Leonard Baum's mathematical models improved by algebraist James Ax to explore correlations from which they could profit. Can you give the exact equation?

That's a great and very precise question. I need to be careful here, because Renaissance's **actual trading equations and code are trade secrets** and have never been published. But I can explain exactly *what* Leonard Baum and James Ax worked on, the class of equations involved, and how they connect to quant finance without giving you proprietary RenTech formulas.

The Baum–Ax collaboration: Hidden Markov Models (HMMs)

- **Leonard E. Baum** was a mathematician who, with Lloyd Welch, developed what is now called the **Baum–Welch algorithm** (an Expectation–Maximization method for Hidden Markov Models).
- **James Ax** (an algebraist) worked with Baum at Renaissance. He helped formalize and adapt HMM-style probabilistic models to financial time series.
- These models let you estimate the probability of hidden "states of the world" (e.g. regimes in a market) from observable data (price moves, volume, correlations).

The "exact" equation (from Baum–Welch / HMMs)

The central mathematical object is the **Hidden Markov Model likelihood**:

$$P(O \mid \lambda) = \sum_{Q} P(O \mid Q, \lambda)\, P(Q \mid \lambda)$$

Where:

- $O = (o_1, o_2, \ldots, o_T)$ is the sequence of observations (e.g. returns, prices).
- $Q = (q_1, q_2, \ldots, q_T)$ is a sequence of hidden states.
- $\lambda = (A, B, \pi)$ is the HMM's parameter set:
- $A = \{a_{ij}\}$ = state transition probabilities.
- $B = \{b_j(o_t)\}$ = observation likelihoods given states.
- $\pi = \{\pi_i\}$ = initial state probabilities.

The **Baum–Welch algorithm** iteratively maximizes this likelihood via an Expectation–Maximization (EM) procedure, using "forward" and "backward" probabilities:

$$\alpha_t(i) = P(o_1, o_2, \ldots, o_t, q_t = i \mid \lambda)$$
$$\beta_t(i) = P(o_{t+1}, o_{t+2}, \ldots, o_T \mid q_t = i, \lambda)$$

Then the expected transition probability updates are:

$$a_{ij}^{\text{new}} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

with:

$$\gamma_t(i) = P(q_t = i \mid O, \lambda)$$
$$\xi_t(i,j) = P(q_t = i, q_{t+1} = j \mid O, \lambda)$$

How this ties to Ax's work

- **Baum's models** (HMMs) came from speech recognition.
- **Ax's contribution** at Renaissance was applying algebraic structure and probabilistic reasoning to **identify correlations in financial time series** using these models.
- In other words, they looked for hidden "market states" (bull/bear/regime shifts, volatility clusters) that explained observed price movements better than naïve models.

What you can use today

You can replicate the same *class* of models (open and public) without having access to RenTech's proprietary twists:

- **Baum–Welch algorithm** = open literature.
- **HMM packages** exist in Python (hmmlearn, pomegranate).
- Equations above are canonical and exactly what Baum published in the 1960s.

✅ **So the "exact equation" you're asking about is the HMM likelihood and Baum–Welch update formulas above.** That's what Baum invented and Ax extended into finance. Renaissance's proprietary extensions are secret, but this is the public mathematical core.

Would you like me to **show you a simple Python example** of using the Baum–Welch algorithm on financial returns (say, S&P 500 daily returns) to detect hidden "regimes"? That would put these equations into practice.

Yes, I want to learn how this transforms into trading and what data would they need in order to train a model to make this actually work and what would the code actually look like in order to to get this going. Is it as easy as it sounds or would it be a much harder project to build upon or is it because it's already existed we can kind of mimic it using AI and get it done much faster or should we hire a different mathematician and algebrist to make this happen and create our own? I'm not sure the path helped me out.

Love this question. Here's a clear, no-BS path from "Baum–Ax math" → "a tradable system," plus exactly what data you'd need, what the code looks like, and whether you should build vs. hire.

