Distributed Stock Price Prediction Using Mean Reversion Signals

Refined Architecture Document

Cloud-Native Trading System

November 9, 2025

1 System Overview

1.1 Design Principles

- **Serverless-first**: Minimize operational overhead and cost by using managed services
- **Separation of concerns**: Batch analytics (Phase 1) decoupled from real-time monitoring (Phase 2)
- **Cost optimization**: Use Cloud Functions/Cloud Run instead of persistent VMs where possible
- **Scalability**: Leverage BigQuery's MPP architecture for parallel processing
- **Low latency**: Direct indicator computation without unnecessary data serialization

1.2 Key Components

1. **Cloud Scheduler**: Triggers daily batch screening workflow
2. **Cloud Function/Run**: Serverless compute for data ingestion
3. **BigQuery**: Analytical warehouse with UDFs (ADF, Hurst, Variance Ratio) – ADF implemented as SQL UDF (approximate) rather than remote function
4. **Firestore/Redis**: Low-latency watchlist storage
5. **Cloud Run/VM**: Real-time monitoring service
6. **Pub/Sub (Prices)**: Serverless bus for 1m/5m price bars (per-symbol ordering keys)
7. **Pub/Sub (Signals)**: Event bus for downstream trading signals
8. **Alpha Vantage/Polygon.io**: Real-time market data provider

