# GlobalPulse MVP - Product Requirements Document

\*\*Version\*\*: 1.0

\*\*Date\*\*: August 27, 2025

\*\*Status\*\*: Draft - Phase Gate Ready

\*\*Document Type\*\*: Single Source of Truth for Autonomous Implementation

---

## 3.1 Executive Summary

### Purpose, Scope & MVP Boundaries

\*\*Purpose\*\*: GlobalPulse is a model-agnostic macroeconomic forecasting and LLM evaluation platform that provides neutral, reproducible benchmarks for comparing how different AI models forecast national economic indicators.

\*\*Core Value Proposition\*\*:

- \*\*Model-Agnostic\*\*: Evaluate any LLM or classical baseline using identical ground truth and rules

- \*\*Governed & Vintaged\*\*: First-release vs latest-vintage scoring with immutable audit trails

- \*\*Automation-First\*\*: Dynamic discovery, deterministic mapping, and reproducible manifests

- \*\*Resource-Efficient\*\*: Single VM deployment capability with <$2,000/month infrastructure cost

\*\*MVP Scope (Phase 1-7)\*\*:

- \*\*Countries\*\*: Sweden, Mexico, New Zealand, Thailand (4 countries)

- \*\*Indicators\*\*: GDP, Inflation CPI, Unemployment Rate, Trade Balance (4 core indicators)

- \*\*Models\*\*: 3 LLMs (GPT, Claude, Gemini) + 1 baseline (ARIMA)

- \*\*Time Granularity\*\*: Monthly where available, quarterly fallback

- \*\*Data Source\*\*: TradingEconomics API (primary)

\*\*What GlobalPulse IS\*\*:

- A neutral benchmark platform for macroeconomic forecasting accuracy and reasoning coherence

- A system with strict provenance, vintage control, and transparent manifests

- An API-first platform for real-time tasks and historical simulations

\*\*What GlobalPulse IS NOT\*\*:

- A trading signal generator or investment advisory product

- A comprehensive data warehouse mirroring statistical offices

- An LLM provider (orchestrates third-party models only)

- Limited to single data source (Trading Economics is MVP entry point)

---

## 3.2 Objectives & Non-Functional Goals

### Explicit NFRs with Target Numbers

\*\*Performance Targets\*\*:

- Ingest ≥500 indicators/day across ≥4 countries

- Normalization latency ≤2s per batch of 100 records

- API p95 latency ≤1s for metadata queries, ≤2s for evaluation queries

- End-to-end evaluation cycle ≤60 minutes (excluding LLM latency)

\*\*Cost Ceilings\*\*:

- Monthly infrastructure cost ≤$2,000 USD

- LLM inference cost ≤$500 USD per evaluation cycle

- API rate limit budget: 500 calls/day per source

\*\*Reliability & Observability\*\*:

- 99.5% uptime for read APIs during business hours

- 100% hash verification success rate for integrity checks

- ≤24 hour RPO for data recovery

- Weekly automated integrity audits with 100% pass rate

\*\*Scalability Targets\*\*:

- Support expansion to ≥10 countries within 6 months

- Handle ≥1M time series observations with partitioned storage

- Horizontal scaling capability for API and worker services

\*\*Accuracy & Quality\*\*:

- ≥95% taxonomy mapping accuracy (verified via spot audit)

- Inter-rater correlation ≥0.7 between automated and human coherence scoring

- Bit-for-bit reproducibility for identical manifest re-runs

---

## 3.3 System Overview (Logical Architecture)

### Services Architecture

```

[External APIs] → [Ingestion] → [Normalization] → [Governance]

↓ ↓ ↓ ↓

[TradingEcon] [Raw Storage] [Norm Storage] [Manifests]

↓ ↓ ↓ ↓

[Orchestrator] ← [Forecaster] ← [Task Builder] ← [Scheduler]

↓ ↓ ↓ ↓

[LLM Connx] → [Predictions] → [Evaluator] → [Leaderboard]

↓ ↓ ↓ ↓

[API Layer] ← [Dashboard] ← [Reporting] ← [Object Store]

```

### Core Components

\*\*Data Layer\*\*:

- \*\*Postgres 15+\*\*: Partitioned tables for time series, predictions, evaluations

- \*\*Redis\*\*: Caching metadata, rate limiting, job queues

- \*\*MinIO/S3\*\*: Reasoning texts, model responses, manifests

\*\*Service Layer\*\*:

- \*\*ingestor\*\*: Python service for API connectivity and raw data collection

- \*\*normalizer\*\*: Unit conversion, taxonomy mapping, quality flags

- \*\*governor\*\*: Integrity checks, manifest generation, audit logging

- \*\*forecaster\*\*: Task orchestration, prompt generation, LLM communication

- \*\*evaluator\*\*: Metrics computation, coherence scoring

- \*\*api\*\*: FastAPI read-only endpoints with OpenAPI spec

- \*\*scheduler\*\*: Prefect/Airflow for orchestrated workflows

\*\*Interface Layer\*\*:

- \*\*React Dashboard\*\*: Leaderboards, coverage heatmaps, time series visualization

- \*\*REST API\*\*: JSON responses with pagination and filtering

- \*\*Admin Interface\*\*: Governance tools, manifest management

### Data Flow Interfaces

| Interface | Protocol | SLA |

|-----------|----------|-----|

| External APIs → Ingestor | HTTP/REST | 500ms timeout, 3 retries |

| Ingestor → Raw Storage | SQL/ACID | Batch commits every 100 records |

| Normalizer → Governor | Message Queue | ≤5s processing time |

| Forecaster → LLM APIs | HTTP/REST | 30s timeout, exponential backoff |

| Evaluator → Storage | SQL/ACID | Atomic transaction per prediction |

| API → Dashboard | HTTP/JSON | ≤1s p95 response time |

---

## 3.4 Phase Plan (Strict, Incremental)

## Phase 1 — API Connectivity & Raw Ingestion

### Objective & Scope

Establish secure connectivity to TradingEconomics API and implement robust raw data ingestion pipeline with error handling and audit trails.

### Dependencies

- Environment variables configured (TE\_API\_KEY, SUPABASE\_\* credentials)

- Supabase project provisioned

- Development environment setup complete

### Functional Requirements

\*\*FR-1.1\*\*: The system MUST authenticate to TradingEconomics API using environment variable TE\_API\_KEY

\*\*FR-1.2\*\*: The system MUST fetch country metadata for Sweden, Mexico, New Zealand, Thailand

\*\*FR-1.3\*\*: The system MUST fetch indicator metadata for GDP, Inflation CPI, Unemployment Rate, Trade Balance

\*\*FR-1.4\*\*: The system MUST ingest 5 years of historical data per country-indicator pair

\*\*FR-1.5\*\*: The system MUST handle API rate limits with exponential backoff (initial delay 1s, max 60s)

\*\*FR-1.6\*\*: The system MUST retry failed requests 3 times before marking as failed

\*\*FR-1.7\*\*: The system MUST log all API calls with timestamp, endpoint, status code, and response time

### Non-Functional Requirements

- API response timeout: 30 seconds

- Maximum concurrent requests: 5

- Batch size: 100 records per commit

- Rate limit: 100 requests per hour per Trading Economics terms

### Deliverables

#### SQL DDL (Supabase)

```sql

-- Raw ingestion tables

CREATE TABLE raw\_te\_metadata (

id SERIAL PRIMARY KEY,

endpoint\_type VARCHAR(50) NOT NULL, -- 'countries' | 'indicators'

country\_code VARCHAR(3),

response\_data JSONB NOT NULL,

fetched\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

api\_status\_code INTEGER,

content\_hash VARCHAR(64) UNIQUE

);

CREATE TABLE raw\_te\_timeseries (

id SERIAL PRIMARY KEY,

country\_code VARCHAR(3) NOT NULL,

indicator\_name VARCHAR(255) NOT NULL,

date\_value DATE NOT NULL,

value\_raw DECIMAL,

unit\_raw VARCHAR(50),

source\_url TEXT,

release\_date DATE,

vintage\_date DATE,

ingested\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

content\_hash VARCHAR(64),

UNIQUE(country\_code, indicator\_name, date\_value, vintage\_date)

);

CREATE TABLE ingestion\_log (

id SERIAL PRIMARY KEY,

run\_id UUID DEFAULT gen\_random\_uuid(),

endpoint VARCHAR(255) NOT NULL,

status VARCHAR(20) NOT NULL, -- 'success' | 'failed' | 'retry'

records\_processed INTEGER DEFAULT 0,

error\_message TEXT,

started\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

completed\_at TIMESTAMP WITH TIME ZONE,

execution\_time\_ms INTEGER

);

-- Indexes

CREATE INDEX idx\_raw\_timeseries\_country\_indicator ON raw\_te\_timeseries(country\_code, indicator\_name);

CREATE INDEX idx\_raw\_timeseries\_date ON raw\_te\_timeseries(date\_value DESC);

CREATE INDEX idx\_ingestion\_log\_run\_id ON ingestion\_log(run\_id);

```

#### JSON Schemas (AJV-compatible)

```json

{

"$schema": "http://json-schema.org/draft-07/schema#",

"title": "TradingEconomicsResponse",

"type": "object",

"properties": {

"country": {"type": "string", "minLength": 2, "maxLength": 3},

"category": {"type": "string"},

"title": {"type": "string"},

"latestValue": {"type": ["number", "null"]},

"latestValueDate": {"type": ["string", "null"], "format": "date"},

"unit": {"type": "string"},

"url": {"type": "string", "format": "uri"},

"categoryGroup": {"type": "string"},

"adjustment": {"type": "string"},

"source": {"type": "string"},

"frequency": {"type": "string"}

},

"required": ["country", "title", "unit"],

"additionalProperties": false

}