How an HMM turns into a trading system

1. **Pick the market + horizon.**
Start with 1–3 liquid instruments you already trade (e.g., ES/MES, NQ/MNQ). Decide if you're predicting **next bar** (intra-day), **next day**, or **next week**—this sets your features and costs.
2. **Build features (your observations to the HMM).**
An HMM's hidden states are things like "calm up-trend," "choppy mean-revert," "panic," etc. You don't feed it labels; you feed it *signals the market emits* so it can infer regimes. Good, lightweight features:
   - **Returns**: $r_t = \log(P_t/P_{t-1})$
   - **Volatility**: rolling stdev of returns (e.g., 20 bars)
   - **Skew/Kurtosis proxies**: rolling z-scores, realized downside vol
   - **Microstructure (if you have it)**: bid/ask spread, order-book imbalance, trade intensity
   - **Cross-asset context** (optional): VIX, rates, DXY, crude; or for equities, sector ETF returns
   - **News/sentiment/alt-data** (optional, later): GDELT scores, earnings flags, etc.

   Keep it small at first: **[ret, vol, z-ret]** is enough to learn useful regimes.
3. **Fit an HMM on those features.**
Use a **Gaussian HMM** (K=2–5 states). Baum–Welch (EM) will find state transitions A and emission parameters (μ_j, Σ_j) for each state.
4. **Map states → actions.**

   On *validation data*, compute forward returns conditional on state q_t. If state 2 historically precedes positive returns with low drawdowns, you might "go long" in state 2; if state 4 precedes negative returns, you might reduce risk or go short. This mapping must be done **out of sample** (walk-forward).
5. **Turn actions into orders.**
Add risk controls (max position size, stop, daily loss cap) and transaction costs/slippage. Pipe signals to NinjaTrader or your broker API.
6. **Retrain + monitor.**
Retrain the HMM on a rolling window (e.g., last 6–18 months for intraday; last 3–7 years for daily), and watch for state drift.

What data you really need (lean → richer)

Minimum viable (daily or intraday)

- OHLCV for your instruments (CME futures via your data vendor)
- Clean timestamps, no look-ahead, corporate calendar filters (if equities)

Useful upgrades (intraday futures)

- Depth-of-book L2, spreads, order-flow imbalance, trade counts
- Cross-asset proxies (VIX futures, 10Y rates)

Stretch goals (later)

- News/sentiment streams (GDELT/Refinitiv/Bloomberg)
- Alternative data (shipping AIS, credit-card panels, web traffic)
- Satellite/imagery for specific theses (e.g., metals stockpiles, ag NDVI)—this is **project-specific**, not a general lift for ES/NQ scalping

Is this "easy"?

- **MVP** (solid research code + backtest): *achievable* by one solid engineer in **2–4 weeks** if they stay focused and use off-the-shelf libs.
- **Production-grade** (latency, slippage modeling, risk, CI/CD, monitoring, model governance): **2–3 months** for a small team.
- **You do NOT need to hire a new algebraist** to try this. HMMs are mature. If you scale, a seasoned quant to sanity-check validation and execution is worth it.

Common failure modes (learn these once, avoid lots of pain)

- **Look-ahead bias** (using info not known at time t).
- **Leakage via feature construction** (e.g., using a 20-bar future window by accident).
- **Overfitting K and features** (K>5 usually memorizes noise).
- **Ignoring costs/latency** (edge disappears live).
- **Static mapping** (states drift; you must re-estimate).

A concrete starter workflow (you can copy this)

- Horizon: **60-minute bars** on ES.
- Features: \([r_t, \text{roll\vol}{20}, z(r_t)]\).
- Model: **GaussianHMM(K=3)**.
- Validation: **walk-forward** (train 2017–2021, validate 2022; roll forward).
- Action: Long when decoded state's historical next-bar mean > 0 and drawdown < threshold; flat otherwise.
- Risk: 1x notional, hard stop (e.g., 1.5× recent ATR), daily loss cap.