2 Architecture Diagram

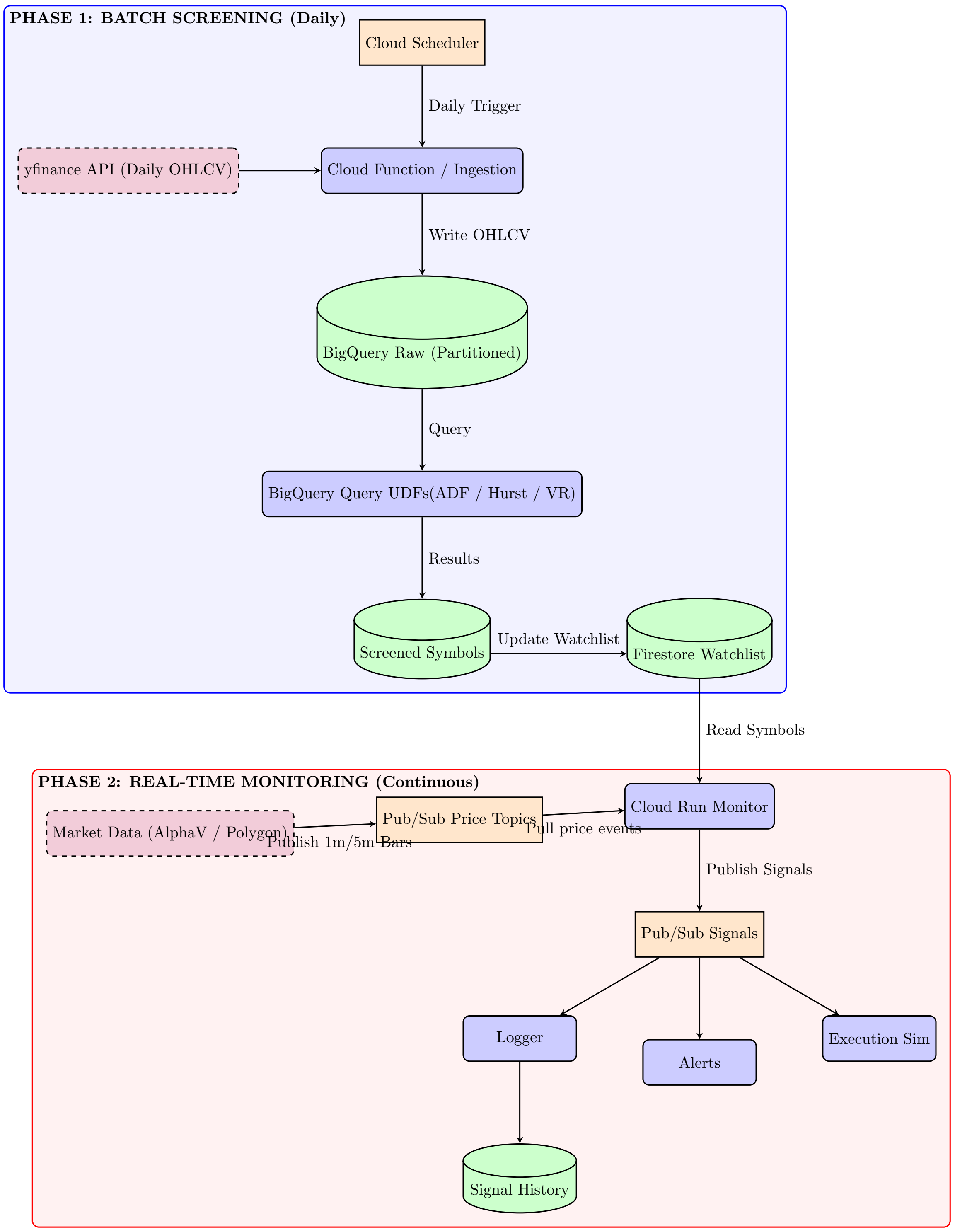

Figure 1: Two-Phase Cloud-Native Trading Architecture

3 Phase 1: Batch Screening (Daily)

3.1 Workflow Description

Phase 1 executes once daily to identify mean-reverting stocks from the entire universe. This phase prioritizes thoroughness and statistical rigor over speed.

3.2 Step-by-Step Process

3.2.1 Step 1: Scheduled Trigger

Component: Cloud Scheduler

- Triggers daily at market close (e.g., 4:30 PM ET)
- Invokes Cloud Function via HTTP POST

3.2.2 Step 2: Data Ingestion

Component: Cloud Function (Python 3.11)
Responsibilities:

- Fetch daily OHLCV data for entire universe (e.g., S&P 500, Russell 3000)
- Use `yfinance` library with batch requests
- Write directly to BigQuery using streaming inserts or load jobs
- Handle retries and logging

Why Cloud Function instead of VM?

- No idle costs (only pay for execution time)
- Auto-scaling and fault tolerance built-in
- Typical execution: 2-5 minutes/day

Alternative: Cloud Run for longer execution times (up to 60 minutes vs. 9 minutes for Cloud Functions)

3.2.3 Step 3: Statistical Analysis

Component: BigQuery with User-Defined Functions (UDFs)
BigQuery executes a single analytical query that:

1. Computes rolling windows (20, 50, 200 days) using window functions
2. Calculates mean reversion statistics via custom UDFs:
   - **Augmented Dickey-Fuller (ADF) test**: Tests for unit root (stationarity)
   - **Hurst exponent**: Measures mean reversion ($H < 0.5$) vs. trending ($H > 0.5$)
   - **Variance Ratio (VR) test**: Detects deviations from random walk
3. Applies filters (liquidity, bid-ask spread, ADV constraints)
4. Ranks candidates by composite score

UDF Implementation Options:

- **Option A**: JavaScript UDF (runs in BigQuery, fastest but limited libraries)
- **Option B**: Python Remote Function (calls Cloud Function/Run) — not used in this project; we standardize on UDFs (including an approximate ADF UDF)

extbfRecommended: Implement ADF, Hurst, and Variance Ratio as BigQuery UDFs. For ADF, use the provided approximate UDF for screening; reserve precise Python stats for offline backtesting if needed.

3.2.4 Step 4: Results Storage

Components: BigQuery (screened symbols table) + Firestore (watchlist)

- BigQuery table stores full screening results with metadata:
  - Symbol, screening date, ADF p-value, Hurst exponent, VR statistic
  - Computed thresholds (Bollinger Bands, z-score levels)
  - Liquidity metrics (ADV, spread)
- Firestore document stores lightweight watchlist for real-time service:
  - Just symbols and trigger conditions
  - Updated via Cloud Function trigger or direct BigQuery export

3.3 Data Schema

3.3.1 BigQuery Raw Data Table

```sql
CREATE TABLE `project.dataset.ohlcv_daily`
(
  symbol STRING NOT NULL,
  date DATE NOT NULL,
  open FLOAT64,
  high FLOAT64,
  low FLOAT64,
  close FLOAT64,
  volume INT64,
  adj_close FLOAT64
)
PARTITION BY date
CLUSTER BY symbol
OPTIONS(
  partition_expiration_days=1095,  -- 3 years retention
  require_partition_filter=TRUE
);
```

Listing 1: Raw OHLCV Schema (Partitioned by date)

3.3.2 BigQuery Screened Symbols Table

```sql
CREATE TABLE `project.dataset.screened_symbols`
(
  screening_date DATE NOT NULL,
  symbol STRING NOT NULL,
  adf_statistic FLOAT64,
  adf_pvalue FLOAT64,
  hurst_exponent FLOAT64,
  vr_statistic FLOAT64,
  mean_reversion_score FLOAT64,
  bb_upper FLOAT64,
  bb_lower FLOAT64,
  bb_period INT64,
  avg_volume FLOAT64,
  avg_spread FLOAT64
)
PARTITION BY screening_date
CLUSTER BY mean_reversion_score;
```

Listing 2: Screening Results Schema

3.3.3 Firestore Watchlist Document

```
{
  "updated_at": "2025-11-09T16:30:00Z",
  "symbols": [
    {
      "symbol": "AAPL",
      "bb_lower": 148.50,
      "bb_upper": 152.30,
      "rsi_oversold": 30,
      "rsi_overbought": 70,
      "hurst": 0.42
    },
    // ... more symbols
  ]
}
```

Listing 3: Firestore Document Structure

4 Phase 2: Real-Time Monitoring (Continuous)

4.1 Workflow Description

Phase 2 continuously monitors the screened watchlist, fetches intraday prices, computes technical indicators locally, and publishes trading signals when conditions are met.