```

#### Sample JSON Payloads

\*\*Raw Metadata Response:\*\*

```json

{

"endpoint\_type": "indicators",

"country\_code": "SWE",

"response\_data": {

"country": "Sweden",

"category": "GDP",

"title": "GDP Growth Rate",

"unit": "percent",

"frequency": "quarterly",

"source": "Statistics Sweden",

"url": "https://tradingeconomics.com/sweden/gdp-growth"

},

"fetched\_at": "2025-08-27T10:30:00Z",

"api\_status\_code": 200,

"content\_hash": "sha256:abc123..."

}

```

\*\*Raw Timeseries Response:\*\*

```json

{

"country\_code": "SWE",

"indicator\_name": "GDP Growth Rate",

"date\_value": "2024-12-01",

"value\_raw": 2.1,

"unit\_raw": "percent",

"source\_url": "https://tradingeconomics.com/sweden/gdp-growth",

"release\_date": "2025-02-15",

"vintage\_date": "2025-02-15",

"ingested\_at": "2025-08-27T10:35:00Z",

"content\_hash": "sha256:def456..."

}

```

#### n8n Workflow JSON

```json

{

"name": "TradingEconomics Ingestion Pipeline",

"nodes": [

{

"parameters": {

"rule": {

"interval": [{"field": "cronExpression", "value": "0 2 \* \* \*"}]

}

},

"name": "Daily Trigger",

"type": "n8n-nodes-base.cron",

"position": [300, 300]

},

{

"parameters": {

"url": "https://api.tradingeconomics.com/countries",

"authentication": "genericCredentialType",

"genericAuthType": "httpQueryAuth",

"options": {

"timeout": 30000,

"retry": {"enabled": true, "maxTries": 3}

}

},

"name": "Fetch Countries",

"type": "n8n-nodes-base.httpRequest",

"credentials": {"httpQueryAuth": {"id": "tradingeconomics\_creds"}}

},

{

"parameters": {

"functionCode": "const countries = ['Sweden', 'Mexico', 'New Zealand', 'Thailand'];\nconst indicators = ['GDP', 'Inflation Rate', 'Unemployment Rate', 'Trade Balance'];\nconst tasks = [];\n\nfor (const country of countries) {\n for (const indicator of indicators) {\n tasks.push({country, indicator, url: `https://api.tradingeconomics.com/historical/country/${country}/indicator/${indicator}`});\n }\n}\n\nreturn tasks.map(task => ({json: task}));"

},

"name": "Generate Tasks",

"type": "n8n-nodes-base.function"

},

{

"parameters": {

"batchSize": 1,

"options": {"continueOnFail": true}

},

"name": "Process Each Task",

"type": "n8n-nodes-base.splitInBatches"

},

{

"parameters": {

"url": "={{$json.url}}",

"qs": {"c": "={{$credentials.tradingeconomics\_creds.key}}", "d1": "2020-01-01"},

"options": {

"timeout": 30000,

"retry": {"enabled": true, "maxTries": 3}

}

},

"name": "Fetch Timeseries",

"type": "n8n-nodes-base.httpRequest",

"credentials": {"httpQueryAuth": {"id": "tradingeconomics\_creds"}}

},

{

"parameters": {

"operation": "executeQuery",

"query": "INSERT INTO raw\_te\_timeseries (country\_code, indicator\_name, date\_value, value\_raw, unit\_raw, source\_url, ingested\_at, content\_hash) VALUES ($1, $2, $3, $4, $5, $6, NOW(), $7) ON CONFLICT DO NOTHING",

"options": {"queryReplacement": "={{$json}}"}

},

"name": "Store Raw Data",

"type": "n8n-nodes-base.postgres",

"credentials": {"postgres": {"id": "supabase\_creds"}}

}

],

"connections": {

"Daily Trigger": {"main": [[{"node": "Fetch Countries", "type": "main", "index": 0}]]},

"Fetch Countries": {"main": [[{"node": "Generate Tasks", "type": "main", "index": 0}]]},

"Generate Tasks": {"main": [[{"node": "Process Each Task", "type": "main", "index": 0}]]},

"Process Each Task": {"main": [[{"node": "Fetch Timeseries", "type": "main", "index": 0}]]},

"Fetch Timeseries": {"main": [[{"node": "Store Raw Data", "type": "main", "index": 0}]]}

}

}

```

#### Error Handling & Recovery

- \*\*Exponential backoff\*\*: Initial delay 1s, multiplier 2x, max delay 60s

- \*\*Circuit breaker\*\*: Open after 5 consecutive failures, half-open after 5 minutes

- \*\*Dead letter queue\*\*: Failed requests stored for manual review after 3 retries

- \*\*Compensating actions\*\*: Mark ingestion run as failed, alert operations team

#### Testing Strategy

\*\*Unit Tests (Pytest)\*\*:

```python

def test\_api\_authentication():

"""Test API key validation and request signing"""

client = TradingEconomicsClient(api\_key="test\_key")

assert client.is\_authenticated()

def test\_rate\_limiting():

"""Test exponential backoff implementation"""

limiter = RateLimiter(max\_requests=5, window=60)

# Simulate rapid requests and verify backoff behavior

def test\_data\_validation():

"""Test JSON schema validation for API responses"""

response = {"country": "Sweden", "title": "GDP", "unit": "percent"}

assert validate\_te\_response(response) == True

```

### Definition of Done (Gate Criteria)

- [ ] All 4 countries metadata successfully fetched and stored

- [ ] All 4 indicators per country successfully ingested with 5 years history

- [ ] Rate limiting implemented and tested with simulated API limits

- [ ] Error handling covers all HTTP error codes (4xx, 5xx)

- [ ] Ingestion logs capture 100% of API calls with timing metrics

- [ ] Unit tests achieve >80% code coverage

- [ ] Integration test successfully runs end-to-end ingestion cycle

### Handover/Context Transfer

\*\*Outputs for Phase 2\*\*:

- Raw timeseries data in `raw\_te\_timeseries` table (clean, deduplicated)

- Metadata mapping in `raw\_te\_metadata` table (countries and indicators)

- Ingestion logs with success/failure rates and performance metrics

- JSON schema validation rules for data quality gates

---

## Phase 2 — Normalization & Storage (Supabase)

### Objective & Scope

Transform raw TradingEconomics data into canonical format with standardized units, create normalized schema with RLS policies, and implement data quality validation pipeline.

### Dependencies (from Phase 1)

- Raw timeseries data in `raw\_te\_timeseries` table

- Metadata records in `raw\_te\_metadata` table

- JSON schema validation working

- Ingestion pipeline operational

### Functional Requirements

\*\*FR-2.1\*\*: The system MUST convert all indicator values to canonical units (GDP: % y/y, CPI: % y/y, Unemployment: %, Trade Balance: USD millions)

\*\*FR-2.2\*\*: The system MUST assign each indicator to one of five categories: Growth, Prices, Labor, Trade, Sentiment

\*\*FR-2.3\*\*: The system MUST validate data quality with flags for: outliers (>3 standard deviations), gaps (>60 days missing), revisions (vintage changes)

\*\*FR-2.4\*\*: The system MUST store provenance linking normalized records to raw sources

\*\*FR-2.5\*\*: The system MUST implement Row Level Security for data access control

\*\*FR-2.6\*\*: The system MUST handle missing values with documented imputation rules

\*\*FR-2.7\*\*: The system MUST validate referential integrity between countries, indicators, and observations

### Non-Functional Requirements

- Normalization processing time: ≤2 seconds per 100-record batch

- Data quality validation: 100% coverage with automated flagging

- RLS policy enforcement: 100% of queries filtered by user role

- Storage efficiency: Partitioned tables by country and year

### Deliverables

#### SQL DDL (Supabase) - Canonical Schema

```sql

-- Core reference tables

CREATE TABLE countries (

country\_code VARCHAR(3) PRIMARY KEY,

country\_name VARCHAR(100) NOT NULL,

region VARCHAR(50),

currency\_code VARCHAR(3),

timezone VARCHAR(50),

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

CREATE TABLE indicators (

indicator\_id SERIAL PRIMARY KEY,

indicator\_name VARCHAR(255) NOT NULL,

canonical\_name VARCHAR(255) NOT NULL,

category VARCHAR(20) NOT NULL CHECK (category IN ('Growth', 'Prices', 'Labor', 'Trade', 'Sentiment')),

canonical\_unit VARCHAR(50) NOT NULL,

frequency VARCHAR(20) NOT NULL CHECK (frequency IN ('daily', 'monthly', 'quarterly', 'annual')),

description TEXT,

is\_primary BOOLEAN DEFAULT FALSE,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

UNIQUE(canonical\_name, category)

);

CREATE TABLE sources (

source\_id SERIAL PRIMARY KEY,

source\_name VARCHAR(100) NOT NULL,

source\_url VARCHAR(500),

tier VARCHAR(5) NOT NULL CHECK (tier IN ('A', 'B', 'C')), -- A=Official, B=Multilateral, C=Commercial

license\_type VARCHAR(50),

attribution\_required BOOLEAN DEFAULT TRUE,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- Main normalized observations (partitioned by country\_code, year)

CREATE TABLE observations (

observation\_id BIGSERIAL,

country\_code VARCHAR(3) NOT NULL,

indicator\_id INTEGER NOT NULL,

observation\_date DATE NOT NULL,

value DECIMAL(15,6),

canonical\_unit VARCHAR(50) NOT NULL,

source\_id INTEGER NOT NULL,

quality\_flags JSONB DEFAULT '{}',

vintage\_date DATE NOT NULL,

release\_date DATE,

raw\_value DECIMAL(15,6),

raw\_unit VARCHAR(50),

conversion\_factor DECIMAL(10,6),

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

FOREIGN KEY (country\_code) REFERENCES countries(country\_code),

FOREIGN KEY (indicator\_id) REFERENCES indicators(indicator\_id),

FOREIGN KEY (source\_id) REFERENCES sources(source\_id),

PRIMARY KEY (observation\_id, country\_code, EXTRACT(YEAR FROM observation\_date))

) PARTITION BY RANGE (country\_code, EXTRACT(YEAR FROM observation\_date));

-- Create partitions for each country and year

CREATE TABLE observations\_swe\_2020\_2025 PARTITION OF observations

FOR VALUES FROM ('SWE', 2020) TO ('SWE', 2026);

CREATE TABLE observations\_mex\_2020\_2025 PARTITION OF observations

FOR VALUES FROM ('MEX', 2020) TO ('MEX', 2026);

CREATE TABLE observations\_nzl\_2020\_2025 PARTITION OF observations

FOR VALUES FROM ('NZL', 2020) TO ('NZL', 2026);

CREATE TABLE observations\_tha\_2020\_2025 PARTITION OF observations

FOR VALUES FROM ('THA', 2020) TO ('THA', 2026);

-- Lineage tracking

CREATE TABLE lineage\_audit (

audit\_id BIGSERIAL PRIMARY KEY,

observation\_id BIGINT NOT NULL,

raw\_record\_id BIGINT NOT NULL,

transformation\_type VARCHAR(50) NOT NULL,

transformation\_params JSONB,

processed\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

processed\_by VARCHAR(100) NOT NULL -- service identifier

);

-- Quality validation results

CREATE TABLE data\_quality\_checks (

check\_id BIGSERIAL PRIMARY KEY,

observation\_id BIGINT NOT NULL,

check\_type VARCHAR(50) NOT NULL,

check\_result VARCHAR(20) NOT NULL CHECK (check\_result IN ('pass', 'warning', 'fail')),

check\_message TEXT,

checked\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- Indexes for performance

CREATE INDEX idx\_observations\_country\_indicator\_date ON observations(country\_code, indicator\_id, observation\_date DESC);

CREATE INDEX idx\_observations\_vintage ON observations(vintage\_date DESC);

CREATE INDEX idx\_quality\_flags\_gin ON observations USING GIN(quality\_flags);

