EventTrader Data Flow Documentation

Executive Summary: This document details the pipeline for ingesting, processing, and storing financial event data (news, reports, transcripts) in the EventTrader system. It utilizes Redis for queuing and state management through distinct stages (raw, processed, returns calculation) driven by Pub/Sub messaging, ultimately storing enriched data in Neo4j, while providing mechanisms to track item lifecycles and manage historical data batches.

1. Overview

The EventTrader system ingests financial news (Benzinga), SEC reports, and earnings call transcripts, processes this data through a Redis-based pipeline involving cleaning, metadata enrichment, and financial returns calculation, and finally stores the structured data and relationships in a Neo4j graph database.

A key distinction is made between live and historical data using Redis key prefixes (**`live:`** vs **`hist:`**), allowing them to be processed potentially in parallel but stored distinctly until final stages.

The primary flow involves: 1. **Ingestion:** Fetching data from external APIs (REST/WebSocket/Polling). 2. **Redis Raw Storage & Queuing:** Storing raw data under a unique key in Redis (e.g., **`*:live:raw:*`**, **`*:hist:raw:*`**) after checking for duplicates against the **`PROCESSED_QUEUE`** list. The new **`raw_key`** name is added to a processing queue (**`RAW_QUEUE`**). 3. **Base Processing:** Consuming keys from the **`RAW_QUEUE`** list, retrieving raw data, cleaning/standardizing it, filtering by a defined symbol universe, adding metadata (including return calculation schedules using **`EventReturnsManager`**), storing the processed data under a new key (**`processed`** state, e.g., **`*:live:processed:*`**), adding the key name to a **`PROCESSED_QUEUE`** list (primarily for deduplication checks), and publishing the processed key name via Redis Pub/Sub (**`*:live:processed`** channel). Raw keys may be deleted after successful processing. 4. **Returns Calculation:** Listening to the **`*:live:processed`** Pub/Sub channel (for live items) and periodically scanning Redis/checking a ZSET (for historical items or scheduled future calculations). Retrieves processed data, calculates available financial returns (hourly, session, daily) using **`EventReturnsManager`** as market data becomes available (scheduling future calculations via a Redis Sorted Set **`*:pending_returns`** where the score is the UTC timestamp when data is expected), updating the stored item with calculated returns, moving the item to **`withreturns`**/**`withoutreturns`** states (e.g., **`*:hist:withoutreturns:*`**), and publishing the item ID via Redis Pub/Sub (**`*:withreturns`**/**`*:withoutreturns`** channels). 5. **Neo4j Storage:** Listening to the **`*:withreturns`**/**`*:withoutreturns`** Pub/Sub channels, retrieving the final data (with returns), creating/updating nodes and relationships in Neo4j using likely idempotent `MERGE` operations, and potentially cleaning up the final **`*:withreturns:*`** Redis key.

Separate handling exists for live streaming data and batch historical data ingestion, primarily distinguished by the `live:` vs `hist:` Redis key prefix. The system aims to track each item from ingestion to final storage or failure.

