed Services:** Consider using managed cloud services to reduce operational burden, although this typically increases cost:
    - Kafka: AWS MSK, Confluent Cloud, Google Cloud Pub/Sub (as alternative), Azure Event Hubs.
    - Flink: AWS Managed Service for Apache Flink [5], Google Cloud Dataflow, Azure Stream Analytics.
    - PostgreSQL/TimescaleDB: AWS RDS with TimescaleDB extension, Timescale Cloud, Google Cloud SQL, Azure Database for PostgreSQL.
- **CI/CD Pipeline:** Implement automated pipelines using tools like GitHub Actions, GitLab CI, or Jenkins.
  - **Continuous Integration (CI):** On every code push/merge, automatically build Docker images, run unit tests, and integration tests.
  - **Continuous Deployment (CD):** Automatically deploy successfully tested builds to staging or production environments (using Docker Compose commands, kubectl apply, or cloud provider deployment tools).

IX. Learning Resources and Upskilling

Acquiring proficiency in the diverse technologies used in this project requires dedicated learning. Below are curated resources based on the research:

A. Real-time Data Pipelines (Kafka, Flink):

- **Kafka:**
  - Official Apache Kafka Documentation: The definitive source for concepts, configuration, and APIs.
  - Confluent Developer Website: Numerous tutorials, blog posts, and courses on Kafka and its ecosystem.
- **Flink:**
  - Official Apache Flink Documentation: Essential reading, especially the DataStream API guide and connector documentation (Kafka, JDBC).[77]

    - Flink Tutorials & Examples: Explore official examples and community tutorials demonstrating stream processing patterns.[4]
- **Integrated Pipelines:**
    - Blog Posts & Articles: Search for specific tutorials on "Kafka Flink TimescaleDB", "real-time analytics pipeline Kafka Flink".[5] Pay attention to articles from data platform vendors (Confluent, Aiven, Timescale) or reputable tech blogs.

B. Backend Development (FastAPI / Java Spring Boot):

- **FastAPI:**
    - Official FastAPI Documentation: The Tutorial - User Guide is excellent for beginners, followed by the Advanced User Guide.[1] Covers core concepts, async, WebSockets, dependency injection, etc.
    - Tutorials: Visual Studio Code tutorial [22], Kinsta guide [25], Teclado course [23], TestDriven.io blogs [81], and video tutorials [82] provide practical examples.
- **Java Spring Boot:**
    - Official Spring Boot Documentation: Comprehensive guides on core concepts, web development, and WebSocket support (specifically the guide on STOMP over WebSocket).[26]
    - Tutorials: Baeldung [29], Devglan [28], Piehost [84], Vonage [27] offer step-by-step guides for setting up Spring Boot WebSockets.[85]

C. WebSockets Integration:

- **Framework Docs:** Refer to the WebSocket sections within the FastAPI [2] and Spring Boot [26] documentation.
- **Client-Side:** MDN WebSockets API documentation for understanding the browser's native WebSocket interface.
- **Examples:** Look at GitHub repositories and blog posts demonstrating WebSocket chat applications or real-time data examples using the chosen backend framework.[2]

D. Sentiment Analysis & NLP for Finance:

- **Python Libraries:**
    - NLTK: Focus on the VADER sentiment analyzer.[74]
    - TextBlob: Explore its sentiment analysis capabilities.[74]
    - spaCy: A powerful library for general NLP tasks, can be used with sentiment models.[74]
    - Hugging Face Transformers: Provides access to state-of-the-art pre-trained models, including sentiment analysis models, but requires more setup.[74]

- **Tutorials:** Search for "financial sentiment analysis python tutorial", "VADER sentiment analysis python", "TextBlob sentiment analysis".[70] Look for examples using financial news or social media data.
- **Advanced:** LangChain documentation [75] if considering integrating Large Language Models (LLMs) for more nuanced analysis.

E. Stock Market Regime Detection:

- **Specialized Resources:** This topic requires delving into quantitative finance resources beyond typical software engineering tutorials.
  - Search academic databases (Google Scholar, arXiv) for papers on "market regime detection", "financial time series analysis", "Hidden Markov Models in finance".
  - Explore quantitative finance blogs, forums (e.g., QuantStackExchange, Nuclear Phynance), and books on algorithmic trading and financial modeling.
  - Look for Python implementations using libraries like pandas, numpy, statsmodels, ta-lib (for technical indicators), and potentially hmmlearn (for Hidden Markov Models). Translating these algorithms to Flink's distributed stream processing paradigm will be the main challenge.