```

#### RLS Policies

```sql

-- Enable RLS on all tables

ALTER TABLE observations ENABLE ROW LEVEL SECURITY;

ALTER TABLE countries ENABLE ROW LEVEL SECURITY;

ALTER TABLE indicators ENABLE ROW LEVEL SECURITY;

ALTER TABLE sources ENABLE ROW LEVEL SECURITY;

-- Admin role: full access

CREATE POLICY admin\_full\_access ON observations

FOR ALL TO authenticated

USING (auth.jwt() ->> 'role' = 'admin');

-- Viewer role: read-only access to public data

CREATE POLICY viewer\_read\_only ON observations

FOR SELECT TO authenticated

USING (

auth.jwt() ->> 'role' IN ('viewer', 'evaluator', 'curator')

AND quality\_flags->>'public\_release' = 'true'

);

-- Curator role: can update quality flags

CREATE POLICY curator\_quality\_updates ON observations

FOR UPDATE TO authenticated

USING (auth.jwt() ->> 'role' = 'curator')

WITH CHECK (auth.jwt() ->> 'role' = 'curator');

-- Service role: internal processing access

CREATE POLICY service\_processing ON observations

FOR ALL TO service\_role

USING (true);

```

#### JSON Schema (Normalized Observation)

```json

{

"$schema": "http://json-schema.org/draft-07/schema#",

"title": "NormalizedObservation",

"type": "object",

"properties": {

"country\_code": {"type": "string", "pattern": "^[A-Z]{3}$"},

"indicator\_id": {"type": "integer", "minimum": 1},

"observation\_date": {"type": "string", "format": "date"},

"value": {"type": ["number", "null"]},

"canonical\_unit": {"type": "string"},

"source\_id": {"type": "integer", "minimum": 1},

"quality\_flags": {

"type": "object",

"properties": {

"outlier": {"type": "boolean"},

"gap\_filled": {"type": "boolean"},

"revision": {"type": "boolean"},

"public\_release": {"type": "boolean"},

"confidence\_score": {"type": "number", "minimum": 0, "maximum": 1}

},

"additionalProperties": false

},

"vintage\_date": {"type": "string", "format": "date"},

"provenance": {

"type": "object",

"properties": {

"raw\_record\_id": {"type": "integer"},

"conversion\_factor": {"type": "number"},

"transformation\_type": {"type": "string"}

},

"required": ["raw\_record\_id"]

}

},

"required": ["country\_code", "indicator\_id", "observation\_date", "canonical\_unit", "source\_id", "vintage\_date"],

"additionalProperties": false

}

```

#### Sample JSON Output (Normalized)

```json

{

"country\_code": "SWE",

"indicator\_id": 1,

"observation\_date": "2024-12-01",

"value": 2.1,

"canonical\_unit": "% y/y",

"source\_id": 1,

"quality\_flags": {

"outlier": false,

"gap\_filled": false,

"revision": false,

"public\_release": true,

"confidence\_score": 0.95

},

"vintage\_date": "2025-02-15",

"release\_date": "2025-02-15",

"provenance": {

"raw\_record\_id": 12345,

"conversion\_factor": 1.0,

"transformation\_type": "unit\_standardization"

}

}

```

#### Governance Rules Embedded

\*\*Unit Conversion Rules\*\*:

```python

UNIT\_CONVERSIONS = {

'Growth': {

'percent': 1.0, # Already in % y/y

'percentage': 1.0,

'rate': 1.0,

'decimal': 100.0 # 0.021 -> 2.1%

},

'Prices': {

'percent': 1.0,

'percentage': 1.0,

'index': lambda x, base=100: ((x / base) - 1) \* 100 # Index to % change

},

'Trade': {

'USD millions': 1.0,

'USD billions': 1000.0, # Billions to millions

'USD thousands': 0.001, # Thousands to millions

'local\_currency': 'CONVERT\_VIA\_EXCHANGE\_RATE'

}

}

```

\*\*Quality Flag Rules\*\*:

```python

def apply\_quality\_flags(value, series\_history, metadata):

flags = {

'outlier': abs(value - series\_history.mean()) > 3 \* series\_history.std(),

'gap\_filled': metadata.get('interpolated', False),

'revision': check\_vintage\_changes(value, metadata),

'public\_release': metadata.get('tier') in ['A', 'B'],

'confidence\_score': calculate\_confidence(value, series\_history, metadata)

}

return flags

```

#### Testing Strategy

\*\*Unit Tests (Pytest)\*\*:

```python

def test\_unit\_conversion():

"""Test canonical unit conversion accuracy"""

converter = UnitConverter()

result = converter.convert(2.1, 'percent', 'Growth')

assert result == 2.1 # No conversion needed

result = converter.convert(0.021, 'decimal', 'Growth')

assert result == 2.1 # Decimal to percentage

def test\_quality\_flags():

"""Test quality flag assignment logic"""

series = [2.0, 2.1, 2.2, 2.0, 8.5] # Last value is outlier

flags = apply\_quality\_flags(8.5, pd.Series(series[:-1]), {})

assert flags['outlier'] == True

assert flags['confidence\_score'] < 0.5

def test\_rls\_policies():

"""Test row level security enforcement"""

# Test viewer can only see public data

# Test admin can see all data

# Test curator can update quality flags

```

\*\*Integration Tests\*\*:

```python

def test\_end\_to\_end\_normalization():

"""Test complete raw -> normalized pipeline"""

# Insert raw data

# Run normalization pipeline

# Verify normalized output matches expected schema

# Verify lineage audit trail exists

# Verify quality flags applied correctly

```

### Definition of Done (Gate Criteria)

- [ ] All raw timeseries normalized with canonical units (100% conversion rate)

- [ ] Quality flags applied to 100% of observations

- [ ] RLS policies enforced (tested with different user roles)

- [ ] Partitioned tables created and data distributed correctly

- [ ] Lineage audit trail connects all normalized records to raw sources

- [ ] Data quality checks identify and flag anomalies

- [ ] Unit tests pass with >85% coverage

- [ ] Integration test validates complete pipeline

### Handover/Context Transfer

\*\*Outputs for Phase 3\*\*:

| Artifact | Schema/Format | Consumer |

|----------|---------------|----------|

| Normalized observations | `observations` table | Taxonomy mapper |

| Quality flags | JSONB in observations | Governance validator |

| Country/indicator metadata | `countries`, `indicators` tables | Mapping algorithm |

| Lineage audit trail | `lineage\_audit` table | Governance auditor |

---

## Phase 3 — Taxonomy & Mapping

### Objective & Scope

Implement deterministic mapping algorithm to assign indicators to canonical categories (Growth, Prices, Labor, Trade, Sentiment) with fallback hierarchy and country-specific customization support.