4.2 Step-by-Step Process

4.2.1 Step 1: Service Initialization

Component: Cloud Run or Compute Engine VM

The monitoring service starts and:

1. Reads the watchlist from Firestore
2. Establishes connection to market data provider
3. Initializes indicator calculation state (e.g., MACD buffers)
4. Subscribes to Pub/Sub for configuration updates (optional)

Cloud Run vs. VM Decision:

- **Cloud Run**: Better for cost if market hours only (6.5 hrs/day)
- **VM (e2-micro)**: Better if running 24/7 or need persistent state
- **Recommendation**: Start with Cloud Run, migrate to VM if needed

4.2.2 Step 2: Market Data Ingestion via Pub/Sub

extbfData Source: Alpha Vantage, Polygon.io, or IEX Cloud ⇒ Pub/Sub Price Topics

Why not yfinance for real-time?

- yfinance is designed for historical data only
- Rate limits and unreliable for sub-hourly intervals
- No official API; scrapes Yahoo Finance (terms of service violations)

extbfIngestion Pattern

- **Producer**: Cloud Run service (or lightweight container) polls market data provider APIs and publishes normalized OHLCV bars to Pub/Sub topics (`prices-1m`, `prices-5m`) using ordering keys per symbol (e.g., `AAPL`)
- **Topics**: Message ordering enabled; acknowledgement deadline tuned (e.g., 30s). Retention for unacknowledged messages max 7 days.
- **Consumer**: Monitoring service creates one subscription per interval or a single subscription with filtering; processes only watched symbols from Firestore
- **Backpressure**: Autoscaling Cloud Run instances + Pub/Sub flow control (max messages in-flight) ensure bounded memory; explicit acks after processing

4.2.3 Step 3: Indicator Computation

Local Processing (No Pub/Sub Yet!)
For each price update, compute:

- **MACD** (Moving Average Convergence Divergence): Trend momentum
- **RSI** (Relative Strength Index): Overbought/oversold
- **ADX** (Average Directional Index): Trend strength
- **Bollinger Bands**: Volatility and reversion bands
- **Z-score**: Current price vs. rolling mean

Key Design Point: Indicators are computed *where data is fetched*. Do NOT publish raw prices to Pub/Sub and compute elsewhere—this adds latency and complexity.

4.2.4 Step 4: Signal Logic

Apply rules to generate trading signals:

```python
def check_buy_signal(symbol, price, indicators):
    """
    Buy signal for mean reversion entry
    """
    bb_lower = indicators['bb_lower']
    rsi = indicators['rsi']
    z_score = indicators['z_score']

    conditions = [
        price <= bb_lower,            # Below lower band
        rsi < 30,                     # Oversold
        z_score < -2.0,               # 2 std devs below mean
        indicators['adx'] < 25        # Weak trend (good for MR)
    ]

    if all(conditions):
        return {
            'action': 'BUY',
            'symbol': symbol,
            'price': price,
            'confidence': 0.85,
            'reason': 'Mean reversion setup'
        }
    return None
```

Listing 4: Example Signal Logic

4.2.5 Step 5: Signal Publication

Component: Pub/Sub Topic
When a signal is generated:

1. Serialize signal to JSON
2. Publish to `trading-signals` Pub/Sub topic
3. Include timestamp, symbol, action, price, confidence, metadata

Message Schema:

```json
{
  "timestamp": "2025-11-09T14:35:22.123Z",
  "symbol": "AAPL",
  "action": "BUY",
  "side": "LONG",
  "price": 148.75,
  "confidence": 0.85,
  "indicators": {
    "rsi": 28.5,
    "z_score": -2.15,
    "bb_lower": 148.50
  },
  "reason": "Mean reversion entry",
  "strategy": "bollinger_rsi_mr_v1"
}
```

Listing 5: Pub/Sub Signal Message

4.2.6 Step 6: Signal Consumption

Subscribers: Multiple services consume signals from Pub/Sub

1. **Logger Service**: Writes all signals to BigQuery for backtesting and analysis
2. **Alert Service**: Sends notifications (email, Slack, SMS) to traders
3. **Execution Simulator**: Paper trading service that simulates order fills
4. **Risk Monitor**: Checks position limits, exposure constraints
5. **Dashboard**: Real-time web UI showing active signals