2. Core Components

- **`config/DataManagerCentral.py` (`DataManager`):** The central orchestrator. Initializes and starts all other components (ingestors, processors, returns calculator, Neo4j listener) based on configuration (live/historical flags). Manages startup and shutdown.
- **Source Managers (`BenzingaNewsManager`, `ReportsManager`, `TranscriptsManager` in `DataManagerCentral.py`):** Manage specific data sources (news, reports, transcripts). Initialize API clients, Redis instances (via **`EventTraderRedis`**), the corresponding **`BaseProcessor`** subclass, and the **`ReturnsProcessor`**. Launched by **`DataManager`**.
- **Ingestors (`benzinga/`, `secReports/`, `transcripts/`):** Fetch data from external sources.
    - **Benzinga/SEC:** Use WebSocket clients (**`bz_websocket.py`**, **`sec_websocket.py`**) for live data and REST API clients (**`bz_restAPI.py`**, **`sec_restAPI.py`**) for historical data. Interact with **`RedisClient`** to store raw data and queue items after deduplication check.
    - **Transcripts:** Uses **`EarningsCallProcessor`** for fetching data (scheduling/polling for live, REST for historical). Interacts with **`RedisClient`**.
- **`redisDB/redisClasses.py` (`EventTraderRedis`, `RedisClient`):** Manages Redis connections (live, hist, admin). Provides methods for storing data (`SET`), managing queues (`LPUSH`/`BRPOP` on Lists), Pub/Sub, Sorted Sets (`ZADD`/`ZRANGEBYSCORE`), and key management using standardized namespaces. Performs initial deduplication check against the **`PROCESSED_QUEUE`** list before adding new raw items.
- **`redisDB/BaseProcessor.py` (`BaseProcessor`):** Abstract base class for initial data processing. Runs in a dedicated thread per source type. Consumes `raw_key` names from the **`RAW_QUEUE`** list, retrieves raw data, cleans/standardizes, filters by symbols in the allowed universe, adds metadata/schedules (via **`EventReturnsManager`**), stores result under a **`processed_key`**, pushes `processed_key` to **`PROCESSED_QUEUE`** list (for deduplication record), publishes `processed_key` to **`*:live:processed`** Pub/Sub, and optionally deletes the original **`raw_key`**. Pushes failed `raw_keys` to **`FAILED_QUEUE`** list upon exceptions.
- **`redisDB/NewsProcessor.py`, `ReportProcessor.py`, `TranscriptProcessor.py`:** Subclasses of **`BaseProcessor`**. Implement source-specific data cleaning (`_clean_content`) and standardization (`_standardize_fields`).
- **`eventReturns/EventReturnsManager.py` (`EventReturnsManager`):** Utility class providing methods for:
    - `process_event_metadata`: Called by **`BaseProcessor`** to generate return *schedules* based on event time and market hours.
    - `process_events`: Called by **`ReturnsProcessor`** (especially for historical batches) to calculate actual financial *returns* based on schedules and market data availability (using **`polygonClass.py`**).
- **`eventReturns/ReturnsProcessor.py` (`ReturnsProcessor`):** Calculates financial returns. Runs in a dedicated thread per source type. Listens to **`*:live:processed`** Pub/Sub for live processed items. Also periodically scans historical **`*:hist:processed:*`** keys (`_process_hist_news`) and checks the **`*:pending_returns`** ZSET for scheduled calculations. Calculates available

returns (using **`EventReturnsManager`**), schedules future calculations in **`*:pending_returns`** ZSET (score is UTC timestamp when data is ready), updates items, stores updated data under **`*:withreturns:*`** or **`*:withoutreturns:*`** keys (deleting the **`*:processed:*`** key), and publishes `item_id` to **`*:withreturns`**/**`*:withoutreturns`** Pub/Sub channels. **Note:** Polygon API or other processing failures might leave items stuck in **`withoutreturns`** state without specific automatic recovery.

- **`neograph/Neo4jProcessor.py` (`Neo4jProcessor`):** Handles interaction with Neo4j.
    - **Live Mode:** Runs `process_with_pubsub` (via **`PubSubMixin`**, started by **`DataManager`**) listening to **`*:withreturns`**/**`*:withoutreturns`** Pub/Sub, retrieves data from Redis, calls **`Neo4jManager`** to write. May delete **`*:withreturns:*`** key on success. **Note:** Needs robust handling for Neo4j write failures (retry, logging) to prevent data loss after Pub/Sub consumption. Includes XBRL reconciliation (triggered after connection establishment, often during initialization sequence) on startup.
    - **Batch Mode:** Provides functions executable via command line (**`neo4j_processor.sh`**) to process data directly from Redis (primarily **`*:withreturns:*`**, **`*:withoutreturns:*`** states) into Neo4j.
- **`neograph/Neo4jManager.py` (`Neo4jManager`):** Executes Cypher queries against Neo4j (creating/merging nodes/relationships). Called by **`Neo4jProcessor`**. Likely uses idempotent `MERGE` operations to handle potential duplicates safely.
- **`neograph/Neo4jInitializer.py` (`Neo4jInitializer`):** Sets up Neo4j schema (constraints, indexes, initial nodes like dates, companies). Run by **`DataManager`** on startup if needed.
- **`scripts/run_event_trader.py`:** Main script to start the **`DataManager`**. Handles command-line args, sets feature flags dynamically, and manages process exit conditions based on run mode.
- **`scripts/*.sh`:** Helper scripts for running the application, batch processing, watchdog, etc. Includes `event_trader.sh` which provides the `chunked-historical` command.