### Dependencies (from Phase 2)

- Normalized observations in `observations` table with quality flags

- Country metadata in `countries` table (Sweden, Mexico, New Zealand, Thailand)

- Indicator metadata in `indicators` table with canonical names

- Unit conversions working and validated

### Functional Requirements

\*\*FR-3.1\*\*: The system MUST implement deterministic mapping algorithm that assigns each country-indicator pair to exactly one canonical category

\*\*FR-3.2\*\*: The system MUST define primary and fallback indicators for each category per country

\*\*FR-3.3\*\*: The system MUST calculate coverage ratio (valid observations / expected observations) for each indicator over 5-year lookback

\*\*FR-3.4\*\*: The system MUST escalate to fallback indicators when primary coverage <80% or unavailable

\*\*FR-3.5\*\*: The system MUST store mapping decisions with rationale and coverage metrics

\*\*FR-3.6\*\*: The system MUST support country-specific aliases and indicator name variations

\*\*FR-3.7\*\*: The system MUST validate mapping consistency across time periods and detect drift

### Non-Functional Requirements

- Mapping algorithm execution: ≤10 seconds for all countries and indicators

- Coverage calculation accuracy: 100% of time periods assessed

- Mapping consistency: 95% stability across monthly recalculations

- Fallback escalation: Deterministic ordering with documented rationale

### Deliverables

#### SQL DDL (Taxonomy Tables)

```sql

-- Canonical category definitions

CREATE TABLE categories (

category\_id SERIAL PRIMARY KEY,

category\_name VARCHAR(20) NOT NULL UNIQUE CHECK (category\_name IN ('Growth', 'Prices', 'Labor', 'Trade', 'Sentiment')),

description TEXT NOT NULL,

canonical\_unit VARCHAR(50) NOT NULL,

frequency\_priority VARCHAR(20)[] DEFAULT ARRAY['monthly', 'quarterly', 'annual'], -- Preference order

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- Indicator taxonomy with hierarchy

CREATE TABLE indicator\_taxonomy (

taxonomy\_id SERIAL PRIMARY KEY,

category\_id INTEGER NOT NULL,

indicator\_pattern VARCHAR(255) NOT NULL, -- Regex pattern for matching

indicator\_aliases TEXT[], -- Alternative names

priority\_rank INTEGER NOT NULL, -- 0=primary, 1=fallback1, 2=fallback2, etc.

unit\_mappings JSONB NOT NULL, -- Unit conversion rules

coverage\_threshold DECIMAL(3,2) DEFAULT 0.80, -- Minimum coverage to use

geographic\_scope VARCHAR(10)[] DEFAULT ARRAY['global'], -- 'global', specific country codes

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

FOREIGN KEY (category\_id) REFERENCES categories(category\_id),

UNIQUE(category\_id, priority\_rank, geographic\_scope)

);

-- Country-specific indicator mappings (resolved)

CREATE TABLE country\_indicator\_mapping (

mapping\_id SERIAL PRIMARY KEY,

country\_code VARCHAR(3) NOT NULL,

category\_id INTEGER NOT NULL,

indicator\_id INTEGER NOT NULL, -- From indicators table

mapping\_type VARCHAR(20) NOT NULL CHECK (mapping\_type IN ('primary', 'fallback\_1', 'fallback\_2', 'fallback\_3')),

coverage\_ratio DECIMAL(5,4) NOT NULL, -- 0.0000 to 1.0000

decision\_reason TEXT NOT NULL,

lookback\_start\_date DATE NOT NULL,

lookback\_end\_date DATE NOT NULL,

decided\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

decided\_by VARCHAR(100) NOT NULL, -- Service/algorithm identifier

is\_active BOOLEAN DEFAULT TRUE,

FOREIGN KEY (country\_code) REFERENCES countries(country\_code),

FOREIGN KEY (category\_id) REFERENCES categories(category\_id),

FOREIGN KEY (indicator\_id) REFERENCES indicators(indicator\_id),

UNIQUE(country\_code, category\_id, is\_active) WHERE is\_active = TRUE

);

-- Taxonomy versioning for governance

CREATE TABLE taxonomy\_versions (

version\_id SERIAL PRIMARY KEY,

version\_number VARCHAR(10) NOT NULL, -- semantic versioning

effective\_date DATE NOT NULL,

taxonomy\_hash VARCHAR(64) NOT NULL, -- SHA-256 of taxonomy rules

change\_summary TEXT,

approved\_by VARCHAR(100),

is\_current BOOLEAN DEFAULT FALSE,

UNIQUE(version\_number),

CHECK (version\_number ~ '^[0-9]+\.[0-9]+\.[0-9]+)

);

-- Insert canonical categories

INSERT INTO categories (category\_name, description, canonical\_unit, frequency\_priority) VALUES

('Growth', 'Real economic output growth indicators', '% y/y', ARRAY['quarterly', 'monthly', 'annual']),

('Prices', 'Inflation and price level indicators', '% y/y', ARRAY['monthly', 'quarterly', 'annual']),

('Labor', 'Employment and labor market indicators', '%', ARRAY['monthly', 'quarterly', 'annual']),

('Trade', 'International trade flow indicators', 'USD millions', ARRAY['monthly', 'quarterly', 'annual']),

('Sentiment', 'Confidence and sentiment indicators', 'index', ARRAY['monthly', 'quarterly', 'annual']);

-- Insert taxonomy patterns (examples for Growth category)

INSERT INTO indicator\_taxonomy (category\_id, indicator\_pattern, indicator\_aliases, priority\_rank, unit\_mappings, geographic\_scope) VALUES

(1, '^GDP.\*Growth.\*Rate, ARRAY['Real GDP Growth', 'GDP YoY', 'Gross Domestic Product Growth'], 0, '{"percent": 1.0, "decimal": 100.0}', ARRAY['global']),

(1, '^Industrial.\*Production.\*', ARRAY['IP Index', 'Manufacturing Output', 'Industrial Output'], 1, '{"percent": 1.0, "index": "yoy\_calculation"}', ARRAY['global']),

(1, '^Employment.\*Growth, ARRAY['Job Growth', 'Employment Rate Change'], 2, '{"percent": 1.0}', ARRAY['global']);

```

#### JSON Schema (Taxonomy Entry)

```json

{

"$schema": "http://json-schema.org/draft-07/schema#",

"title": "CountryIndicatorMapping",

"type": "object",

"properties": {

"country\_code": {"type": "string", "pattern": "^[A-Z]{3}$"},

"category": {"type": "string", "enum": ["Growth", "Prices", "Labor", "Trade", "Sentiment"]},

"primary\_indicator": {

"type": "object",

"properties": {

"indicator\_id": {"type": "integer"},

"indicator\_name": {"type": "string"},

"coverage\_ratio": {"type": "number", "minimum": 0, "maximum": 1}

},

"required": ["indicator\_id", "indicator\_name", "coverage\_ratio"]

},

"fallback\_indicators": {

"type": "array",

"items": {

"type": "object",

"properties": {

"rank": {"type": "integer", "minimum": 1},

"indicator\_id": {"type": "integer"},

"indicator\_name": {"type": "string"},

"coverage\_ratio": {"type": "number", "minimum": 0, "maximum": 1},

"reason": {"type": "string"}

},

"required": ["rank", "indicator\_id", "indicator\_name", "coverage\_ratio", "reason"]

}

},

"selected\_mapping": {

"type": "object",

"properties": {

"mapping\_type": {"type": "string", "enum": ["primary", "fallback\_1", "fallback\_2", "fallback\_3"]},

"indicator\_id": {"type": "integer"},

"coverage\_ratio": {"type": "number"},

"decision\_reason": {"type": "string"}

},

"required": ["mapping\_type", "indicator\_id", "coverage\_ratio", "decision\_reason"]

},

"correlates": {

"type": "array",

"items": {

"type": "object",

"properties": {

"indicator\_id": {"type": "integer"},

"indicator\_name": {"type": "string"},

"correlation\_score": {"type": "number", "minimum": -1, "maximum": 1}

}

}

}

},

"required": ["country\_code", "category", "selected\_mapping"],

"additionalProperties": false

}

```

#### Sample JSON (Taxonomy Mapping Result)

```json

{

"country\_code": "SWE",

"category": "Growth",

"primary\_indicator": {

"indicator\_id": 1,

"indicator\_name": "GDP Growth Rate",

"coverage\_ratio": 0.98

},

"fallback\_indicators": [

{

"rank": 1,

"indicator\_id": 15,

"indicator\_name": "Industrial Production YoY",

"coverage\_ratio": 0.95,

"reason": "Strong correlation with GDP, monthly frequency"

},

{

"rank": 2,

"indicator\_id": 23,

"indicator\_name": "Employment Growth",

"coverage\_ratio": 0.87,

"reason": "Labor market proxy for economic activity"

}

],

"selected\_mapping": {

"mapping\_type": "primary",

"indicator\_id": 1,

"coverage\_ratio": 0.98,

"decision\_reason": "Primary indicator available with excellent coverage (98%)"

},

"correlates": [

{

"indicator\_id": 45,

"indicator\_name": "Consumer Confidence",

"correlation\_score": 0.67

},

{

"indicator\_id": 12,

"indicator\_name": "Unemployment Rate",

"correlation\_score": -0.72

}

]

}