4.3 Implementation Details

4.3.1 Monitoring Service Structure

```python
import json
import time
from collections import deque, defaultdict
from google.cloud import firestore, pubsub_v1
import pandas as pd
import ta  # Technical Analysis library

PRICE_WINDOW = 120  # keep last 120 bars (~2h for 1m)

class TradingMonitor:
  def __init__(self):
    self.db = firestore.Client()
    self.publisher = pubsub_v1.PublisherClient()
    self.signal_topic = self.publisher.topic_path('PROJECT', 'trading-signals')
    self.watchlist = set(self.load_watchlist())
    self.price_windows = defaultdict(lambda: deque(maxlen=PRICE_WINDOW))
    self.subscriber = pubsub_v1.SubscriberClient()
    self.subscription_1m = self.subscriber.subscription_path('PROJECT', 'prices
    -1m-sub')
    self.subscription_5m = self.subscriber.subscription_path('PROJECT', 'prices
    -5m-sub')

  def load_watchlist(self):
    doc = self.db.collection('config').document('watchlist').get()
    return doc.to_dict().get('symbols', [])

  def compute_indicators(self, symbol):
    window = list(self.price_windows[symbol])
    if len(window) < 25:  # not enough data yet
      return None
    df = pd.DataFrame(window)
    macd = ta.trend.MACD(df['close'])
    rsi = ta.momentum.RSIIndicator(df['close'], window=14)
    bb = ta.volatility.BollingerBands(df['close'])
    z_score = (df['close'].iloc[-1] - df['close'].rolling(20).mean().iloc[-1]) / df['close'].rolling(20).std().iloc[-1]
    return {
      'macd': macd.macd().iloc[-1],
      'rsi': rsi.rsi().iloc[-1],
      'bb_lower': bb.bollinger_lband().iloc[-1],
      'bb_upper': bb.bollinger_hband().iloc[-1],
      'z_score': z_score
    }

  def generate_signal(self, symbol, price, indicators):
    # Placeholder: implement strategy rules
    if indicators and indicators['rsi'] < 30 and price <= indicators['bb_lower'] and indicators['z_score'] < -2:
      return {
        'symbol': symbol,
        'action': 'BUY',
        'timestamp': int(time.time()),
        'price': price,
        'strategy': 'mr_v1'
      }
    return None

  def publish_signal(self, signal):
    data = json.dumps(signal).encode('utf-8')
    self.publisher.publish(self.signal_topic, data, ordering_key=signal['symbol'])
    print(f"Published signal: {signal}")

  def handle_message(self, message):
    try:
      event = json.loads(message.data.decode('utf-8'))
      symbol = event.get('symbol')
      if symbol not in self.watchlist:
        message.ack()
        return
      bar = event.get('bar')  # {'open':..., 'high':..., 'low':..., 'close':..., 'volume':...}
      if not bar:
        message.ack()
        return
      self.price_windows[symbol].append(bar)
      indicators = self.compute_indicators(symbol)
      if indicators:
        signal = self.generate_signal(symbol, bar['close'], indicators)
        if signal:
          self.publish_signal(signal)
      message.ack()
    except Exception as e:
      print(f"Error processing message: {e}")
      message.nack()

  def run(self):
    flow_control = pubsub_v1.types.FlowControl(max_messages=100, max_bytes
    =10*1024*1024)
    streaming_pull_future_1m = self.subscriber.subscribe(
      self.subscription_1m, callback=self.handle_message, flow_control=
    flow_control
    )
    streaming_pull_future_5m = self.subscriber.subscribe(
      self.subscription_5m, callback=self.handle_message, flow_control=
    flow_control
    )
    print("Monitoring service started. Listening for price events...")
    try:
      streaming_pull_future_1m.result()
      streaming_pull_future_5m.result()
    except KeyboardInterrupt:
      streaming_pull_future_1m.cancel()
      streaming_pull_future_5m.cancel()

if __name__ == '__main__':
  TradingMonitor().run()
```

Listing 6: Real-Time Monitoring Service (Pub/Sub Subscriber)

5 Statistical Methods Implementation

5.1 Augmented Dickey-Fuller (ADF) Test UDF

5.1.1 Approximate In-SQL Implementation

While a full ADF test involves regression with lagged differences and can be computationally intensive, a simplified approximation can be implemented as a BigQuery SQL UDF for screening purposes. We estimate the test statistic using first differences and a basic AR(1) regression without deterministic trend terms.

extbfNull Hypothesis: Unit root (non-stationary)
extbfAlternative: Stationary (mean-reverting)