3. Live Data Ingestion: WebSocket vs. Scheduled/Polled

A key difference exists in how live data enters the system:

- **News & Reports (Benzinga/SEC): WebSocket Push**
    - Utilize persistent WebSocket connections (**`bz_websocket.py`**, **`sec_websocket.py`**), started conditionally based on `ENABLE_LIVE_DATA` flag.
    - External providers push data in real-time.
    - **`DataManager`** manages dedicated WebSocket client threads.
    - **Result:** Low-latency, event-driven ingestion when live mode is active.
- **Transcripts: Scheduled/Polled Fetch**
    - **No WebSocket.** Relies on fetching data based on known schedules or polling.
    - **Scheduling:** **`TranscriptsManager._initialize_transcript_schedule`** fetches *today's* earnings calendar, schedules processing times (call time + 30 min)

in Redis ZSET (**`admin:transcripts:schedule`**). This runs on startup regardless of flags (considered safe).
    - **Fetching Trigger:** A background thread started by **`TranscriptProcessor`** (`_run_transcript_scheduling`) monitors the **`admin:transcripts:schedule`** ZSET and triggers fetching (`_fetch_and_process_transcript`) when an item's scheduled time is reached.
    - **Result:** Higher latency, dependent on scheduling accuracy and background thread timing.

This difference impacts the real-time availability of transcript data compared to news and SEC filings.

5. Redis Usage Details

Redis is central to the workflow, used for queuing, intermediate data storage, state management, and inter-process communication.

- **Key Namespaces & States:** Keys structure: `{source}:{prefix}:{state}:{id}`.
    - `source`: **`news`**, **`reports`**, **`transcripts`**.
    - `prefix`: **`live`**, **`hist`**. (Distinguishes real-time vs. historical).
    - `state`: Represents processing stage:
        - **`raw`**: Initial data via `SET`. Deleted by **`BaseProcessor`** (optional).
        - **`processed`**: Cleaned data + metadata/schedule via `SET`. Deleted by **`ReturnsProcessor`**.
        - **`withoutreturns`**: Returns calculation in progress/pending via `SET`. Deleted by **`ReturnsProcessor`** on full completion.
        - **`withreturns`**: Final state (all returns calculated) via `SET`. May be deleted by **`Neo4jProcessor`** (live mode) after successful Neo4j write.
    - `id`: Unique item identifier (e.g., `newsId.timestamp`, `accessionNo.timestamp`, `symbol_timestamp`).
- **Queues (Redis Lists):**
    - `{source}:queues:raw` (**`RAW_QUEUE`**): Stores `raw_key` names. Pushed by ingestors (`LPUSH`) after dedupe check, consumed by **`BaseProcessor`** (`BRPOP`). Primary work queue.
    - `{source}:queues:processed` (**`PROCESSED_QUEUE`**): Stores `processed_key` names. Pushed by **`BaseProcessor`** (`LPUSH`). Primarily used by **`RedisClient`** for deduplication checks *before* adding new items to `RAW_QUEUE`. Not actively consumed by processors.
    - `{source}:queues:failed` (**`FAILED_QUEUE`**): Stores `raw_key` names of items failing in **`BaseProcessor`**. Pushed by **`BaseProcessor`** (`LPUSH`). **Requires manual investigation/recovery.**
- **Pub/Sub Channels:** For event-driven notifications.
    - `{source}:live:processed`: **`BaseProcessor`** publishes `processed_key` name. **`ReturnsProcessor`** subscribes (for live items).
    - `{source}:withreturns`, `{source}:withoutreturns`: **`ReturnsProcessor`** publishes `item_id`. **`Neo4jProcessor`** (live mode) subscribes.
- **Sorted Set (ZSET):**
    - `{source}:pending_returns` (**`*:pending_returns`**): Used by **`ReturnsProcessor`** to schedule future return calculations. Members: `{item_id}:{return_type}`. Score: **UTC Unix timestamp (float)** when

market data should be available (schedule time + Polygon delay). Checked via `ZRANGEBYSCORE`.

- **Other Keys:**
    - **`admin:*`**: Used by **`EventTraderRedis`** for shared config (stock universe: **`admin:tradable_universe:*`**) and transcript scheduling (**`admin:transcripts:schedule`**).
    - **`batch:{source}:{date_range}:fetch_complete`**: Flag set by historical fetch functions upon completion. Used by `run_event_trader.py`'s historical monitoring loop.