A structured learning approach is recommended. Start with the official documentation for each core technology (FastAPI/Spring, Kafka, Flink, TimescaleDB, React) to build a solid foundation. Supplement this with high-quality, step-by-step tutorials from reputable sources. Use GitHub examples and blog posts for practical implementation patterns, but critically assess their relevance, quality, and whether they follow best practices. For the specialized domain of market regime detection, recognize that standard software development resources will be insufficient; dedicated study of financial modeling techniques and literature is necessary.

X. Alternative Technologies and Data Sources

While the primary technology stack and data sources have been defined, exploring alternatives can be valuable for comparison, cost optimization, or addressing potential limitations.

A. Real-time Stock Data APIs:

If the Dhan API proves unsuitable due to cost, data limitations, or reliability concerns, several alternatives exist, particularly within the Indian market context, as well as global providers:

- **Indian Brokers:**

  - **Zerodha Kite Connect:** Very popular in India, offers REST APIs and WebSocket streaming (KiteTicker).[96] However, it has a significant cost (₹2000/month).[97] Python client library available (pykiteconnect).[96]
  - **Upstox API:** Another major Indian broker offering APIs.[99] Check their documentation for features, pricing, and real-time capabilities.
  - **Fyers API:** Provides APIs for trading and data.[99] Known for being relatively developer-friendly. Investigate their WebSocket feed and pricing.
  - **Angel One, 5paisa, Alice Blue, etc.:** Other brokers also offer APIs, but documentation quality, features, and reliability can vary.
- **Global Platforms/Vendors:**
  - **Alpaca Markets:** Offers commission-free trading APIs (primarily US stocks) and market data APIs. Provides free and paid tiers for market data, including real-time WebSocket streaming for US equities and crypto.[51] Well-documented and popular among algo traders.
  - **Interactive Brokers (IBKR):** Provides a comprehensive API (TWS API or Client Portal Web API) but is known for its complexity. Requires a funded account. Offers global market data but can be challenging to integrate.
  - **Polygon.io:** A dedicated financial data provider offering real-time and historical data for stocks, options, forex, crypto via REST and WebSockets. Generally considered high-quality but comes at a higher cost than broker APIs.
  - **IEX Cloud:** Provides financial data APIs with various pricing tiers. Known for ease of use. Check their real-time data offerings and limitations.
  - **Twelve Data:** Another market data vendor offering real-time WebSockets and REST APIs across various asset classes.

Table 2: Stock Data API Alternatives Comparison

Provider	Key Features	Pricing Model (Monthly Est.)	Key Rate Limits (Example)	Real-time Method	Pros	Cons
Dhan	Indian Equities, F&O, MCX; WebSocket/REST	Trading: Free; Data: ₹499+tax (or free) [30]	Liberal (e.g., 25 Order req/sec) [37]	WebSocket (Binary)	Free trading API, good limits, Indian focus	Data API cost (unless active trader), Binary WS requires

						parsing
Zerodha Kite	Indian Equities, F&O, MCX; WebSocket/REST	~₹2000 [98]	Moderate (check docs)	WebSocket (JSON?)	Popular in India, extensive community support	Significant monthly cost
Alpaca Markets	US Equities, Crypto; WebSocket/REST	Free Tier (limited); Paid tiers ($ varies)	High limits on paid tiers [51]	WebSocket (JSON)	Free tier available, good docs, popular for US stocks	Primarily US market focus
Interactive Brokers	Global Equities, Options, Futures, FX; API	Requires funded account; Market data fees vary	Complex limits (check docs)	Streaming API	Very comprehensive data, global access	High complexity, steep learning curve
Polygon.io	US/Global Stocks, Options, FX, Crypto; WS/REST	Paid tiers ($–$$) \$$	High limits on paid tiers \	WebSocket (JSON) \	High-quality data, extensive coverage \	Expensive compared to broker APIs \
\	**IEX Cloud** \	US Equities; REST \	Free tier (limited); Paid tiers (-$$$)	Message-based quotas	SSE/REST (Polling)	Easy to use, good documentation

B. News/Sentiment Data Sources:

Beyond the primary News and Social Media APIs:

- **StockTwits API:** This platform is a social network specifically for investors and traders. An official API might exist (requires investigation via their developer resources) that could provide highly relevant, finance-focused social sentiment.
- **RSS Feeds:** Many financial news outlets (Bloomberg, Reuters, Economic Times, Moneycontrol, etc.) offer RSS feeds. These can be polled periodically using

standard Python libraries (feedparser, requests). While often free, the data is less structured than APIs, requires parsing HTML/XML content, and updates might not be truly real-time.