Simplified statistic (approx.):

$$\phi = \frac{\sum_{t=2}^{n}(y_t - y_{t-1})y_{t-1}}{\sum_{t=2}^{n} y_{t-1}^2}$$

We derive an approximate p-value via a heuristic mapping (not exact MacKinnon critical values). This is acceptable for coarse pre-filtering; exact statistical verification can be deferred to offline Python backtests.

```sql
CREATE OR REPLACE FUNCTION `project.dataset.adf_test`(prices ARRAY<FLOAT64>)
RETURNS STRUCT<adf_stat FLOAT64, p_value FLOAT64>
AS (
  WITH indexed AS (
    SELECT
      OFFSET + 1 AS t,
      price,
      LAG(price) OVER (ORDER BY OFFSET) AS prev_price
    FROM UNNEST(prices) AS price WITH OFFSET
  ), diffs AS (
    SELECT t, price, prev_price, price - prev_price AS diff
    FROM indexed
    WHERE prev_price IS NOT NULL
  ), components AS (
    SELECT
      SUM(diff * prev_price) AS num,
      SUM(prev_price * prev_price) AS denom,
      COUNT(*) AS n
    FROM diffs
  ), stat AS (
    SELECT
      SAFE_DIVIDE(num, denom) AS adf_stat,
      -- Heuristic p-value: lower phi (more negative) => more mean reversion
      CASE
        WHEN SAFE_DIVIDE(num, denom) < -0.25 THEN 0.01
        WHEN SAFE_DIVIDE(num, denom) < -0.15 THEN 0.05
        WHEN SAFE_DIVIDE(num, denom) < -0.08 THEN 0.10
        ELSE 0.30
      END AS p_value
    FROM components
  )
  SELECT STRUCT(adf_stat, p_value) FROM stat
);
```

Listing 7: Approximate ADF UDF in BigQuery

extbfLimitations: This approximation does not replace the full ADF procedure (no lag selection, no deterministic trend modeling, no MacKinnon critical values). It functions as a lightweight filter to narrow candidates before deeper Python statistical validation.

5.2 Hurst Exponent

5.2.1 Theory

The Hurst exponent H characterizes the long-term memory of a time series:

- $H < 0.5$: Anti-persistent (mean-reverting)
- $H = 0.5$: Random walk (Brownian motion)
- $H > 0.5$: Persistent (trending)

Calculated using rescaled range (R/S) analysis:

$$H = \frac{\log(R/S)}{\log(n)}$$

5.2.2 JavaScript UDF Implementation

```
CREATE OR REPLACE FUNCTION `project.dataset.hurst_exponent`(prices ARRAY<
    FLOAT64>)
RETURNS FLOAT64
LANGUAGE js AS r"""
  if (prices.length < 20) return null;

  // Calculate log returns
  let returns = [];
  for (let i = 1; i < prices.length; i++) {
    returns.push(Math.log(prices[i] / prices[i-1]));
  }

  // Mean return
  const mean = returns.reduce((a,b) => a+b) / returns.length;

  // Cumulative deviations
  let cumdev = [0];
  for (let i = 0; i < returns.length; i++) {
    cumdev.push(cumdev[i] + (returns[i] - mean));
  }

  // Range
  const R = Math.max(...cumdev) - Math.min(...cumdev);

  // Standard deviation
  const variance = returns.reduce((sum, r) =>
    sum + Math.pow(r - mean, 2), 0) / returns.length;
  const S = Math.sqrt(variance);

  // Hurst exponent
  if (S === 0 || R === 0) return 0.5;
  const H = Math.log(R/S) / Math.log(returns.length);

  return H;
""";
```

Listing 8: Hurst Exponent UDF in BigQuery

5.3 Variance Ratio Test

5.3.1 Theory

The VR test compares variance of returns at different horizons. For a random walk:

$$VR(k) = \frac{\mathrm{Var}(r_t + r_{t-1} + \cdots + r_{t-k+1})}{k \cdot \mathrm{Var}(r_t)} \approx 1$$

- $VR(k) < 1$: Mean reversion
- $VR(k) > 1$: Momentum/trending

5.3.2 BigQuery UDF Implementation

```
CREATE OR REPLACE FUNCTION `project.dataset.variance_ratio`(
  prices ARRAY<FLOAT64>,
  lag INT64
)
RETURNS FLOAT64
LANGUAGE js AS r"""
  if (prices.length < lag * 2) return null;

  // Calculate returns
  let returns = [];
  for (let i = 1; i < prices.length; i++) {
    returns.push(Math.log(prices[i] / prices[i-1]));
  }

  // 1-period variance
  const mean1 = returns.reduce((a,b) => a+b) / returns.length;
  const var1 = returns.reduce((sum, r) =>
    sum + Math.pow(r - mean1, 2), 0) / returns.length;

  // k-period returns
  let kReturns = [];
  for (let i = lag; i < returns.length; i++) {
    let sum = 0;
    for (let j = 0; j < lag; j++) {
      sum += returns[i-j];
    }
    kReturns.push(sum);
  }

  // k-period variance
  const meank = kReturns.reduce((a,b) => a+b) / kReturns.length;
  const vark = kReturns.reduce((sum, r) =>
    sum + Math.pow(r - meank, 2), 0) / kReturns.length;

  // Variance ratio
  if (var1 === 0) return 1.0;
  return vark / (lag * var1);
""";
```