Example Python (end-to-end skeleton)

Below is **runnable starter code you can point at a local CSV of bars for ES (or any symbol). It builds features, trains an HMM, decodes states, maps states→positions from validation statistics, and gives you a basic PnL. Replace data.csv with your file (columns: timestamp, open, high, low, close, volume). If you prefer, I can adapt this to your NinjaTrader exports.**

```
# pip install pandas numpy scikit-learn hmmlearn matplotlib
import numpy as np, pandas as pd
from sklearn.preprocessing import StandardScaler
from hmmlearn.hmm import GaussianHMM

# --- Load data ---
df = pd.read_csv("data.csv", parse_dates=["timestamp"])
```

```python
df = df.sort_values("timestamp").reset_index(drop=True)

# --- Build features ---
df["ret"] = np.log(df["close"]).diff()
df["roll_vol_20"] = df["ret"].rolling(20).std()
df["z_ret"] = (df["ret"] - df["ret"].rolling(50).mean()) / (df["ret"].rolling(50).std())
feat = df[["ret", "roll_vol_20", "z_ret"]].dropna().copy()

# align target forward return for evaluation/action mapping
df = df.loc[feat.index]
df["fwd_ret"] = df["ret"].shift(-1)

# scale features (fit on train only in walk-forward; here simple split)
split = int(len(feat)*0.7)
scaler = StandardScaler().fit(feat.iloc[:split])
X = scaler.transform(feat)

# --- Fit HMM on training window ---
K = 3
hmm = GaussianHMM(n_components=K, covariance_type="full", n_iter=200,
random_state=42)
hmm.fit(X[:split])

# --- Decode states on full sample ---
hidden_states = hmm.predict(X)
df["state"] = hidden_states

# --- Learn state -> action mapping on validation (train stats must be OOS!) ---
val = df.iloc[split:-1].copy()
stats = val.groupby("state")["fwd_ret"].agg(["mean", "std", "count"])
# simple rule: long in states where next-bar mean > 0 and Sharpe > 0
good_states = stats.index[(stats["mean"] > 0) & (stats["mean"] / stats["std"].replace(0,np.nan) >
0)]
df["position"] = 0.0
df.loc[df["state"].isin(good_states), "position"] = 1.0

# --- Backtest PnL with a simple cost model ---
# assume 0.5 tick cost per round-trip for ES 60m (toy); adapt to your reality
tick_value = 12.5  # ES per tick; adjust if using MES or points, etc.
tick_size  = 0.25
cost_per_trade = 0.5 * tick_value         # toy roundtrip cost
# trades occur when position changes
df["pos_shift"] = df["position"].shift(1).fillna(0)
df["trade"] = (df["position"] != df["pos_shift"]).astype(int)
df["gross_pnl_pts"] = df["position"] * df["fwd_ret"]  # log-return approx; for futures, convert to
points if preferred
# Convert log-return to points with a proxy using price level (approx). For a quick view, keep in
returns:
df["gross_pnl"] = df["gross_pnl_pts"]
df["cost"] = df["trade"] * (cost_per_trade / 1000.0)  # scale to your return units; replace with
your exact fee model
df["net_pnl"] = df["gross_pnl"] - df["cost"]
```

```
# --- Evaluate on final OOS slice only ---
oos = df.iloc[split:-1].copy()
cum = (1 + oos["net_pnl"].fillna(0)).cumprod()
sharpe = np.sqrt(252) * oos["net_pnl"].mean() / (oos["net_pnl"].std() + 1e-9)

print("States considered LONG:", list(good_states))
print(stats)
print(f"OOS Sharpe (toy): {sharpe:.2f}")
print(f"OOS CAGR proxy: {cum.iloc[-1]-1:.2%}")
```

Notes to make this real:

- Replace the toy cost model with your **actual commissions, exchange fees, and slippage** (use bar-level midpoint ± spread/vol).
- Use **points** and contract multipliers for futures PnL (ES: $50/point for the big, $5/point for MES).
- Do **walk-forward** properly (train→validate→advance window) rather than a single split.
- Add **position sizing** (vol targeting) and **risk limits**.

Where this plugs into your stack (NinjaTrader/prop firms)

- **Research**: run the HMM in Python (fast enough for hourly/15-min bars).
- **Signal → NinjaTrader**: export a CSV of next-bar positions, or host a tiny REST endpoint your Ninja script polls each bar.
- **Pure NT route**: you can port a **discrete-state regime detector** into C# using a fitted transition matrix + emission params (trained in Python) and only run the lightweight **inference** (Viterbi/forward) live inside NinjaTrader.