- **Financial News APIs:** Revisit the alternatives discussed in Section II.B, such as Financelayer [48], Real-Time Finance Data on RapidAPI [49], EODHD [50], Alpaca News API [51], and Finnhub [52], comparing their costs, data relevance, and real-time capabilities.
- **Alternative Social Platforms:** Monitoring specialized forums (like WallStreetBets on Reddit - though API access rules apply), Discord servers, or other niche platforms might yield insights, but reliable, public APIs are rare, and scraping is often against terms of service and unreliable.

C. Open Financial Datasets:

For backtesting strategies, training models, or enriching the real-time analysis with historical context:

- **Nasdaq Data Link (formerly Quandl):** A large repository of financial, economic, and alternative datasets. Offers both free and premium datasets covering historical prices, fundamentals, economic indicators, etc.
- **Yahoo Finance (via libraries):** Libraries like yfinance in Python provide a convenient way to download free historical stock price data (OHLCV), dividends, splits, basic fundamentals, and options data directly from Yahoo Finance. While useful for historical analysis and backtesting, it's not suitable for reliable real-time streaming due to potential delays and lack of official API support/rate limits.
- **SEC EDGAR Database:** The official source for US public company filings (Annual Reports 10-K, Quarterly Reports 10-Q, etc.). Provides raw text data that can be programmatically accessed and parsed for fundamental analysis or extracting textual data for long-term sentiment analysis (requires significant NLP effort).
- **Kaggle Datasets:** A platform hosting numerous user-contributed datasets. Search for "financial news sentiment" [75], "stock market historical data", "economic indicators", etc. Useful for finding pre-processed datasets for model training or exploring historical trends.

XI. Open Source References and Further Reading

Leveraging existing open-source projects and technical articles can accelerate development and provide valuable implementation patterns.

A. Relevant GitHub Repositories:

- **Financial Dashboards:** Search GitHub for terms like:

    - react financial dashboard [100]
    - tradingview clone react
    - lightweight charts react example [76]
    - Look for projects using similar charting libraries or UI structures. Be mindful that many complex dashboards might use different backend/data pipeline technologies.
- **FastAPI WebSocket Examples:** Find examples demonstrating:
    - fastapi websocket chat
    - fastapi websocket broadcast [68]
    - fastapi websocket real-time data [65]
    - Pay attention to connection management and message handling patterns.
- **Java Spring Boot WebSocket Examples:** Search for:
    - spring boot websocket chat [92]
    - spring boot websocket stomp example [93]
    - spring boot websocket stock ticker [91]
    - Look for examples using STOMP and SockJS if those features are desired.
- **Kafka/Flink Pipelines:** Explore repositories demonstrating:
    - kafka flink pipeline example [72]
    - flink kafka consumer producer example [16]
    - flink sentiment analysis kafka [71]
    - flink jdbc sink postgresql or flink timescale sink [12]
    - Examples involving financial data processing are particularly relevant.[4]
- **Trading Bots/API Clients:**
    - Official Broker Clients: Look for official libraries like DhanHQ-py [67] or pykiteconnect.[96]
    - Community Clients/Frameworks: Search for specific broker names (e.g., "Dhan API python") or general terms like "python trading bot".[97] Evaluate the quality, maintenance status, and features of community projects carefully.

B. Insightful Technical Blog Posts:

- Search for specific implementation details and tutorials:
    - FastAPI WebSocket real-time data tutorial [2]
    - Kafka Flink TimescaleDB tutorial or real-time analytics pipeline Kafka Flink [6]
    - financial sentiment analysis python tutorial [70]
    - integrate Lightweight Charts React [76]
- Prioritize content from official sources (e.g., FastAPI/Tiangolo, Apache Flink project, Confluent, Timescale) and reputable technical blogs known for high-quality content (e.g., Baeldung for Java/Spring, TestDriven.io for Python/Web). Articles detailing specific challenges, like Flink offset management [4]

or integrating AI models [71], can be particularly valuable.