Listing 9: Variance Ratio UDF

5.4 Complete Screening Query

```
WITH price_data AS (
  SELECT
    symbol,
```

```sql
    date,
    adj_close,
    volume,
    -- Rolling windows
    AVG(adj_close) OVER w20 as ma_20,
    STDDEV(adj_close) OVER w20 as std_20,
    AVG(adj_close) OVER w50 as ma_50,
    AVG(volume) OVER w20 as avg_volume_20
  FROM `project.dataset.ohlcv_daily`
  WHERE date >= DATE_SUB(CURRENT_DATE(), INTERVAL 250 DAY)
  WINDOW
    w20 AS (PARTITION BY symbol ORDER BY date ROWS BETWEEN 19 PRECEDING AND
   CURRENT ROW),
    w50 AS (PARTITION BY symbol ORDER BY date ROWS BETWEEN 49 PRECEDING AND
   CURRENT ROW)
),

statistical_tests AS (
  SELECT
    symbol,
    MAX(date) as latest_date,
    -- Collect price arrays for statistical tests
    ARRAY_AGG(adj_close ORDER BY date) as price_array,
    AVG(avg_volume_20) as avg_volume,

    -- Bollinger Bands (latest values)
    ARRAY_AGG(ma_20 ORDER BY date DESC LIMIT 1)[OFFSET(0)] +
      2 * ARRAY_AGG(std_20 ORDER BY date DESC LIMIT 1)[OFFSET(0)] as bb_upper,
    ARRAY_AGG(ma_20 ORDER BY date DESC LIMIT 1)[OFFSET(0)] -
      2 * ARRAY_AGG(std_20 ORDER BY date DESC LIMIT 1)[OFFSET(0)] as bb_lower,

    -- Current price
    ARRAY_AGG(adj_close ORDER BY date DESC LIMIT 1)[OFFSET(0)] as current_price
  FROM price_data
  GROUP BY symbol
  HAVING COUNT(*) >= 100  -- Minimum data points
),

scored_symbols AS (
  SELECT
    symbol,
    latest_date,
    current_price,
    avg_volume,
    bb_upper,
    bb_lower,

    -- Statistical tests
    `project.dataset.hurst_exponent`(price_array) as hurst,
    `project.dataset.variance_ratio`(price_array, 5) as vr_5,
    `project.dataset.variance_ratio`(price_array, 10) as vr_10,
    `project.dataset.adf_test`(price_array).adf_stat as adf_stat,
    `project.dataset.adf_test`(price_array).p_value as adf_pvalue
  FROM statistical_tests
)

SELECT
  symbol,
  latest_date as screening_date,
  current_price,
  hurst,
  vr_5,
  vr_10,
  adf_stat,
  adf_pvalue,
  bb_upper,
  bb_lower,
  avg_volume,

  -- Composite mean reversion score (0-100)
  CASE
    WHEN hurst < 0.5 AND adf_pvalue < 0.05 AND vr_5 < 1.0 THEN
      (0.5 - hurst) * 50 +  -- Hurst component (0-25)
      (1.0 - vr_5) * 30 +   -- VR component (0-30)
      (0.05 - adf_pvalue) * 200  -- ADF component (0-10)
    ELSE 0
  END as mean_reversion_score

FROM scored_symbols
WHERE
  -- Filters
  hurst IS NOT NULL
  AND hurst < 0.5  -- Mean-reverting
  AND adf_pvalue < 0.05  -- Statistically significant stationarity
  AND vr_5 < 1.0  -- Mean reversion at 5-day horizon
  AND avg_volume > 1000000  -- Minimum liquidity

ORDER BY mean_reversion_score DESC
LIMIT 50;
```

Listing 10: Full BigQuery Screening Query with Statistical Tests

6 Technical Indicator Computation

6.1 Moving Average Convergence Divergence (MACD)

Formula:

$$\begin{aligned}\text{MACD Line} &= \text{EMA}_{12}(\text{Close}) - \text{EMA}_{26}(\text{Close})\\ \text{Signal Line} &= \text{EMA}_{9}(\text{MACD Line})\\ \text{Histogram} &= \text{MACD Line} - \text{Signal Line}\end{aligned}$$

Trading Signals:

- Bullish: MACD crosses above signal line
- Bearish: MACD crosses below signal line
- Divergence: Price makes new low but MACD doesn't (bullish reversal)

6.2 Relative Strength Index (RSI)

Formula: $\text{RSI} = 100 - \frac{100}{1+\text{RS}}$ where $\text{RS} = \frac{\text{Average Gain}}{\text{Average Loss}}$ over 14 periods.