6. Tracking Item Lifecycle Summary Table

This table summarizes the typical happy-path journey of an item:

Stage	Redis State/Action	Responsible Component	Trigger	Output / Next Step
1. Ingestion Attempt	Check `PROCESSED_QUEUE` for processed_key	Ingestor / RedisClient	External Data / API Call	Proceed if not found
2. Ingestion Success	`SET *:raw:*`, `LPUSH RAW_QUEUE`	Ingestor / RedisClient	Data Received (Not Dupe)	`raw_key` in `RAW_QUEUE`
3. Base Processing Start	`BRPOP RAW_QUEUE`	BaseProcessor	Item in `RAW_QUEUE`	Get raw data
4. Base Processing Success	`DEL *:raw:*`(opt), `SET *:processed:*`, `LPUSH PROCESSED_QUEUE`, `PUBLISH *:live:processed`	BaseProcessor	Successful Processing	`processed_key` published
5. Returns Calc Start	`GET *:processed:*`, `DEL *:processed:*`, `SET *:withoutreturns:*`, `PUBLISH *:withoutreturns`, `ZADD *:pending_returns`	ReturnsProcessor	Pub/Sub `*:live:processed` or Hist Scan	`item_id` published, future returns scheduled
6. Pending Return Calc	`GET *:withoutreturns:*`, Calc Return, `SET *:withoutreturns:*`, `PUBLISH *:withoutreturns`	ReturnsProcessor	ZSET Check (`ZRANGEBYSCORE`)	Item updated, `item_id` published
7. All Returns Calc	`GET *:withoutreturns:*`, Calc Final Return, `DEL *:withoutreturns:*`,	ReturnsProcessor	Last Pending Return Calc	Item moved to final state, `item_id` published

Stage	Redis State/Action	Responsible Component	Trigger	Output / Next Step
	`SET *:withreturns:*`, `PUBLISH *:withreturns`, `ZREM *:pending_returns`			
8. Neo4j Ingestion (Live)	`GET *:withreturns:*/*:withoutreturns:*`, Write to DB, `DEL *:withreturns:*` (opt.)	Neo4jProcessor (Live)	Pub/Sub `*:withreturns/` etc.	Data in Neo4j, optional Redis cleanup
9. Neo4j Ingestion (Batch)	`SCAN/GET *:withreturns:*/*:withoutreturns:*`, Write to DB	Neo4jProcessor (Batch)	Manual Script Run	Data in Neo4j

Failure Paths: * If Stage 4 fails -> `raw_key` pushed to **`FAILED_QUEUE`**. Processing stops for that item. Requires manual review. * If Stage 5/6/7 fails -> Item might remain stuck in **`withoutreturns`** state indefinitely. Requires investigation. * If Stage 8 fails -> Data might not reach Neo4j; depends on retry logic. `withreturns` key might not be cleaned up.

7. Historical Data Ingestion

- **Trigger:** Initiated by **`scripts/run_event_trader.py`** with `--from-date`, `--to-date` and `-historical` flag active. The **`hist:`** prefix is used in Redis keys (e.g., `news:hist:raw:...`).
- **Fetching:** Source managers conditionally call REST clients (`get_historical_data`).
- **Storage & Queuing:** Data stored under **`*:hist:raw:*`** keys, keys pushed to **`RAW_QUEUE`** after dedupe check against **`PROCESSED_QUEUE`**. Fetch function sets `batch:...:fetch_complete` flag.
- **Processing:** Flows through the same **`BaseProcessor`** and **`ReturnsProcessor`** instances as live data.
    - **`BaseProcessor`** creates **`*:hist:processed:*`** keys, pushes to **`PROCESSED_QUEUE`**, publishes key to **`*:live:processed`** Pub/Sub.
    - **`ReturnsProcessor`** handles historical `processed` items via its batch scan (`_process_hist_news`), creates **`*:hist:withreturns:*`** or **`*:hist:withoutreturns:*`** keys (deleting **`*:hist:processed:*`**), schedules future returns in ZSET (**`*:pending_returns`**), and publishes `item_id` to **`*:withreturns`**/**`*:withoutreturns`**.
- **Neo4j Loading:** Can be picked up by the live **`Neo4jProcessor`** listener (if running) or processed in bulk using the manual **`neo4j_processor.sh`** script. Idempotent `MERGE` operations in **`Neo4jManager`** are crucial for handling potential overlaps if live/historical runs cover same periods.