XII. Conclusion

Summary: This report has outlined a comprehensive, phased roadmap for developing a real-time financial analytics dashboard. The proposed architecture leverages React/SCSS/jQuery for the frontend, FastAPI (or Spring Boot) with WebSockets for the backend, and a robust Kafka/Flink/TimescaleDB pipeline for data ingestion and processing. The plan covers environment setup, API investigation, incremental development of backend, data pipeline, and frontend components, integration, testing strategies, deployment considerations, learning resources, and potential alternatives.

Key Challenges: Successfully completing this project requires navigating several significant challenges:

1. **API Costs and Limitations:** Accessing real-time, high-volume data from social media (Twitter, Reddit) and potentially news sources necessitates paid API subscriptions, representing a considerable ongoing cost and dependency. Free tiers are generally inadequate.
2. **Real-time System Complexity:** Building and managing a distributed system involving WebSockets, Kafka, and Flink introduces complexities in ensuring data consistency, handling failures gracefully (reconnection, checkpointing [4]), managing state, and debugging issues across components. The binary nature of the Dhan WebSocket feed adds specific parsing challenges.[33]
3. **Market Regime Logic:** Implementing meaningful market regime detection requires significant financial domain expertise and advanced stream processing techniques within Flink.[5] This goes beyond standard NLP or web development tasks.
4. **End-to-End Testing:** Verifying the correctness of asynchronous, real-time data flows across the entire stack demands sophisticated E2E testing strategies capable of handling timing variations and asserting state across multiple system boundaries.

Next Steps: The recommended path forward involves starting with Phase 0: meticulously setting up the development environment using Docker Compose and thoroughly investigating the costs, rate limits, and technical requirements of all external APIs. This initial investigation is crucial for confirming feasibility within budget and potentially adjusting the project scope (especially regarding real-time social media sentiment) before significant development effort is invested. Proceed incrementally through the subsequent phases, prioritizing the core stock data flow (Dhan -> Backend -> Frontend) before tackling the more complex data pipeline elements. Continuous testing at each stage (unit, integration) is vital for managing complexity and ensuring reliability.

Final Encouragement: Building this real-time financial analytics dashboard is an ambitious undertaking that integrates multiple advanced technologies. While challenging, successfully implementing this system offers a powerful platform for market analysis and represents a significant technical achievement. By following a structured, phased approach, leveraging the provided resources, and addressing the inherent complexities proactively, the project goals are achievable.

Works cited

1. First Steps - FastAPI, accessed April 11, 2025, https://fastapi.tiangolo.com/tutorial/first-steps/
2. Fastapi Real-Time Data Processing Examples | Restackio, accessed April 11, 2025, https://www.restack.io/p/real-time-ai-inference-answer-fastapi-examples-cat-ai
3. ci-compass/kafka-flink-cluster: A simple docker-compose cluster for local development - GitHub, accessed April 11, 2025, https://github.com/ci-compass/kafka-flink-cluster
4. Challenges and Solutions for Flink Offset Management During Kafka Cluster Migration, accessed April 11, 2025, https://github.com/AutoMQ/automq/wiki/Challenges-and-Solutions-for-Flink-Offset-Management-During--Kafka-Cluster-Migration
5. Publish and enrich real-time financial data feeds using Amazon MSK and Amazon Managed Service for Apache Flink | AWS Big Data Blog, accessed April 11, 2025, https://aws.amazon.com/blogs/big-data/publish-and-enrich-real-time-financial-data-feeds-using-amazon-msk-and-amazon-managed-service-for-apache-flink/
6. Green Data, Clean Insights: How Kafka and Flink Power ESG Transformations, accessed April 11, 2025, https://www.kai-waehner.de/blog/2024/02/10/green-data-clean-insights-how-kafka-and-flink-power-esg-transformations/
7. Install TimescaleDB on Docker - Timescale documentation, accessed April 11, 2025, https://docs.timescale.com/self-hosted/latest/install/installation-docker/
8. Build a time series data stream with Redpanda and TimescaleDB, accessed April 11, 2025, https://www.redpanda.com/blog/build-data-stream-detect-anomalies-timescale-kafka-connect
9. Introduction to TimescaleDB - ConSol Labs, accessed April 11, 2025, https://labs.consol.de/development/2018/10/31/introduction-to-timescale-db.html
10. Real-Time Pipeline Using Kafka, Flink, TimescaleDB and Streamlit for Monitorig Autonomous Vehicle | by Lokesh Reddy Lingala | Medium, accessed April 11, 2025, https://medium.com/@lokeshreddy.lingala18/real-time-pipeline-using-kafka-flink-timescaledb-and-streamlit-for-monitorig-autonomous-vehicle-9c80169e26fe

