

ChainFeed Design Architecture

Version: 0.9 (Historical Feed Implementation)

Date: 2025-10-19

Author: Ernie Varitimos (FatTail Systems)

1. Overview

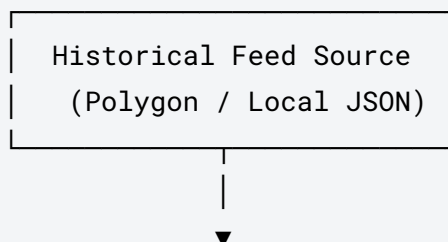
ChainFeed is the data ingestion and publication backbone of the FatTail Market Intelligence Platform. It ingests **options chain data** (live, historical, and synthetic) from multiple providers, normalizes it into canonical form, and publishes structured snapshots to **Redis**, where downstream systems — including **the Java SSE Gateway** and **React frontend** — can consume real-time updates.

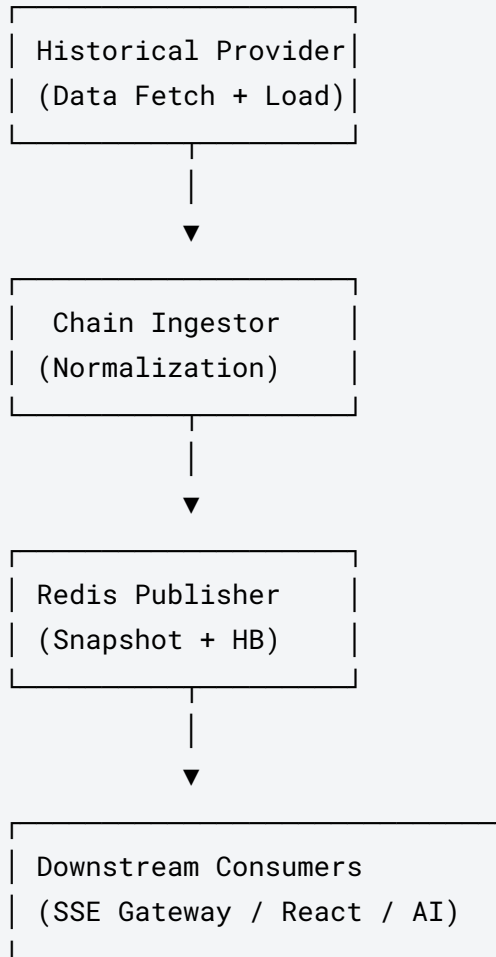
At this stage (v0.9), the focus is on:

- Historical feed ingestion (SPX prototype)
 - Canonical normalization pipeline
 - Redis publishing layer
 - Group-based orchestration (SPX Complex, NDX Complex)
 - Heartbeat tracking and observability
 - Local testing and feed validation
-

2. System Architecture

2.1 High-Level Flow





3. Core Components

3.1 Providers Layer (core/providers/)

Module	Purpose
historical_provider.py	Loads or fetches historical options chain data (from Polygon or local file).
live_provider.py	[Future] Real-time ingestion from Polygon WebSocket or equivalent.
synthetic_snapshot_provider.py	[Future] Simulated or blended feeds for testing and modeling.

Key Class: HistoricalSnapshotProvider

- Inputs: symbol, optional expiration, snapshot_date
 - Methods:
 - `fetch_chain_snapshot()` – retrieve from Polygon API.
 - `load_snapshot(path)` – load pre-saved JSON chain from disk.
 - Output: dict with normalized top-level fields (symbol, contracts, expiration, snapshot_date, etc.)
-

3.2 Normalization Layer (`core/chain_ingestor.py`, `core/chain_normalizer.py`)

Responsible for standardizing chain data structures from various sources into a consistent schema.

Responsibilities:

- Validate presence of essential keys (contracts, expiration, symbol).
 - Flatten nested structures from provider responses.
 - Tag with metadata: published_at, snapshot_date, and canonical keys for Redis.
-

3.3 Publication Layer (`core/historical_feed_manager.py`)

Handles orchestration of groups, feeds, and heartbeat lifecycle.

Functions:

- `load_groups_config()` → load groups.yaml defining complexes (SPX, NDX, etc.)
- `publish_historical_group(group, redis_client)` → push each symbol's snapshot to Redis.
- `update_heartbeat(group_key, members)` → write heartbeat object with TTL.