```

#### Governance Rules (Mapping Algorithm)

```python

class TaxonomyMapper:

def \_\_init\_\_(self):

self.COVERAGE\_THRESHOLD = 0.80

self.LOOKBACK\_YEARS = 5

self.MIN\_OBSERVATIONS = 10

def map\_country\_indicators(self, country\_code: str) -> Dict[str, Dict]:

"""

Deterministic mapping algorithm for country indicators

"""

mappings = {}

for category in CATEGORIES:

mapping = self.\_map\_category(country\_code, category)

mappings[category] = mapping

return mappings

def \_map\_category(self, country\_code: str, category: str) -> Dict:

"""

Map single category using priority hierarchy

"""

# Get taxonomy rules for category

taxonomy\_rules = self.\_get\_taxonomy\_rules(category)

# Get available indicators for country

available\_indicators = self.\_get\_country\_indicators(country\_code)

# Calculate coverage for each potential indicator

candidates = []

for rule in taxonomy\_rules:

for indicator in available\_indicators:

if self.\_matches\_pattern(indicator.name, rule.pattern, rule.aliases):

coverage = self.\_calculate\_coverage(

country\_code, indicator.id, self.LOOKBACK\_YEARS

)

candidates.append({

'indicator\_id': indicator.id,

'indicator\_name': indicator.name,

'priority\_rank': rule.priority\_rank,

'coverage\_ratio': coverage,

'rule': rule

})

# Sort by priority rank, then by coverage

candidates.sort(key=lambda x: (x['priority\_rank'], -x['coverage\_ratio']))

# Select first candidate meeting threshold

selected = None

for candidate in candidates:

if candidate['coverage\_ratio'] >= self.COVERAGE\_THRESHOLD:

selected = candidate

break

# If no candidate meets threshold, take best available

if not selected and candidates:

selected = candidates[0]

# Determine mapping type

mapping\_type = 'primary' if selected['priority\_rank'] == 0 else f'fallback\_{selected["priority\_rank"]}'

return {

'mapping\_type': mapping\_type,

'indicator\_id': selected['indicator\_id'],

'coverage\_ratio': selected['coverage\_ratio'],

'decision\_reason': self.\_generate\_decision\_reason(selected, candidates),

'all\_candidates': candidates

}

def \_calculate\_coverage(self, country\_code: str, indicator\_id: int, years: int) -> float:

"""

Calculate coverage ratio over lookback period

"""

end\_date = datetime.now().date()

start\_date = end\_date - timedelta(days=years \* 365)

# Get expected vs actual observation count

expected\_count = self.\_get\_expected\_observations(indicator\_id, start\_date, end\_date)

actual\_count = self.\_get\_actual\_observations(country\_code, indicator\_id, start\_date, end\_date)

if expected\_count == 0:

return 0.0

return min(1.0, actual\_count / expected\_count)

def \_generate\_decision\_reason(self, selected: Dict, candidates: List[Dict]) -> str:

"""

Generate human-readable decision rationale

"""

if selected['priority\_rank'] == 0:

return f"Primary indicator available with {selected['coverage\_ratio']:.1%} coverage"

else:

primary\_available = any(c['priority\_rank'] == 0 for c in candidates)

if primary\_available:

primary = next(c for c in candidates if c['priority\_rank'] == 0)

return f"Primary indicator coverage too low ({primary['coverage\_ratio']:.1%}), using fallback with {selected['coverage\_ratio']:.1%} coverage"

else:

return f"Primary indicator not available, using fallback rank {selected['priority\_rank']} with {selected['coverage\_ratio']:.1%} coverage"

```

#### Testing Strategy

\*\*Unit Tests (Pytest)\*\*:

```python

def test\_coverage\_calculation():

"""Test coverage ratio calculation accuracy"""

mapper = TaxonomyMapper()

# Mock data with known gaps

coverage = mapper.\_calculate\_coverage('SWE', 1, 2)

assert 0.0 <= coverage <= 1.0

def test\_pattern\_matching():

"""Test indicator name pattern matching"""

mapper = TaxonomyMapper()

assert mapper.\_matches\_pattern("GDP Growth Rate", "^GDP.\*Growth.\*Rate$", [])

assert mapper.\_matches\_pattern("Real GDP Growth", "^GDP.\*Growth.\*Rate$", ["Real GDP Growth"])

assert not mapper.\_matches\_pattern("Unemployment Rate", "^GDP.\*Growth.\*Rate$", [])

def test\_deterministic\_mapping():

"""Test mapping algorithm determinism"""

mapper = TaxonomyMapper()

result1 = mapper.map\_country\_indicators('SWE')

result2 = mapper.map\_country\_indicators('SWE')

assert result1 == result2 # Must be identical

def test\_fallback\_escalation():

"""Test fallback hierarchy logic"""

# Mock scenario where primary has low coverage

# Verify system selects appropriate fallback

pass

```

### Definition of Done (Gate Criteria)

- [ ] Mapping algorithm produces deterministic results (100% reproducibility)

- [ ] All 4 countries mapped to all 5 categories with documented rationale

- [ ] Coverage calculations validated against manual spot checks

- [ ] Fallback escalation working when primary indicators insufficient

- [ ] Taxonomy versioning implemented with hash verification

- [ ] Unit tests cover all mapping logic branches (>90% coverage)

- [ ] Integration test validates end-to-end mapping pipeline

### Handover/Context Transfer

\*\*Outputs for Phase 4\*\*:

| Artifact | Schema/Format | Consumer |

|----------|---------------|----------|

| Country-indicator mappings | `country\_indicator\_mapping` table | Governance validator |

| Taxonomy version manifest | `taxonomy\_versions` table | Audit system |

| Coverage metrics | JSON with rationales | Forecast task builder |

| Correlate indicators | Correlation matrix | Context provider |

---

## Phase 4 — Governance & Integrity

### Objective & Scope

Implement comprehensive data governance framework with validation rules, audit trails, conflict resolution workflows, and integrity monitoring to ensure data quality and regulatory compliance.

### Dependencies (from Phase 3)

- Country-indicator mappings in `country\_indicator\_mapping` table

- Taxonomy version control system operational

- Normalized observations with quality flags

- Lineage audit trail from normalization phase

### Functional Requirements

\*\*FR-4.1\*\*: The system MUST validate all ingested data against cutoff timestamp rules (no future-dated observations beyond release date)

\*\*FR-4.2\*\*: The system MUST implement provenance tracking from raw source to final evaluation results

\*\*FR-4.3\*\*: The system MUST detect and flag duplicate records across vintages and sources

\*\*FR-4.4\*\*: The system MUST implement anomaly detection for statistical outliers and data distribution shifts

\*\*FR-4.5\*\*: The system MUST maintain immutable audit log of all data modifications and access

\*\*FR-4.6\*\*: The system MUST implement conflict resolution workflow for disputed data or mappings

\*\*FR-4.7\*\*: The system MUST generate data quality reports with coverage, accuracy, and timeliness metrics

\*\*FR-4.8\*\*: The system MUST enforce data retention and archival policies per regulatory requirements

### Non-Functional Requirements

- Governance rule validation: ≤500ms per observation batch

- Audit log write performance: ≤100ms per log entry

- Integrity check completion: ≤10 minutes for full database scan

- Conflict resolution SLA: 48 hours for standard disputes, 24 hours for critical

### Deliverables

#### SQL DDL (Governance Tables)

```sql

-- Governance policies and rules

CREATE TABLE governance\_policies (

policy\_id SERIAL PRIMARY KEY,

policy\_name VARCHAR(100) NOT NULL UNIQUE,

policy\_type VARCHAR(50) NOT NULL CHECK (policy\_type IN ('validation', 'retention', 'access', 'quality')),

policy\_rules JSONB NOT NULL,

effective\_date DATE NOT NULL,

expiry\_date DATE,

created\_by VARCHAR(100) NOT NULL,

approved\_by VARCHAR(100),

is\_active BOOLEAN DEFAULT TRUE,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- Comprehensive audit log

CREATE TABLE audit\_log (

audit\_id BIGSERIAL PRIMARY KEY,

entity\_type VARCHAR(50) NOT NULL, -- 'observation', 'mapping', 'evaluation', etc.

entity\_id BIGINT NOT NULL,

action\_type VARCHAR(50) NOT NULL, -- 'create', 'update', 'delete', 'access'

actor\_type VARCHAR(50) NOT NULL, -- 'user', 'service', 'system'

actor\_id VARCHAR(100) NOT NULL,

session\_id UUID,

change\_details JSONB,

ip\_address INET,

user\_agent TEXT,

request\_id UUID,

timestamp TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

severity VARCHAR(20) DEFAULT 'info' CHECK (severity IN ('debug', 'info', 'warning', 'error', 'critical'))

);

-- Data validation results

CREATE TABLE validation\_results (

validation\_id BIGSERIAL PRIMARY KEY,

entity\_type VARCHAR(50) NOT NULL,

entity\_id BIGINT NOT NULL,

validation\_rule VARCHAR(100) NOT NULL,

validation\_status VARCHAR(20) NOT NULL CHECK (validation\_status IN ('pass', 'warning', 'fail', 'skip')),

validation\_message TEXT,

validation\_details JSONB,

validated\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

validated\_by VARCHAR(100) NOT NULL

);

-- Conflict resolution tracking

CREATE TABLE data\_conflicts (

conflict\_id SERIAL PRIMARY KEY,

conflict\_type VARCHAR(50) NOT NULL CHECK (conflict\_type IN ('duplicate\_data', 'source\_mismatch', 'mapping\_dispute', 'quality\_question')),

entity\_type VARCHAR(50) NOT NULL,

entity\_id BIGINT NOT NULL,

description TEXT NOT NULL,

affected\_records JSONB, -- Array of affected record IDs

reported\_by VARCHAR(100) NOT NULL,

assigned\_to VARCHAR(100),

status VARCHAR(20) DEFAULT 'open' CHECK (status IN ('open', 'investigating', 'resolved', 'closed')),

priority VARCHAR(10) DEFAULT 'medium' CHECK (priority IN ('low', 'medium', 'high', 'critical')),

resolution TEXT,

resolved\_by VARCHAR(100),

reported\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

resolved\_at TIMESTAMP WITH TIME ZONE

);

-- Data quality metrics

CREATE TABLE data\_quality\_metrics (

metric\_id BIGSERIAL PRIMARY KEY,

country\_code VARCHAR(3),

indicator\_id INTEGER,

metric\_type VARCHAR(50) NOT NULL,

metric\_value DECIMAL(10,4),

metric\_unit VARCHAR(20),

measurement\_period\_start DATE,

measurement\_period\_end DATE,

measured\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

FOREIGN KEY (country\_code) REFERENCES countries(country\_code),

FOREIGN KEY (indicator\_id) REFERENCES indicators(indicator\_id)

);

-- Immutable snapshots for point-in-time recovery

CREATE TABLE data\_snapshots (

snapshot\_id SERIAL PRIMARY KEY,

snapshot\_type VARCHAR(50) NOT NULL, -- 'daily', 'pre\_evaluation', 'milestone'

content\_hash VARCHAR(64) NOT NULL,

metadata JSONB,

storage\_path TEXT NOT NULL,

compression\_type VARCHAR(20) DEFAULT 'gzip',

size\_bytes BIGINT,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

expires\_at TIMESTAMP WITH TIME ZONE

);

-- Indexes for performance

CREATE INDEX idx\_audit\_log\_entity ON audit\_log(entity\_type, entity\_id);

CREATE INDEX idx\_audit\_log\_timestamp ON audit\_log(timestamp DESC);

CREATE INDEX idx\_audit\_log\_actor ON audit\_log(actor\_type, actor\_id);

CREATE INDEX idx\_validation\_results\_entity ON validation\_results(entity\_type, entity\_id);

CREATE INDEX idx\_conflicts\_status ON data\_conflicts(status, priority);

CREATE INDEX idx\_quality\_metrics\_country\_indicator ON data\_quality\_metrics(country\_code, indicator\_id);