11. Debezium and TimescaleDB, accessed April 11, 2025, https://debezium.io/blog/2024/01/11/Debezium-and-TimescaleDB/
12. How I Dockerized Apache Flink, Kafka, and PostgreSQL for Real-Time Data Streaming, accessed April 11, 2025, https://medium.com/towards-data-science/how-i-dockerized-apache-flink-kafka-and-postgresql-for-real-time-data-streaming-c4ce38598336
13. Build a data pipeline with Apache Kafka and TimescaleDB | Timescale, accessed April 11, 2025, https://www.timescale.com/blog/create-a-data-pipeline-with-timescaledb-and-kafka
14. Real-time insights: Telemetry Pipeline - PYBLOG, accessed April 11, 2025, https://www.pyblog.xyz/telemetry-pipeline
15. rgupta3349/flink-kafka-docker-compose - GitHub, accessed April 11, 2025, https://github.com/rgupta3349/flink-kafka-docker-compose
16. Flink (on docker) to consume data from Kafka (on docker) - Stack Overflow, accessed April 11, 2025, https://stackoverflow.com/questions/70085088/flink-on-docker-to-consume-data-from-kafka-on-docker
17. Adding Bootstrap - Create React App, accessed April 11, 2025, https://create-react-app.dev/docs/adding-bootstrap/
18. How to add SCSS styles to a React project? - Stack Overflow, accessed April 11, 2025, https://stackoverflow.com/questions/67352418/how-to-add-scss-styles-to-a-react-project
19. how to include sass file in reactjs - Stack Overflow, accessed April 11, 2025, https://stackoverflow.com/questions/52406396/how-to-include-sass-file-in-reactjs
20. Adding a Sass Stylesheet - Create React App, accessed April 11, 2025, https://create-react-app.dev/docs/adding-a-sass-stylesheet/
21. Getting Started with Bootstrap 5, React, and Sass - Designmodo, accessed April 11, 2025, https://designmodo.com/bootstrap-react-sass/
22. FastAPI Tutorial in Visual Studio Code, accessed April 11, 2025, https://code.visualstudio.com/docs/python/tutorial-fastapi
23. How to set up a FastAPI Project - Teclado, accessed April 11, 2025, https://teclado.com/fastapi-for-beginners/how-to-set-up-fastapi-project/
24. Tutorial - User Guide - FastAPI, accessed April 11, 2025, https://fastapi.tiangolo.com/tutorial/
25. Build an App With FastAPI for Python - Kinsta®, accessed April 11, 2025, https://kinsta.com/blog/fastapi/
26. Getting Started | Using WebSocket to build an interactive web application - Spring, accessed April 11, 2025, https://spring.io/guides/gs/messaging-stomp-websocket/
27. Creating a WebSocket Server with Spring Boot - Vonage, accessed April 11, 2025, https://developer.vonage.com/ja/blog/create-websocket-server-spring-boot-dr
28. Spring Boot Websocket Example | DevGlan, accessed April 11, 2025,