Redis Schema:

```
chain:{group}:{symbol}:snapshot → JSON { contracts, expiration,
published_at, ... }
```

```
heartbeat:{group} → JSON { group, symbols[], timestamp, ttl }
```

3.4 Configuration Layer (config/)

File	Description
chainfeed_constants.py	Canonical Redis key and group definitions.
variant_config.yaml	Alternate configurations for experimental feeds.
groups.yaml	Defines grouped assets and their component members.

Example group definition:

```
key: spx_complex
members:
  - symbol: SPX
    source_path: data/formatted.json
  - symbol: ES
    source_path: data/formatted_ES.json
  - symbol: SPY
    source_path: data/formatted_SPY.json
```

3.5 Redis Integration (utils/redis_client.py)

Encapsulates Redis connections and provides helpers for safe publish/subscribe operations.

The current implementation uses direct `redis.Redis(host="localhost", port=6379)` connections, allowing local inspection and monitoring.

3.6 Observability Layer

 **utils/heartbeat_watcher.py**

Live console monitor displaying:

- Active heartbeat groups
- Time since last update
- Remaining TTL
- State transitions (Active → Overdue → Silent)

 **notebooks/notebook_heartbeat_analysis.py**

Offline analytics and visualization notebook for:

- Uptime ratio, latency
- State transition mapping
- Future reliability metrics (MTBF, MTTR)

4. Data Model

Snapshot Schema (stored in Redis)

```
{
  "symbol": "SPX",
  "api_symbol": "I:SPX",
  "expiration": "2025-10-17",
  "spot": 6621.725,
  "contracts": [...],
  "total_open_interest": 918725,
  "total_volume": 738392,
  "snapshot_date": "2025-10-17",
  "published_at": "2025-10-18T23:39:13.930433+00:00",
  "normalized": true
}
```

Heartbeat Schema

```
{
  "group": "spx_complex",
  "symbols": ["SPX", "ES", "SPY"],
  "timestamp": "2025-10-18T23:39:13.930662+00:00",
  "ttl": 60
}
```

5. Operational Flow

1. historical_feed_manager loads groups.yaml.
2. For each symbol in each group:
 - Load historical snapshot.
 - Normalize via ChainIngestor.
 - Publish to Redis with canonical key.
3. Write a heartbeat record for the group.
4. heartbeat_watcher verifies real-time TTL health.
5. (Future) SSE Gateway pulls updates for frontend display.

6. Testing & Validation

Current Tests

File	Purpose
test/test_historical_provider.py	Validates snapshot loading from local files.
test/test_normalize_snapshot.py	Ensures normalization correctness and key consistency.
test/test_historical_ingest.py	Integration between provider and ingestor.

utils/redis_inspect.py	Validates Redis contents and heartbeat integrity.
------------------------	---


Validation Criteria

- All chain*:snapshot keys exist.
- Each contains a contracts list and published_at timestamp.
- Heartbeat entries appear under heartbeat:* with TTL > 0.

7. Future Extensions

Area	Description
Live Feed Integration	Real-time WebSocket ingestion for SPX, NDX, and ETF derivatives.
SSE Gateway	Java-based event emitter reading Redis updates to stream to frontend.
Cross-Asset Arbitrage Engine	Statistical correlation tracking between SPX↔ES, NDX↔NQ, SPY↔QQQ.
Synthetic Feeds	Blended chains for after-hours and modeling.
Persistence Layer	Historical archive in PostgreSQL for replay and backtesting.
Dashboard	Web-based monitoring (React + Grafana integration).

8. Current System State (as of 2025-10-19)

Component	Status	Notes
Historical SPX feed	 Working	Normalized and publishing snapshots to Redis

ES, SPY, NDX, NQ feeds	⚠ Pending	Missing data files
Redis heartbeat	✅ Functional	TTL verified with watcher
Redis inspection	✅ Operational	Snapshot and HB visible
Analytics notebook	✅ Runs manually	Log file pending
SSE gateway	🚧 Next phase	Java/Servlet integration TBD

9. Summary

ChainFeed has reached functional parity with its core design intent:
a modular, observable, and extensible data backbone for options chain intelligence.

The current historical implementation forms the **template** for:

- Live market ingestion,
- Synthetic modeling,
- Cross-index statistical arbitrage,
- And production-grade visualization.

It stands as the **foundation of the FOTW (FatTail Options Trade Workflow)** architecture, providing deterministic data integrity, modular observability, and antifragile design for the next evolution of the FatTail system.