```

#### JSON Schema (Validation Rule)

```json

{

"$schema": "http://json-schema.org/draft-07/schema#",

"title": "GovernanceValidationRule",

"type": "object",

"properties": {

"rule\_name": {"type": "string"},

"rule\_type": {"type": "string", "enum": ["temporal", "statistical", "referential", "business"]},

"entity\_types": {"type": "array", "items": {"type": "string"}},

"conditions": {

"type": "object",

"properties": {

"field\_checks": {

"type": "array",

"items": {

"type": "object",

"properties": {

"field": {"type": "string"},

"operator": {"type": "string", "enum": ["eq", "ne", "gt", "lt", "gte", "lte", "between", "in", "regex"]},

"value": {},

"message": {"type": "string"}

},

"required": ["field", "operator", "message"]

}

},

"cross\_field\_checks": {

"type": "array",

"items": {

"type": "object",

"properties": {

"expression": {"type": "string"},

"message": {"type": "string"}

}

}

},

"statistical\_checks": {

"type": "object",

"properties": {

"outlier\_detection": {

"type": "object",

"properties": {

"method": {"type": "string", "enum": ["zscore", "iqr", "isolation\_forest"]},

"threshold": {"type": "number"},

"window\_size": {"type": "integer"}

}

}

}

}

}

},

"severity": {"type": "string", "enum": ["warning", "error", "critical"]},

"auto\_remediation": {

"type": "object",

"properties": {

"enabled": {"type": "boolean"},

"action": {"type": "string", "enum": ["flag", "quarantine", "correct", "reject"]},

"parameters": {"type": "object"}

}

}

},

"required": ["rule\_name", "rule\_type", "conditions", "severity"],

"additionalProperties": false

}

```

#### Sample JSON (Governance Audit Event)

```json

{

"audit\_id": 12345,

"entity\_type": "observation",

"entity\_id": 67890,

"action\_type": "create",

"actor\_type": "service",

"actor\_id": "normalizer-v1.2.3",

"session\_id": "550e8400-e29b-41d4-a716-446655440000",

"change\_details": {

"operation": "normalize\_observation",

"input": {

"raw\_value": 2.1,

"raw\_unit": "percent"

},

"output": {

"value": 2.1,

"canonical\_unit": "% y/y",

"quality\_flags": {

"outlier": false,

"confidence\_score": 0.95

}

},

"transformation": {

"type": "unit\_standardization",

"conversion\_factor": 1.0,

"rule\_version": "taxonomy\_v1.0.0"

}

},

"ip\_address": "10.0.1.15",

"request\_id": "req\_550e8400-e29b-41d4-a716-446655440001",

"timestamp": "2025-08-27T14:30:00Z",

"severity": "info"

}

```

#### Governance Rules Implementation

```python

class GovernanceEngine:

def \_\_init\_\_(self):

self.rules = self.\_load\_governance\_rules()

self.audit\_logger = AuditLogger()

def validate\_observation(self, observation: Dict) -> ValidationResult:

"""

Apply all governance rules to observation

"""

results = []

# Temporal validation

temporal\_result = self.\_validate\_temporal\_constraints(observation)

results.append(temporal\_result)

# Statistical validation

stats\_result = self.\_validate\_statistical\_bounds(observation)

results.append(stats\_result)

# Referential integrity

ref\_result = self.\_validate\_references(observation)

results.append(ref\_result)

# Business rules

business\_result = self.\_validate\_business\_rules(observation)

results.append(business\_result)

# Log validation activity

self.audit\_logger.log\_validation(observation, results)

return self.\_aggregate\_validation\_results(results)

def \_validate\_temporal\_constraints(self, obs: Dict) -> ValidationResult:

"""

Validate temporal consistency and cutoff rules

"""

checks = []

# Cutoff timestamp validation

if obs.get('observation\_date'):

obs\_date = datetime.strptime(obs['observation\_date'], '%Y-%m-%d').date()

release\_date = datetime.strptime(obs.get('release\_date', obs['observation\_date']), '%Y-%m-%d').date()

if obs\_date > release\_date:

checks.append({

'rule': 'cutoff\_timestamp',

'status': 'fail',

'message': f'Observation date {obs\_date} is after release date {release\_date}'

})

else:

checks.append({

'rule': 'cutoff\_timestamp',

'status': 'pass',

'message': 'Temporal constraints satisfied'

})

# Vintage date validation

if obs.get('vintage\_date'):

vintage\_date = datetime.strptime(obs['vintage\_date'], '%Y-%m-%d').date()

today = datetime.now().date()

if vintage\_date > today:

checks.append({

'rule': 'vintage\_future',

'status': 'fail',

'message': f'Vintage date {vintage\_date} is in the future'

})

else:

checks.append({

'rule': 'vintage\_future',

'status': 'pass',

'message': 'Vintage date valid'

})

return ValidationResult('temporal', checks)

def \_validate\_statistical\_bounds(self, obs: Dict) -> ValidationResult:

"""

Validate statistical outliers and distribution shifts

"""

checks = []

if obs.get('value') is not None:

value = float(obs['value'])

# Get historical context for indicator

history = self.\_get\_indicator\_history(

obs['country\_code'],

obs['indicator\_id'],

lookback\_months=24

)

if len(history) >= 10: # Need minimum history

mean = np.mean(history)

std = np.std(history)

z\_score = abs(value - mean) / std if std > 0 else 0

if z\_score > 3:

checks.append({

'rule': 'outlier\_detection',

'status': 'warning',

'message': f'Value {value} is {z\_score:.2f} standard deviations from mean',

'details': {'z\_score': z\_score, 'mean': mean, 'std': std}

})

else:

checks.append({

'rule': 'outlier\_detection',

'status': 'pass',

'message': f'Value within normal range (z-score: {z\_score:.2f})'

})

else:

checks.append({

'rule': 'outlier\_detection',

'status': 'skip',

'message': 'Insufficient historical data for outlier detection'

})

return ValidationResult('statistical', checks)

def \_validate\_references(self, obs: Dict) -> ValidationResult:

"""

Validate referential integrity across tables

"""

checks = []

# Country code exists

if not self.\_country\_exists(obs.get('country\_code')):

checks.append({

'rule': 'country\_reference',

'status': 'fail',

'message': f'Country code {obs.get("country\_code")} not found in countries table'

})

else:

checks.append({

'rule': 'country\_reference',

'status': 'pass',

'message': 'Country reference valid'

})

# Indicator ID exists

if not self.\_indicator\_exists(obs.get('indicator\_id')):

checks.append({

'rule': 'indicator\_reference',

'status': 'fail',

'message': f'Indicator ID {obs.get("indicator\_id")} not found in indicators table'

})

else:

checks.append({

'rule': 'indicator\_reference',

'status': 'pass',

'message': 'Indicator reference valid'

})

return ValidationResult('referential', checks)

def detect\_duplicates(self, country\_code: str, indicator\_id: int, period: str) -> List[Dict]:

"""

Detect duplicate records across sources and vintages

"""

query = """

SELECT observation\_id, value, vintage\_date, source\_id, content\_hash

FROM observations

WHERE country\_code = %s AND indicator\_id = %s

AND observation\_date = %s

ORDER BY vintage\_date DESC

"""

records = self.db.execute(query, (country\_code, indicator\_id, period)).fetchall()

duplicates = []

seen\_values = {}

for record in records:

key = (record['value'], record['vintage\_date'])

if key in seen\_values:

duplicates.append({

'type': 'exact\_duplicate',

'records': [seen\_values[key], record],

'message': 'Identical value and vintage date'

})

else:

seen\_values[key] = record

return duplicates

# Create triggers for automatic audit logging

CREATE OR REPLACE FUNCTION audit\_trigger\_function()

RETURNS TRIGGER AS $

BEGIN

INSERT INTO audit\_log (entity\_type, entity\_id, action\_type, actor\_type, actor\_id, change\_details)

VALUES (

TG\_TABLE\_NAME,

COALESCE(NEW.observation\_id, NEW.mapping\_id, NEW.id),

TG\_OP,

'system',

'database\_trigger',

jsonb\_build\_object(

'old', to\_jsonb(OLD),

'new', to\_jsonb(NEW),

'table', TG\_TABLE\_NAME

)

);

RETURN COALESCE(NEW, OLD);

END;

$ LANGUAGE plpgsql;

-- Apply triggers to critical tables

CREATE TRIGGER audit\_observations

AFTER INSERT OR UPDATE OR DELETE ON observations

FOR EACH ROW EXECUTE FUNCTION audit\_trigger\_function();

CREATE TRIGGER audit\_mappings

AFTER INSERT OR UPDATE OR DELETE ON country\_indicator\_mapping

FOR EACH ROW EXECUTE FUNCTION audit\_trigger\_function();