https://www.devglan.com/spring-boot/spring-boot-websocket-example
29. Intro to WebSockets with Spring - Baeldung, accessed April 11, 2025, https://www.baeldung.com/websockets-spring
30. Start Trading & Investing with Dhan APIs: DhanHQ, accessed April 11, 2025, https://dhanhq.co/
31. Is Dhan API (Algo Trading) Review - Chittorgarh, accessed April 11, 2025, https://www.chittorgarh.com/broker/dhan/api-for-algo-trading-review/176/
32. API : . - Dhan, accessed April 11, 2025, https://knowledge.dhan.co/support/solutions/folders/82000695180
33. Live Market Feed - DhanHQ Ver 1.0 / API Document, accessed April 11, 2025, https://dhanhq.co/docs/v1/live-market-feed/
34. Live Market Feed - DhanHQ Ver 2.0 / API Document, accessed April 11, 2025, https://dhanhq.co/docs/v2/live-market-feed/
35. Get Nifty 50 Price using Dhan API - MadeForTrade, accessed April 11, 2025, https://madefortrade.in/t/get-nifty-50-price-using-dhan-api/23462
36. Market Quote - DhanHQ Ver 2.0 / API Document, accessed April 11, 2025, https://dhanhq.co/docs/v2/market-quote/
37. Introduction - DhanHQ Ver 2.0 / API Document, accessed April 11, 2025, https://dhanhq.co/docs/v2/
38. How many maximum orders can be placed using Dhan API? : ., accessed April 11, 2025, https://knowledge.dhan.co/support/solutions/articles/82000891156-how-many-maximum-orders-can-be-placed-using-dhan-api-
39. (Resolved) What's wrong with the API rate limits? - MadeForTrade, accessed April 11, 2025, https://madefortrade.in/t/resolved-whats-wrong-with-the-api-rate-limits/19828
40. What are the API limits per second in Dhan ? : ., accessed April 11, 2025, https://knowledge.dhan.co/support/solutions/articles/82000891163-what-are-the-api-limits-per-second-in-dhan-
41. Update on APIs: Now Unlimited Rate Limits on DhanHQ APIs - MadeForTrade, accessed April 11, 2025, https://madefortrade.in/t/update-on-apis-now-unlimited-rate-limits-on-dhanhq-apis/21353
42. Twitter API pricing, limits: detailed overlook | Data365.co, accessed April 11, 2025, https://data365.co/guides/twitter-api-limitations-and-pricing
43. X API Rate Limits (Formerly Twitter) - 9meters, accessed April 11, 2025, https://9meters.com/entertainment/social-media/x-api-rate-limits-formerly-twitter
44. Twitter API Essential Guide - Rollout, accessed April 11, 2025, https://rollout.com/integration-guides/twitter/api-essentials
45. In-Depth Look at Twitter API Documentation - CoreCommerce, accessed April 11, 2025, https://corecommerce.com/blog/in-depth-look-at-twitter-api-documentation/
46. Reddit API: Features, Pricing & Set-ups - Apidog, accessed April 11, 2025, https://apidog.com/blog/reddit-api-guide/

47. Pricing - News API, accessed April 11, 2025, https://newsapi.org/pricing
48. Finance News API - APILayer, accessed April 11, 2025, https://apilayer.com/marketplace/financelayer-api
49. Real-Time Finance Data - Rapid API, accessed April 11, 2025, https://rapidapi.com/letscrape-6bRBa3QguO5/api/real-time-finance-data
50. API Limits: calls, requests, consumption | EODHD APIs Documentation - EOD Historical Data, accessed April 11, 2025, https://eodhd.com/financial-apis/api-limits
51. Introducing News API for Real-Time Stock & Crypto Stories - Alpaca, accessed April 11, 2025, https://alpaca.markets/blog/introducing-news-api-for-real-time-fiancial-news/
52. Real-time Market News API - Finnhub, accessed April 11, 2025, https://finnhub.io/docs/api/market-news
53. API Documentation | Finnhub - Free APIs for realtime stock, forex, and cryptocurrency. Company fundamentals, economic data, and alternative data., accessed April 11, 2025, https://finnhub.io/docs/api
54. Reddit Data API Wiki - Reddit Help, accessed April 11, 2025, https://support.reddithelp.com/hc/en-us/articles/16160319875092-Reddit-Data-API-Wiki
55. Dive Into The Reddit API: Full Guide and Controversy | Zuplo Blog, accessed April 11, 2025, https://zuplo.com/blog/2024/10/01/reddit-api-guide
56. API Update: Enterprise Level Tier for Large Scale Applications - Reddit, accessed April 11, 2025, https://www.reddit.com/r/redditdev/comments/13wsiks/api_update_enterprise_level_tier_for_large_scale/
57. Reddit API Essential Guide - Rollout, accessed April 11, 2025, https://rollout.com/integration-guides/reddit/api-essentials
58. Different ratelimits : r/redditdev, accessed April 11, 2025, https://www.reddit.com/r/redditdev/comments/15gwi25/different_ratelimits/
59. Confused on How to Use Pushshift - Reddit, accessed April 11, 2025, https://www.reddit.com/r/pushshift/comments/1c2ndiu/confused_on_how_to_use_pushshift/
60. Not able to retrieve Reddit submissions and comments with Pushift API as before, accessed April 11, 2025, https://www.reddit.com/r/pushshift/comments/148fv2n/not_able_to_retrieve_reddit_submissions_and/
61. Pushshift, accessed April 11, 2025, https://pushshift.io/
62. Pushshift Live Again and How Moderators Can Request Pushshift Access - Reddit, accessed April 11, 2025, https://www.reddit.com/r/pushshift/comments/14ei799/pushshift_live_again_and_how_moderators_can/
63. Reddit Data API Update: Changes to Pushshift Access : r/modnews, accessed April 11, 2025, https://www.reddit.com/r/modnews/comments/134tjpe/reddit_data_api_update_changes_to_pushshift_access/