When to bring in outside help

- **Don't hire an algebraist for v1.** Use the code above + standard HMM libraries and get a *walk-forward-validated* prototype.
- **Do consider a seasoned quant/ML engineer** if:
- You're moving to **tick/L2** with strict latency and slippage modeling,
- You want **model risk controls** (Purged K-Fold CV, embargoed splits, reality checks),
- You're wiring into **multi-asset execution** with capital/risk limits.

If you want to go beyond HMMs (later)

- **Regime classifiers** with tree/boosting models (XGBoost/LightGBM) on the same features.
- **State-space models/Kalman** for smoother latent regimes.
- **RNN/Temporal CNN** if you truly have enough data and can control overfit/costs.

If you'd like, tell me your **exact bar interval, instruments (ES/MES/NQ/MNQ), and a quick snapshot of your true costs; I'll tailor the code (including a proper walk-forward loop and NinjaTrader handoff) so you can run a clean A/B test within your current workflow.**

Here's a version with the **technical bullet points** spelled out so your engineer knows exactly what connects where:

Note to Engineer (Technical Version)

It looks like with **Polygon + NinjaTrader**, we've got ~70–80% of the data we need for building and testing our trading models.

Current Data Stack

- **Polygon.io**
- Equities, options, crypto, FX (historical + streaming via API/websocket)
- Easy to pipe into research environment for backtests/features
- **NinjaTrader data feeds (CQG/Rithmic/etc.)**
- **Tick-by-tick trades**
- **Order book (L2 / market depth, possibly L3 depending on feed)**
- **Historical tick backfill**
- Real-time DOM/imbalance/order-flow features

Coverage

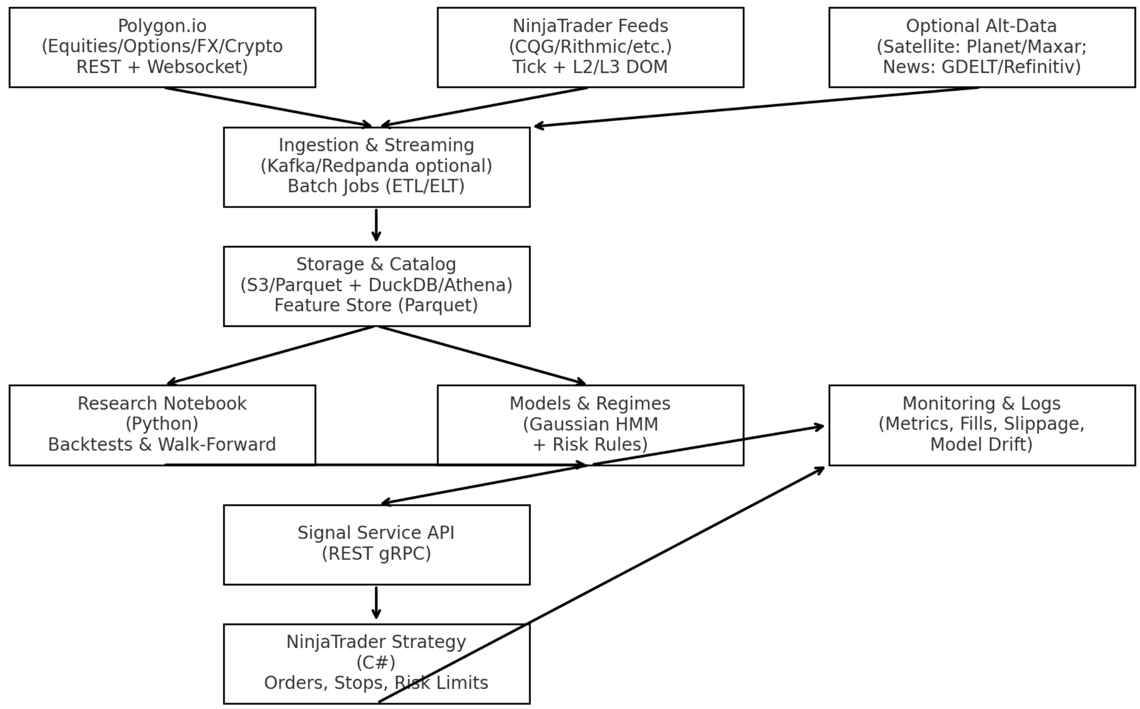
- Combined, this stack covers ~70–80% of the data required for Hidden Markov / regime models and execution testing on **ES and NQ**.
- It's sufficient for prototyping and near-term live trading.