Trading Signals:

- RSI ¡ 30: Oversold (potential buy)
- RSI ¿ 70: Overbought (potential sell)
- For mean reversion: Look for extreme RSI values

6.3 Average Directional Index (ADX)

Purpose: Measures trend strength (not direction)

Interpretation:

- ADX ¡ 20: Weak trend (good for mean reversion strategies)
- ADX 20-40: Moderate trend
- ADX ¿ 40: Strong trend (avoid mean reversion)

6.4 Bollinger Bands

Formula:

$$\begin{aligned}\text{Middle Band} &= \text{SMA}_{20}(\text{Close})\\ \text{Upper Band} &= \text{Middle Band} + 2 \times \sigma_{20}\\ \text{Lower Band} &= \text{Middle Band} - 2 \times \sigma_{20}\end{aligned}$$

Mean Reversion Logic:

- Price touches/breaks lower band: Potential buy (oversold)
- Price touches/breaks upper band: Potential sell (overbought)
- Expected reversion to middle band

7 Deployment and Operations

7.1 Deployment Checklist

7.1.1 Phase 1 Deployment

1. Create BigQuery dataset and tables
2. Deploy UDFs (ADF, Hurst, VR)
3. Deploy data ingestion Cloud Function
4. Set up Cloud Scheduler job
5. Configure Firestore database
6. Test end-to-end batch workflow
7. Set up BigQuery cost alerts

7.1.2 Phase 2 Deployment

1. Create Pub/Sub topics and subscriptions
2. Deploy monitoring service to Cloud Run
3. Configure market data API credentials
4. Deploy subscriber services (logger, alerts, simulator)
5. Set up monitoring dashboards
6. Configure alerting for service failures
7. Test signal generation with paper trading

7.2 Monitoring and Observability

7.2.1 Key Metrics

Metric	**Target**	**Alert Condition**
Data ingestion success rate	$> 99\%$	<95% in 24h window
BigQuery job duration	<5 min	>10 min
Screening results count	20-100	<10 or >200 symbols
Signal latency (fetch→publish)	<5 sec	>15 sec
Pub/Sub publish errors	0	>5 in 1h
Market data API errors	<1%	>5% in 1h
Cloud Run CPU utilization	<70%	>85%
Pub/Sub subscription backlog (msgs)	<100	>1000 for 5 min
Pub/Sub oldest unacked age	<30s	>120s sustained

Table 1: Operational Metrics and Alerts

7.2.2 Logging Strategy

- **Structured logging**: Use JSON format with consistent fields
- **Log levels**: ERROR for failures, INFO for signals, DEBUG for detailed tracing
- **Centralized logs**: All services log to Cloud Logging
- **Log-based metrics**: Create metrics from log patterns (e.g., signal count/hour)

7.3 Error Handling and Resilience

7.3.1 Data Ingestion

- Retry failed symbol fetches up to 3 times with exponential backoff
- Log failed symbols to separate table for manual review
- Continue processing remaining symbols on partial failure
- Send alert if ¿10% of symbols fail

7.3.2 Market Data Feed

- Implement circuit breaker pattern for API failures
- Cache last known prices (TTL: 5 minutes)
- Fall back to cached data if API unavailable
- Automatically switch to backup data provider if configured

7.3.3 Signal Generation

- Validate all inputs before indicator calculation
- Catch and log exceptions per symbol (don't crash entire service)
- Implement signal deduplication (don't re-emit same signal)
- Add signal expiry timestamp (signals older than 5 min are stale)

7.4 Security Considerations

- **API Keys**: Store in Secret Manager, rotate quarterly
- **IAM**: Principle of least privilege for all service accounts
- **Network**: Use VPC for VM/Cloud Run, restrict egress
- **Data**: Encrypt BigQuery tables (automatic), enable audit logging
- **Pub/Sub**: Enable authentication, use service accounts for subscribers

8 Performance Optimization

8.1 BigQuery Optimization

8.1.1 Query Performance

- **Partitioning**: Always partition by date
- **Clustering**: Cluster by symbol for better pruning
- **Partition filters**: Always include date filter in WHERE clause
- **Avoid SELECT ***: Select only needed columns
- **Materialized views**: Consider for frequently accessed aggregations

8.2 Real-Time Service Optimization

- **Batch API calls**: Request multiple symbols per API call
- **Caching**: Cache indicator buffers in memory
- **Async processing**: Use asyncio for concurrent symbol processing
- **Connection pooling**: Reuse HTTP connections
- **Smart polling**: Increase frequency during market hours only

9 Testing Strategy

9.1 Unit Tests

- Statistical functions (ADF, Hurst, VR) against known datasets
- Indicator calculations (MACD, RSI, ADX) against TA-Lib reference
- Signal logic with synthetic price data
- Data validation and error handling