64. PullPush Reddit API, accessed April 11, 2025, https://pullpush.io/
65. Build dynamic, secure APIs with FastAPI: Features DB integration, real-time WebSocket, streaming, and efficient request handling with middleware, powered by Starlette and Pydantic. - GitHub, accessed April 11, 2025, https://github.com/zhiyuan8/FastAPI-websocket-tutorial
66. Project Setup - JetBrains Guide, accessed April 11, 2025, https://www.jetbrains.com/guide/python/tutorials/fastapi-aws-kubernetes/project_setup/
67. dhan-oss - GitHub, accessed April 11, 2025, https://github.com/dhan-oss
68. FastAPI and WebSockets: Comprehensive Tutorial - Orchestra, accessed April 11, 2025, https://www.getorchestra.io/guides/fastapi-and-websockets-comprehensive-tutorial
69. FastAPI and WebSockets: A Comprehensive Guide - Orchestra, accessed April 11, 2025, https://www.getorchestra.io/guides/fastapi-and-websockets-a-comprehensive-guide
70. Sentiment Analysis Using Python | NewsCatcher, accessed April 11, 2025, https://www.newscatcherapi.com/blog/sentiment-analysis-using-python
71. Building a real-time AI pipeline for data analysis with Apache Flink® and OpenAI - Aiven, accessed April 11, 2025, https://aiven.io/developer/building-a-real-time-pipeline-for-data-analysis-with-gpt-models
72. NashTech-Labs/kafka-flink-data-pipeline.g8 - GitHub, accessed April 11, 2025, https://github.com/NashTech-Labs/kafka-flink-data-pipeline.g8
73. FA3001/E-Commerce-Analytics-With-Apache-Flink - GitHub, accessed April 11, 2025, https://github.com/FA3001/E-Commerce-Analytics-With-Apache-Flink
74. Introduction to Sentiment Analysis in Python | The PyCharm Blog, accessed April 11, 2025, https://blog.jetbrains.com/pycharm/2024/12/introduction-to-sentiment-analysis-in-python/
75. Sentiment analysis of Financial News using LangChain | by Patrick Gomes - Medium, accessed April 11, 2025, https://patotricks15.medium.com/sentiment-analysis-of-financial-news-using-langchain-43b39eb401a7
76. How to Integrate TradingView's Lightweight Charts in a React Application - Stackademic, accessed April 11, 2025, https://blog.stackademic.com/how-to-integrate-tradingviews-lightweight-charts-in-a-react-application-94e0dbd0657d
77. apache/flink-connector-kafka - GitHub, accessed April 11, 2025, https://github.com/apache/flink-connector-kafka
78. Sentiment analysis in action: building your real-time pipeline - JavaZone 2024, accessed April 11, 2025, https://2024.javazone.no/program/76e53866-3153-4ecb-b5eb-9812bd5e614c
79. krinart/twitter-realtime-pipeline: An example of Twitter realtime analysis with