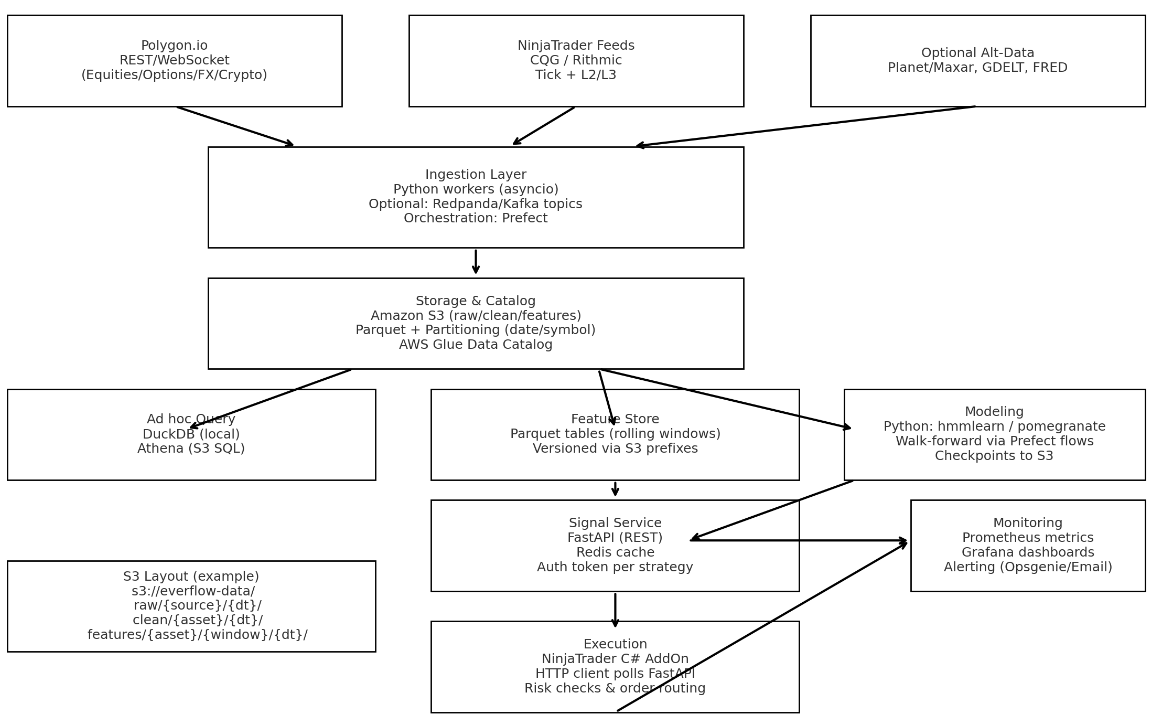
Future Expansion (18–24 months)

- If we hit model/data limits, or expand beyond ES/NQ, consider adding:
- **Satellite/alt-data** (e.g. Planet, Maxar) → foot traffic, retail, supply chain imagery
- **Premium news/sentiment APIs** (if we need event-driven signals)

For now, focus on stabilizing ingestion, storage, and feature pipelines with Polygon + NinjaTrader.

Would you like me to also **draw you a simple architecture diagram** (showing Polygon + NinjaTrader feeds flowing into your feature pipeline → HMM model → signals → NinjaTrader execution)? That way your engineer has a visual blueprint.