8. Chunked Historical Processing & Completion Tracking

To reliably process large historical date ranges without overwhelming system resources or external APIs, the `chunked-historical` command in `scripts/event_trader.sh` provides a sequential batch processing workflow.

Overall Goal: Ensure that all critical processing stages (fetching, base processing, returns calculation) related to one historical date chunk are complete *according to Redis state* before stopping the system and starting the fetch for the next chunk.

Workflow:

1. **Shell Script Orchestration (`event_trader.sh`):**
   - The `process_chunked_historical` function calculates sequential date chunks (e.g., 5 days) based on the overall date range provided.
   - For each chunk:
     - It executes the main Python script (`scripts/run_event_trader.py`) with the chunk's specific start/end dates and the `-historical` flag.
     - It **waits** for that Python process to complete and exit.
     - It checks the Python process's exit code. If successful (0), it proceeds.
     - It calls `$0 stop-all` to terminate all related background processes (processors, etc.) ensuring a clean slate for the next chunk.
     - It advances to the next date chunk and repeats the process.
2. **Python Script Responsibility (`run_event_trader.py` in `-historical` mode):**
   - **Fetch & Start Background Tasks:** Initializes `DataManagerCentral`, starts historical data fetching for the *current chunk*, and launches background daemon threads (BaseProcessors, ReturnsProcessor, Neo4jProcessor, XBRL workers).
   - **Fetch Completion Signal:** The historical fetch functions for each source (`news`, `reports`, `transcripts`) set a specific Redis flag (`batch:{source}:{chunk_start}-{chunk_end}:fetch_complete`) just before they finish fetching data for the chunk.
   - **Wait for Processing Completion (Redis Monitoring):** Instead of exiting immediately or looping forever, the main Python thread enters a monitoring loop. This loop **periodically checks Redis** to determine if all processing related to the chunk appears finished. It waits until **all** of the following conditions are met for **all three** sources (`news`, `reports`, `transcripts`):
     1. **Fetch Complete Flag is Set:** (`GET batch:{source}:{...}:fetch_complete` == “1”) - *Confirms fetching is done.*
     2. **Raw Queue is Empty:** (`LLEN {source}:queues:raw` == 0) - *Confirms BaseProcessor has picked up all raw items.*
     3. **Pending Returns Set is Empty:** (`ZCARD {source}:pending_returns` == 0) - *Confirms ReturnsProcessor has processed all scheduled returns for this chunk.*
     4. **`withoutreturns` Namespace is Empty:** (No keys match `{source}:withoutreturns:*`) - *Confirms ReturnsProcessor has finished with items needing future returns calculation (moved to `withreturns`).*
     5. **`withreturns` Namespace is Empty:** (No keys match `{source}:withreturns:*`) - *Confirms Neo4jProcessor (or other consumers) has likely processed final items from this chunk.*

    - **Clean Exit:** Only when all the above conditions are met does the Python monitoring loop end. It then calls `manager.stop()` (for graceful background thread shutdown) and `sys.exit(0)` to terminate the Python process successfully.
3. **Shell Script Proceeds:** The shell script detects the successful Python exit (code 0), runs `stop-all`, and moves to the next chunk.

Handling Asynchronous Tasks (XBRL):

- XBRL processing is triggered during report processing but runs asynchronously in a background thread pool.
- The Python monitoring loop **does not** directly wait for XBRL tasks to finish (it only monitors Redis state).
- Therefore, the `stop-all` command *can* interrupt active XBRL worker threads.
- To handle this, an **XBRL reconciliation mechanism** runs after the `Neo4jProcessor` establishes its connection (typically during the initialization sequence at the start of each chunk). It queries Neo4j for reports with `xbrl_status` 'QUEUED' or 'PROCESSING' and automatically re-queues them using the existing XBRL processing infrastructure.

This combined shell orchestration and Python monitoring ensures sequential processing of historical chunks while allowing necessary background tasks like returns calculation and Neo4j updates to complete based on Redis state, with a specific reconciliation step for asynchronous XBRL processing.