Kubernetes, Flink, Kafka, Kafka Connect, Cassandra, Elasticsearch/Kibana, Docker, Sentiment Analysis, Xgboost and Websockets - GitHub, accessed April 11, 2025, https://github.com/krinart/twitter-realtime-pipeline
80. Building a Real-Time IoT Analytics Pipeline: Key Concepts and Tools | by Team Timescale, accessed April 11, 2025, https://medium.com/timescale/building-a-real-time-iot-analytics-pipeline-key-concepts-and-tools-3756cd093724
81. Building a Real-time Dashboard with FastAPI and Svelte | TestDriven.io, accessed April 11, 2025, https://testdriven.io/blog/fastapi-svelte/
82. Learn EVERYTHING About FastAPI in Just ONE Project (No Fluff!) - YouTube, accessed April 11, 2025, https://www.youtube.com/watch?v=I1rDs_H2iAo
83. Python FastAPI Tutorial: Build a REST API in 15 Minutes - YouTube, accessed April 11, 2025, https://www.youtube.com/watch?v=iWS9ogMPOI0
84. Getting Started with WebSocket in Springboot - PieHost, accessed April 11, 2025, https://piehost.com/websocket/getting-started-with-websocket-in-springboot
85. WebSocket Tutorial with Spring Boot | Build One On One Chat Application - YouTube, accessed April 11, 2025, https://www.youtube.com/watch?v=7T-HnTE6v64
86. Implementing Web Sockets with Spring Boot Application - YouTube, accessed April 11, 2025, https://www.youtube.com/watch?v=IzTPUI3WsBg
87. Java WebSocket Client Spring boot - Stack Overflow, accessed April 11, 2025, https://stackoverflow.com/questions/49590598/java-websocket-client-spring-boot
88. WebSocket Tutorial with Spring Boot | Build One On One Chat Application : r/SpringBoot, accessed April 11, 2025, https://www.reddit.com/r/SpringBoot/comments/17zoobj/websocket_tutorial_with_spring_boot_build_one_on/
89. WebSocket example with FastAPI and React. - GitHub, accessed April 11, 2025, https://github.com/ustropo/websocket-example
90. Websocket: Update datastreaming for different stock tickers #10348 - GitHub, accessed April 11, 2025, https://github.com/tiangolo/fastapi/discussions/10348
91. gregwhitaker/springwebflux-websockets-example - GitHub, accessed April 11, 2025, https://github.com/gregwhitaker/springwebflux-websockets-example
92. TechPrimers/spring-boot-websocket-example - GitHub, accessed April 11, 2025, https://github.com/TechPrimers/spring-boot-websocket-example
93. rstoyanchev/spring-websocket-portfolio - GitHub, accessed April 11, 2025, https://github.com/rstoyanchev/spring-websocket-portfolio
94. Realtime Dashboard with FastAPI, Streamlit and Next.js - Part 1 Data Producer, accessed April 11, 2025, https://jaehyeon.me/blog/2025-02-18-realtime-dashboard-1/
95. Sentiment Analysis with Python - A Beginner's Guide - AlgoTrading101 Blog, accessed April 11, 2025, https://algotrading101.com/learn/sentiment-analysis-python-guide/
96. zerodha/pykiteconnect: The official Python client library for the Kite Connect trading APIs, accessed April 11, 2025, https://github.com/zerodha/pykiteconnect

97. Algo Trading Bot - Kite Connect developer forum, accessed April 11, 2025, https://kite.trade/forum/discussion/14763/algo-trading-bot
98. zerodha-kite · GitHub Topics, accessed April 11, 2025, https://github.com/topics/zerodha-kite?l=python&o=asc&s=stars
99. Indian-Algorithmic-Trading-Community repositories - GitHub, accessed April 11, 2025, https://github.com/orgs/Indian-Algorithmic-Trading-Community/repositories
100. Meraj-Mazidi/react-financial-dashboard - GitHub, accessed April 11, 2025, https://github.com/Meraj-Mazidi/react-financial-dashboard
101. play-java-websocket-example/app/stocks/Stock.java at 2.7.x - GitHub, accessed April 11, 2025, https://github.com/playframework/play-java-websocket-example/blob/2.7.x/app/stocks/Stock.java
102. marketcalls/openalgo: Open Source Algo Trading Platform for Everyone - GitHub, accessed April 11, 2025, https://github.com/marketcalls/openalgo