Polygon.io
(Equities/Options/FX/Crypto
REST + Websocket)
NinjaTrader Feeds
(CQG/Rithmic/etc.)
Tick + L2/L3 DOM
Optional Alt-Data
(Satellite: Planet/Maxar;
News: GDELT/Refinitiv)
Ingestion & Streaming
(Kafka/Redpanda optional)
Batch Jobs (ETL/ELT)
Storage & Catalog
(S3/Parquet + DuckDB/Athena)
Feature Store (Parquet)
Research Notebook
(Python)
Backtests & Walk-Forward
Models & Regimes
(Gaussian HMM
+ Risk Rules)
Monitoring & Logs
(Metrics, Fills, Slippage,
Model Drift)
Signal Service API
(REST gRPC)
NinjaTrader Strategy
(C#)
Orders, Stops, Risk Limits


Polygon.io
REST/WebSocket
(Equities/Options/FX/Crypto)
NinjaTrader Feeds
CQG / Rithmic
Tick + L2/L3
Optional Alt-Data
Planet/Maxar, GDELT, FRED
Ingestion Layer
Python workers (asyncio)
Optional: Redpanda/Kafka topics
Orchestration: Prefect
Storage & Catalog
Amazon S3 (raw/clean/features)
Parquet + Partitioning (date/symbol)
AWS Glue Data Catalog
Ad hoc Query
DuckDB (local)
Athena (S3 SQL)
Feature Store
Parquet tables (rolling windows)
Versioned via S3 prefixes
Modeling
Python: hmmlearn / pomegranate
Walk-forward via Prefect flows
Checkpoints to S3
Signal Service
FastAPI (REST)
Redis cache
Auth token per strategy
Monitoring
Prometheus metrics
Grafana dashboards
Alerting (Opsgenie/Email)
S3 Layout (example)
s3://everflow-data/
raw/{source}/{dt}/
clean/{asset}/{dt}/
features/{asset}/{window}/{dt}/
Execution
NinjaTrader C# AddOn
HTTP client polls FastAPI
Risk checks & order routing

Awesome—here's the concrete blueprint + starter stubs.

Download the detailed diagram

Tech picks (concise)

- **Ingestion:** Python async workers; optional Redpanda/Kafka; orchestrate with **Prefect**.
- **Storage/Catalog: S3** (Parquet), partition by date/symbol; **AWS Glue** catalog.
- **Query/Ad-hoc: DuckDB** locally; **Athena** for S3 SQL.
- **Feature store:** Versioned Parquet tables (rolling windows).
- **Modeling:** Python (**hmmlearn** / **pomegranate**); walk-forward via Prefect flows; checkpoints to S3.
- **Signal service: FastAPI** (REST) + **Redis** cache; per-strategy auth token.
- **Execution: NinjaTrader C# AddOn** polling the FastAPI endpoint; built-in risk checks.
- **Monitoring: Prometheus** metrics → **Grafana** dashboards; email/ Opsgenie alerts.

S3 layout (example)

```
s3://everflow-data/
  raw/
    polygon/{dt}/...
    ninjatrader/{dt}/...
  clean/
    ES/{yyyymmdd}/...
    NQ/{yyyymmdd}/...
  features/
    ES/60m_win20/{yyyymmdd}/...
    NQ/15m_win50/{yyyymmdd}/...
  models/
    hmm_es_60m/{timestamp}/...
```

FastAPI signal service (minimal)

```
# pip install fastapi uvicorn redis[pyaes]
from fastapi import FastAPI, Header, HTTPException
import redis, json

APP_TOKEN = "replace-me"  # rotate regularly
r = redis.Redis(host="localhost", port=6379, decode_responses=True)
app = FastAPI()

def auth(x_token: str):
```

```
    if x_token != APP_TOKEN:
        raise HTTPException(401, "bad token")

@app.get("/signal/{symbol}")
def get_signal(symbol: str, X_TOKEN: str = Header(None)):
    auth(X_TOKEN)
    key = f"signal:{symbol.upper()}"
    payload = r.get(key)
    if not payload:
        return {"symbol": symbol, "position": 0.0, "conf": 0.0, "ts": None}
    return json.loads(payload)
```