9.2 Integration Tests

- End-to-end batch workflow (Cloud Function → BigQuery → Firestore)
- Pub/Sub message flow (publish → subscribe → acknowledge)
- Market data API integration with retries
- Database read/write operations

9.3 Backtesting

- Out-of-sample validation on historical data (2020-2024)
- Walk-forward analysis with rolling windows
- Transaction cost sensitivity analysis
- Parameter optimization with cross-validation
- Stress testing in high-volatility periods (COVID-19, 2022 drawdown)

10 Risk Management

10.1 Trading Risk Controls

```python
class RiskManager:
    def __init__(self):
        self.max_position_size = 10000  # USD per symbol
        self.max_portfolio_size = 50000  # Total USD
        self.max_daily_loss = 1000  # USD
        self.max_symbols = 10  # Concurrent positions

    def check_risk_limits(self, signal, current_positions, pnl_today):
        """
        Validate signal against risk limits
        Returns: (allowed: bool, reason: str)
        """
        # Check daily loss limit
        if pnl_today < -self.max_daily_loss:
            return False, "Daily loss limit exceeded"

        # Check max symbols
        if len(current_positions) >= self.max_symbols:
            if signal['symbol'] not in current_positions:
                return False, "Max concurrent positions reached"

        # Check position size
        position_value = signal['price'] * signal.get('quantity', 100)
        if position_value > self.max_position_size:
            return False, "Position size exceeds limit"

        # Check portfolio exposure
        total_exposure = sum(p['value'] for p in current_positions.values())
        if total_exposure + position_value > self.max_portfolio_size:
            return False, "Portfolio size limit exceeded"

        return True, "OK"
```

Listing 11: Example Risk Management Rules

10.2 Operational Risk

- **Data quality**: Validate prices for anomalies (>20% daily change)
- **Look-ahead bias**: Ensure all calculations use only past data
- **Survivorship bias**: Include delisted stocks in backtesting
- **Latency risk**: Time-stamp all events, alert on ¿10s latency
- **Disaster recovery**: Daily BigQuery exports to Cloud Storage

11 Conclusion

This architecture provides a robust, scalable, and cost-effective foundation for mean reversion trading. Key advantages:

1. **Separation of concerns**: Batch analytics decoupled from real-time execution
2. **Scalability**: BigQuery handles millions of rows, Cloud Run auto-scales
3. **Flexibility**: Easy to add new indicators, strategies, or data sources
4. **Observability**: Comprehensive logging, monitoring, and alerting

The two-phase design allows for thorough daily screening using computationally expensive statistical tests, while maintaining low-latency signal generation for time-sensitive trading decisions. The system is production-ready for paper trading and can be extended to live execution with appropriate broker integration.

12 Appendix: Configuration Examples

12.1 Cloud Scheduler Configuration

```
gcloud scheduler jobs create http daily-screening \
  --schedule="30 16 * * 1-5" \
  --time-zone="America/New_York" \
  --uri="https://us-central1-PROJECT.cloudfunctions.net/ingest-daily-data" \
  --http-method=POST \
  --headers="Content-Type=application/json" \
  --message-body='{"universe": "SP500"}'
```

12.2 Firestore Security Rules

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /config/watchlist {
      // Only allow service accounts to write
      allow read: if request.auth != null;
      allow write: if request.auth.token.email.matches('.*@.*iam.
    gserviceaccount.com');
    }
  }
}
```

12.3 Pub/Sub Subscription Configuration

```
gcloud pubsub topics create trading-signals

gcloud pubsub subscriptions create logger-subscription \
  --topic=trading-signals \
  --ack-deadline=30 \
  --message-retention-duration=7d \
  --expiration-period=never

gcloud pubsub subscriptions create alert-subscription \
  --topic=trading-signals \
  --push-endpoint="https://alerts.example.com/webhook"
```

12.4 Environment Variables

```bash
# Phase 1 (Data Ingestion)
export GCP_PROJECT="your-project-id"
export BQ_DATASET="trading_data"
export BQ_TABLE="ohlcv_daily"
export UNIVERSE="SP500"

# Phase 2 (Real-Time Monitoring)
export MARKET_DATA_API_KEY="your-alpha-vantage-key"
export MARKET_DATA_PROVIDER="alphavantage"
export PUBSUB_TOPIC="trading-signals"
export FIRESTORE_COLLECTION="config"
export POLL_INTERVAL_SECONDS=60
export MAX_SYMBOLS=50
export PRICES_TOPIC_1M="prices-1m"
export PRICES_TOPIC_5M="prices-5m"
export PRICES_SUBSCRIPTION_1M="prices-1m-sub"
export PRICES_SUBSCRIPTION_5M="prices-5m-sub"
export USE_ORDERING_KEYS="true"
```