```

#### n8n Workflow JSON (Conflict Resolution)

```json

{

"name": "Data Conflict Resolution Workflow",

"nodes": [

{

"parameters": {

"mode": "webhook",

"httpMethod": "POST",

"path": "conflict-detected"

},

"name": "Conflict Webhook",

"type": "n8n-nodes-base.webhook",

"position": [200, 300]

},

{

"parameters": {

"functionCode": "// Extract conflict details from webhook\nconst conflict = $json.body;\n\n// Determine priority based on type and impact\nlet priority = 'medium';\nif (conflict.type === 'source\_mismatch' && conflict.impact\_score > 0.8) {\n priority = 'high';\n}\nif (conflict.type === 'duplicate\_data' && conflict.affected\_count > 100) {\n priority = 'critical';\n}\n\nreturn {json: {...conflict, priority, assigned\_to: 'governance-team'}};"

},

"name": "Classify Conflict",

"type": "n8n-nodes-base.function"

},

{

"parameters": {

"operation": "insert",

"table": "data\_conflicts",

"columns": "conflict\_type,entity\_type,entity\_id,description,priority,assigned\_to,reported\_by",

"values": "={{$json.type}},{{$json.entity\_type}},{{$json.entity\_id}},{{$json.description}},{{$json.priority}},{{$json.assigned\_to}},system"

},

"name": "Log Conflict",

"type": "n8n-nodes-base.postgres",

"credentials": {"postgres": {"id": "supabase\_creds"}}

},

{

"parameters": {

"conditions": {

"string": [{"value1": "={{$json.priority}}", "operation": "equal", "value2": "critical"}]

}

},

"name": "Critical Priority?",

"type": "n8n-nodes-base.if"

},

{

"parameters": {

"url": "https://hooks.slack.com/services/YOUR/SLACK/WEBHOOK",

"requestMethod": "POST",

"jsonBody": "{\"text\": \"🚨 CRITICAL Data Conflict: {{$json.description}}\\nEntity: {{$json.entity\_type}} ID {{$json.entity\_id}}\\nPriority: {{$json.priority}}\"}"

},

"name": "Alert Critical",

"type": "n8n-nodes-base.httpRequest"

},

{

"parameters": {

"subject": "Data Conflict Assigned - Priority {{$json.priority}}",

"text": "A data conflict has been detected and assigned to you:\\n\\nType: {{$json.type}}\\nEntity: {{$json.entity\_type}} ID {{$json.entity\_id}}\\nDescription: {{$json.description}}\\nPriority: {{$json.priority}}\\n\\nPlease review and resolve within the SLA timeframe."

},

"name": "Email Assignment",

"type": "n8n-nodes-base.emailSend"

}

],

"connections": {

"Conflict Webhook": {"main": [[{"node": "Classify Conflict", "type": "main", "index": 0}]]},

"Classify Conflict": {"main": [[{"node": "Log Conflict", "type": "main", "index": 0}]]},

"Log Conflict": {"main": [[{"node": "Critical Priority?", "type": "main", "index": 0}]]},

"Critical Priority?": {

"main": [

[{"node": "Alert Critical", "type": "main", "index": 0}],

[{"node": "Email Assignment", "type": "main", "index": 0}]

]

}

}

}

```

#### Testing Strategy

\*\*Unit Tests (Pytest)\*\*:

```python

def test\_governance\_rules\_enforcement():

"""Test all governance validation rules"""

engine = GovernanceEngine()

# Test temporal validation

future\_obs = {

'observation\_date': '2026-01-01',

'release\_date': '2025-12-15',

'country\_code': 'SWE',

'indicator\_id': 1

}

result = engine.validate\_observation(future\_obs)

assert result.has\_failures()

assert 'cutoff\_timestamp' in [check.rule for check in result.failed\_checks]

def test\_outlier\_detection():

"""Test statistical outlier flagging"""

engine = GovernanceEngine()

# Mock historical data with normal range 1.5-2.5

engine.\_get\_indicator\_history = lambda c, i, l: [2.0, 2.1, 1.9, 2.2, 1.8, 2.0, 2.1]

outlier\_obs = {

'value': 8.5, # Clear outlier

'country\_code': 'SWE',

'indicator\_id': 1

}

result = engine.validate\_observation(outlier\_obs)

assert any('outlier' in check.rule for check in result.warnings)

def test\_duplicate\_detection():

"""Test duplicate record identification"""

engine = GovernanceEngine()

duplicates = engine.detect\_duplicates('SWE', 1, '2024-12-01')

# Assert duplicate detection logic

```

### Definition of Done (Gate Criteria)

- [ ] All governance validation rules implemented and tested

- [ ] Audit logging captures 100% of data modifications with complete provenance

- [ ] Conflict resolution workflow operational with SLA tracking

- [ ] Data quality metrics calculated for all country-indicator pairs

- [ ] Integrity checks run successfully on full dataset

- [ ] Immutable snapshot system creates daily backups

- [ ] Unit tests cover all governance rules (>90% coverage)

- [ ] Integration tests validate end-to-end audit trail

### Handover/Context Transfer

\*\*Outputs for Phase 5\*\*:

| Artifact | Schema/Format | Consumer |

|----------|---------------|----------|

| Validated observations | `observations` with quality flags | Forecast engine |

| Audit trail | `audit\_log` table | Evaluation framework |

| Data quality metrics | `data\_quality\_metrics` table | Reporting system |

| Governance manifests | JSON with validation rules | Reproducibility system |

---

## Phase 5 — Evaluation Framework (LLM & Forecast Scoring)

### Objective & Scope

Implement comprehensive evaluation system for LLM forecasts including numerical accuracy metrics, coherence scoring, and multi-model leaderboard generation with reproducible scoring methodology.

### Dependencies (from Phase 4)

- Validated observations in `observations` table with governance approval

- Data quality metrics and audit trails operational

- Country-indicator mappings finalized and versioned

- Governance policies active and enforcing data integrity

### Functional Requirements

\*\*FR-5.1\*\*: The system MUST implement numerical accuracy metrics: MAE, RMSE, sMAPE, MAPE, Directional Accuracy

\*\*FR-5.2\*\*: The system MUST implement coherence scoring with weighted components: Data Relevance (40%), Logical Structure (40%), Economic Plausibility (20%)

\*\*FR-5.3\*\*: The system MUST support evaluation for at least 3 LLMs (GPT, Claude, Gemini) and 1 baseline model (ARIMA)

\*\*FR-5.4\*\*: The system MUST handle both first-release and latest-vintage scoring modes

\*\*FR-5.5\*\*: The system MUST generate composite scores combining numerical accuracy (70%) and coherence (30%)

\*\*FR-5.6\*\*: The system MUST create leaderboards aggregated by model, country, category, and global rankings

\*\*FR-5.7\*\*: The system MUST store complete evaluation provenance including model versions, data vintages, and scoring parameters

\*\*FR-5.8\*\*: The system MUST handle missing actuals gracefully with partial scoring and coverage weighting

### Non-Functional Requirements

- Evaluation processing time: ≤5 seconds per prediction

- Coherence scoring latency: ≤10 seconds per reasoning text

- Leaderboard refresh: ≤30 seconds for global rankings

- Scoring reproducibility: 100% identical results for same inputs

### Deliverables

#### SQL DDL (Evaluation Tables)

```sql

-- Model registry and metadata

CREATE TABLE models (

model\_id SERIAL PRIMARY KEY,

model\_name VARCHAR(100) NOT NULL,

model\_vendor VARCHAR(100) NOT NULL,

model\_version VARCHAR(50) NOT NULL,

model\_type VARCHAR(20) NOT NULL CHECK (model\_type IN ('llm', 'classical', 'hybrid')),

api\_endpoint TEXT,

context\_limit INTEGER,

cost\_per\_1k\_tokens DECIMAL(8,4),

capabilities JSONB DEFAULT '{}',

is\_active BOOLEAN DEFAULT TRUE,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

UNIQUE(model\_name, model\_version)

);

-- Forecast tasks and context

CREATE TABLE forecast\_tasks (

task\_id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

country\_code VARCHAR(3) NOT NULL,

category VARCHAR(20) NOT NULL,

indicator\_id INTEGER NOT NULL,

target\_period DATE NOT NULL,

horizon\_days INTEGER NOT NULL,

cutoff\_timestamp TIMESTAMP WITH TIME ZONE NOT NULL,

context\_data JSONB NOT NULL, -- Historical data provided to models

correlates JSONB DEFAULT '[]', -- Additional context indicators

task\_manifest\_id UUID NOT NULL,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

FOREIGN KEY (country\_code) REFERENCES countries(country\_code),

FOREIGN KEY (indicator\_id) REFERENCES indicators(indicator\_id)

);

-- Model predictions

CREATE TABLE predictions (

prediction\_id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

task\_id UUID NOT NULL,

model\_id INTEGER NOT NULL,

prediction\_value DECIMAL(15,6),

prediction\_unit VARCHAR(50) NOT NULL,

confidence\_interval\_lower DECIMAL(15,6),

confidence\_interval\_upper DECIMAL(15,6),

reasoning\_text TEXT,

reasoning\_storage\_path TEXT,

model\_response\_raw JSONB,

prediction\_timestamp TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

processing\_time\_ms INTEGER,

token\_usage JSONB,

FOREIGN KEY (task\_id) REFERENCES forecast\_tasks(task\_id),

FOREIGN KEY (model\_id) REFERENCES models(model\_id)

);

-- Evaluation results

CREATE TABLE evaluations (

evaluation\_id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

prediction\_id UUID NOT NULL,

actual\_value DECIMAL(15,6),

actual\_vintage\_mode VARCHAR(20) NOT NULL CHECK (actual\_vintage\_mode IN ('first\_release', 'latest')),

-- Numerical metrics

mae DECIMAL(10,6),

rmse DECIMAL(10,6),

mape DECIMAL(10,6),

smape DECIMAL(10,6),

directional\_accuracy DECIMAL(3,2), -- 0.00 to 1.00

-- Coherence components

data\_relevance\_score DECIMAL(5,2), -- 0.00 to 100.00

logical\_structure\_score DECIMAL(5,2),

economic\_plausibility\_score DECIMAL(5,2),

coherence\_composite DECIMAL(5,2),

-- Final scores

numerical\_accuracy\_normalized DECIMAL(5,2),

composite\_score DECIMAL(5,2),

evaluation\_timestamp TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

evaluation\_version VARCHAR(20) NOT NULL,

evaluator\_id VARCHAR(100) NOT NULL,

FOREIGN KEY (prediction\_id) REFERENCES predictions(prediction\_id)

);

-- Leaderboard materialized views for performance

CREATE MATERIALIZED VIEW leaderboard\_global AS

SELECT

m.model\_name,

m.model\_vendor,

m.model\_version,

COUNT(e.evaluation\_id) as prediction\_count,

AVG(e.composite\_score) as avg\_composite\_score,

AVG(e.numerical\_accuracy\_normalized) as avg\_numerical\_accuracy,

AVG(e.coherence\_composite) as avg\_coherence\_score,

STDDEV(e.composite\_score) as score\_std\_dev,

MIN(e.evaluation\_timestamp) as first\_evaluation,

MAX(e.evaluation\_timestamp) as latest\_evaluation

FROM models m

JOIN predictions p ON m.model\_id = p.model\_id

JOIN evaluations e ON p.prediction\_id = e.prediction\_id

WHERE e.evaluation\_timestamp >= NOW() - INTERVAL '90 days'

GROUP BY m.model\_id, m.model\_name, m.model\_vendor, m.model\_version

HAVING COUNT(e.evaluation\_id) >= 10

ORDER BY avg\_composite\_score DESC;

-- Category-specific leaderboards

CREATE MATERIALIZED VIEW leaderboard\_by\_category AS

SELECT

ft.category,

m.model\_name,

COUNT(e.evaluation\_id) as prediction\_count,

AVG(e.composite\_score) as avg\_composite\_score,

ROW\_NUMBER() OVER (PARTITION BY ft.category ORDER BY AVG(e.composite\_score) DESC) as rank

FROM forecast\_tasks ft

JOIN predictions p ON ft.task\_id = p.task\_id

JOIN models m ON p.model\_id = m.model\_id

JOIN evaluations e ON p.prediction\_id = e.prediction\_id

WHERE e.evaluation\_timestamp >= NOW() - INTERVAL '90 days'

GROUP BY ft.category, m.model\_id, m.model\_name

HAVING COUNT(e.evaluation\_id) >= 5

ORDER BY ft.category, rank;

-- Insert baseline models

INSERT INTO models (model\_name, model\_vendor, model\_version, model\_type, capabilities) VALUES

('GPT-4', 'OpenAI', '4.0', 'llm', '{"reasoning": true, "context\_limit": 128000}'),

('Claude-3.5-Sonnet', 'Anthropic', '3.5', 'llm', '{"reasoning": true, "context\_limit": 200000}'),

('Gemini-1.5-Pro', 'Google', '1.5', 'llm', '{"reasoning": true, "context\_limit": 1000000}'),

('ARIMA-Baseline', 'Internal', '1.0', 'classical', '{"statistical": true, "uncertainty": true}');