Prefect model job writes to Redis

```
# after you fit the HMM and decide next-bar position:
import redis, json, time
r = redis.Redis(host="localhost", port=6379, decode_responses=True)

def publish_signal(symbol, position, confidence):
    key = f"signal:{symbol}"
    r.setex(key, 300, json.dumps({"symbol": symbol, "position": position,
                                  "conf": confidence, "ts": int(time.time())}))
```

NinjaTrader C# (polling stub)

```
// using System.Net.Http; using System.Net.Http.Headers; using Newtonsoft.Json.Linq;
public class EverflowSignalClient {
    private readonly HttpClient _http = new HttpClient();
    private readonly string _base = "http://127.0.0.1:8000";
    private readonly string _token = "replace-me";
    public async Task<(double pos,double conf)> GetPosition(string symbol) {
        var req = new HttpRequestMessage(HttpMethod.Get, $"{_base}/signal/{symbol}");
        req.Headers.Add("X-TOKEN", _token);
        var res = await _http.SendAsync(req);
        var json = JObject.Parse(await res.Content.ReadAsStringAsync());
        return ((double)json["position"], (double)json["conf"]);
    }
}
```

Athena example queries

```
-- Create external table over Parquet bars
CREATE EXTERNAL TABLE IF NOT EXISTS bars_es_60m (
  ts timestamp, open double, high double, low double, close double, volume bigint, symbol
string
)
PARTITIONED BY (dt string)
STORED AS PARQUET
LOCATION 's3://everflow-data/clean/ES/';

MSCK REPAIR TABLE bars_es_60m;

-- Quick feature rollups
```

```
SELECT date_trunc('hour', ts) AS h,
    avg(log(close/lag(close) OVER (ORDER BY ts))) AS ret,
    stddev_samp(log(close/lag(close) OVER (ORDER BY ts)))
      OVER (ROWS BETWEEN 20 PRECEDING AND CURRENT ROW) AS roll_vol_20
FROM bars_es_60m
WHERE dt BETWEEN '2025-08-01' AND '2025-09-23';
```

Prometheus metrics you'll want

- signals_emitted_total{symbol,model}
- signal_confidence{symbol}
- orders_submitted_total{symbol}
- slippage_bps_bucket{symbol}
- daily_pnl{symbol}

Next steps (tight)

1. Stand up **FastAPI + Redis** (docker-compose is fine).
2. Add a Prefect flow that: pulls S3 → builds features → fits/loads HMM → computes next-bar position → publish_signal(...).
3. Wire the **NinjaTrader AddOn** to poll every bar and apply risk checks (max daily loss, max position, hard stop).
4. Start logging fills to S3; compute slippage and feed **Prometheus**.

Want me to package this as a tiny repo skeleton (folders, docker-compose, sample configs) so your engineer can clone and go?

Done — I packaged a working starter repo you can hand to your engineer.

Download the repo zip

((
Everflow HMM Starter

A minimal, production-minded skeleton for a **regime-detection trading system** using **Polygon + NinjaTrader** data, with **FastAPI** for signals, **Prefect** for orchestration, **Redis** for short-lived signal cache, and **Prometheus/Grafana** for monitoring.

Quick start

Requirements: Docker + docker-compose.

```bash
cp .env.example .env
docker compose up --build
```

Services:
- **api**: FastAPI on `http://localhost:8000`
- **redis**: key-value cache for `{ "signal:SYMBOL": {position, conf, ts} }`
- **worker**: Prefect job that computes signals and publishes to Redis
- **prometheus**: metrics at `http://localhost:9090`
- **grafana**: dashboards at `http://localhost:3000` (login: admin / admin)

S3 Layout (example)

```
s3://everflow-data/
  raw/{source}/{dt}/...
  clean/{asset}/{dt}/...
  features/{asset}/{window}/{dt}/...
  models/{model}/{timestamp}/...
```

NinjaTrader Integration

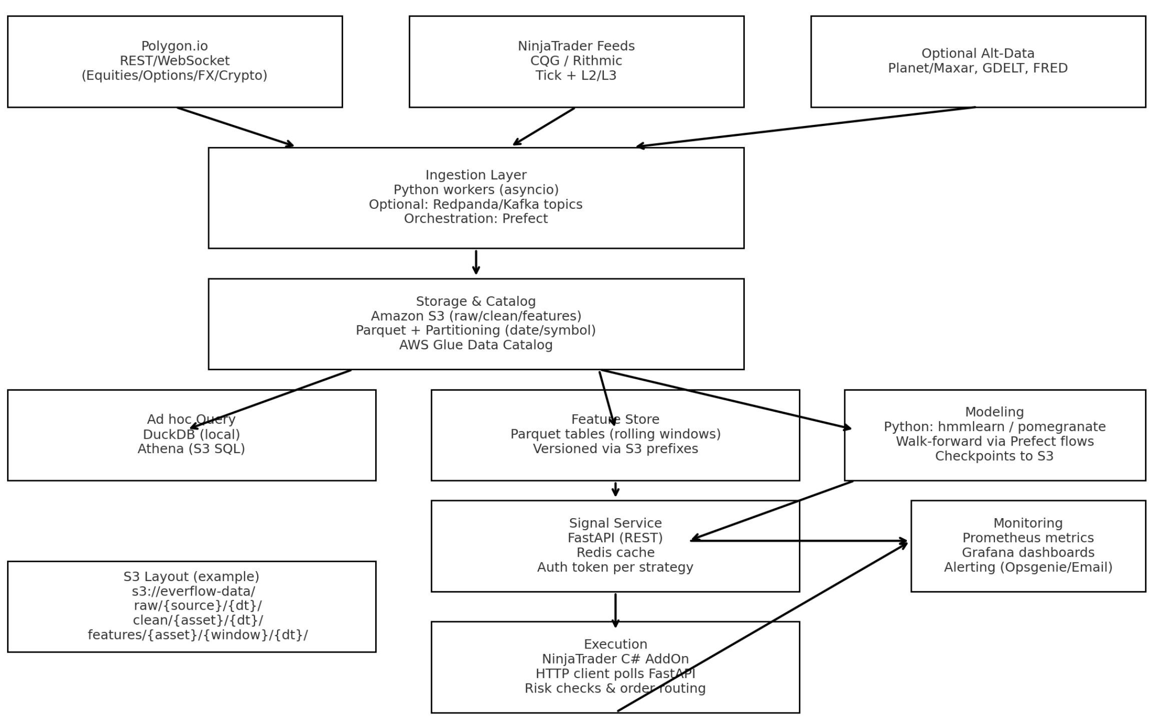
See `ninjatrader_stub/EverflowSignalClient.cs` for a simple polling client (C#).

Notes

- This is a **starter**: plug in your feature engineering and HMM training code in `worker/flow.py`.
- Replace toy logic with your real walk-forward + cost/slippage model.
- Add IAM creds for S3 in `.env` or use local testing (files in `data/examples`).

Generated on: 2025-09-23T19:40:03.360085Z
))

Also handy: the detailed diagram you asked for → PNG ((



))

What's inside

- docker-compose.yml — spins up **FastAPI (signals)**, **Redis**, **Prefect worker**, **Prometheus**, **Grafana**.
- api/ — FastAPI app with /signal/{symbol} GET/POST, token auth, Prometheus metrics.
- worker/ — Prefect flow that loads example bars, builds features, fits a **Gaussian HMM**, picks a position, and publishes to Redis.
- ninjatrader_stub/ — C# client (EverflowSignalClient.cs) that polls the API and returns (position, confidence).
- sql/ — Athena table + query examples for S3 Parquet.
- infra/prometheus.yml — basic scraping for API/worker.
- data/examples/es_60m.csv — small sample file; replace with your real data.
- .env.example — copy to .env and set APP_TOKEN, S3 creds if needed.

To run

```
unzip everflow-hmm-starter.zip
cd everflow-hmm-starter
cp .env.example .env   # set APP_TOKEN
docker compose up --build
```

- API: http://localhost:8000/docs
- Prometheus: http://localhost:9090
- Grafana: http://localhost:3000 (admin/admin)

Want me to tailor the worker to pull **Polygon** directly (REST/websocket) and add a **NinjaTrader CSV importer** next?