```

#### JSON Schema (Forecast Request/Response)

```json

{

"$schema": "http://json-schema.org/draft-07/schema#",

"title": "ForecastRequest",

"type": "object",

"properties": {

"task\_id": {"type": "string", "format": "uuid"},

"model\_id": {"type": "integer"},

"country\_code": {"type": "string", "pattern": "^[A-Z]{3}$"},

"category": {"type": "string", "enum": ["Growth", "Prices", "Labor", "Trade", "Sentiment"]},

"target\_indicator": {"type": "string"},

"target\_period": {"type": "string", "format": "date"},

"horizon\_days": {"type": "integer", "minimum": 1},

"cutoff\_timestamp": {"type": "string", "format": "date-time"},

"context\_data": {

"type": "object",

"properties": {

"main\_series": {

"type": "array",

"items": {

"type": "object",

"properties": {

"date": {"type": "string", "format": "date"},

"value": {"type": "number"},

"unit": {"type": "string"}

}

}

},

"correlates": {

"type": "array",

"items": {

"type": "object",

"properties": {

"indicator\_name": {"type": "string"},

"data": {"type": "array"}

}

}

}

}

}

},

"required": ["task\_id", "model\_id", "country\_code", "category", "target\_period", "context\_data"]

}

```

#### Sample JSON (Evaluation Result)

```json

{

"evaluation\_id": "550e8400-e29b-41d4-a716-446655440000",

"prediction\_id": "550e8400-e29b-41d4-a716-446655440001",

"task\_metadata": {

"country\_code": "SWE",

"category": "Prices",

"target\_period": "2025-07-01",

"model\_name": "GPT-4"

},

"prediction": {

"value": 2.3,

"unit": "% y/y",

"timestamp": "2025-06-25T10:00:00Z"

},

"actual": {

"value": 2.1,

"vintage\_mode": "first\_release",

"release\_date": "2025-08-15"

},

"numerical\_metrics": {

"mae": 0.2,

"rmse": 0.2,

"mape": 9.52,

"smape": 9.09,

"directional\_accuracy": 1.0

},

"coherence\_scores": {

"data\_relevance": 85.0,

"logical\_structure": 90.0,

"economic\_plausibility": 95.0,

"composite": 89.5

},

"final\_scores": {

"numerical\_accuracy\_normalized": 87.3,

"coherence\_composite": 89.5,

"composite\_score": 88.1

},

"evaluation\_metadata": {

"evaluation\_version": "1.0.0",

"evaluated\_at": "2025-08-15T14:30:00Z",

"evaluator\_id": "evaluation-engine-v1.2"

}

}

```

#### Coherence Scoring Implementation

```python

class CoherenceScorer:

def \_\_init\_\_(self):

self.weights = {

'data\_relevance': 0.40,

'logical\_structure': 0.40,

'economic\_plausibility': 0.20

}

def score\_coherence(self, reasoning\_text: str, context: Dict, category: str) -> Dict[str, float]:

"""

Score reasoning coherence across three dimensions

"""

scores = {

'data\_relevance': self.\_score\_data\_relevance(reasoning\_text, context),

'logical\_structure': self.\_score\_logical\_structure(reasoning\_text),

'economic\_plausibility': self.\_score\_economic\_plausibility(reasoning\_text, category)

}

# Calculate weighted composite

composite = sum(scores[dim] \* self.weights[dim] for dim in scores)

scores['composite'] = composite

return scores

def \_score\_data\_relevance(self, text: str, context: Dict) -> float:

"""

Score based on reference to provided data points (0-100)

"""

score = 0.0

max\_score = 100.0

# Check for main indicator references

main\_series = context.get('main\_series', [])

if main\_series:

recent\_values = [point['value'] for point in main\_series[-6:]]

mentioned\_values = self.\_extract\_numeric\_mentions(text)

# Award points for mentioning recent values within 10% tolerance

for value in recent\_values[-3:]: # Focus on most recent 3 points

for mentioned in mentioned\_values:

if abs(mentioned - value) / value <= 0.10:

score += 20.0

break

# Check for explicit date references

recent\_dates = [point['date'] for point in main\_series[-3:]]

date\_mentions = self.\_extract\_date\_mentions(text)

overlap = len(set(recent\_dates) & set(date\_mentions))

score += min(20.0, overlap \* 10.0)

# Check for correlate indicator mentions

correlates = context.get('correlates', [])

correlate\_names = [c['indicator\_name'] for c in correlates]

mentioned\_indicators = self.\_extract\_indicator\_mentions(text)

correlate\_overlap = len(set(correlate\_names) & set(mentioned\_indicators))

score += min(30.0, correlate\_overlap \* 15.0)

return min(max\_score, score)

def \_score\_logical\_structure(self, text: str) -> float:

"""

Score logical flow and structure (0-100)

"""

score = 0.0

# Check for structured reasoning patterns

structure\_patterns = [

r'(first|firstly|1\.|step 1)',

r'(second|secondly|2\.|step 2|next|then)',

r'(finally|lastly|therefore|conclusion|thus)',

r'(because|due to|as a result|consequently)',

r'(however|but|although|despite)'

]

pattern\_matches = 0

for pattern in structure\_patterns:

if re.search(pattern, text, re.IGNORECASE):

pattern\_matches += 1

# Award points for structured reasoning

score += min(40.0, pattern\_matches \* 10.0)

# Check for contradiction detection

contradiction\_score = self.\_detect\_contradictions(text)

score += contradiction\_score # 0-30 points

# Check for evidence-conclusion linkage

linkage\_score = self.\_score\_evidence\_linkage(text)

score += linkage\_score # 0-30 points

return min(100.0, score)

def \_score\_economic\_plausibility(self, text: str, category: str) -> float:

"""

Score economic reasoning plausibility by category (0-100)

"""

economic\_rules = {

'Prices': [

('oil prices.\*increase.\*inflation', 20),

('supply.\*constrain.\*price.\*increase', 15),

('demand.\*strong.\*price.\*pressure', 15),

('central bank.\*tight.\*monetary.\*policy', 15),

('wage.\*growth.\*inflation.\*expectation', 10)

],

'Growth': [

('consumer.\*spending.\*drive.\*growth', 20),

('investment.\*capital.\*expansion', 15),

('export.\*trade.\*surplus.\*growth', 15),

('employment.\*labor.\*market.\*growth', 15),

('government.\*fiscal.\*stimulus', 10)

],

'Labor': [

('economic.\*growth.\*employment', 20),

('productivity.\*wage.\*growth', 15),

('demographic.\*labor.\*force', 15),

('automation.\*job.\*displacement', 10),

('education.\*skill.\*employment', 10)

],

'Trade': [

('exchange.\*rate.\*export.\*competitive', 20),

('tariff.\*import.\*restriction', 15),

('commodity.\*price.\*trade.\*balance', 15),

('global.\*demand.\*export', 15),

('supply.\*chain.\*trade.\*flow', 10)

],

'Sentiment': [

('economic.\*outlook.\*confidence', 25),

('uncertainty.\*consumer.\*sentiment', 20),

('inflation.\*expectation.\*confidence', 15),

('employment.\*security.\*sentiment', 15),

('stock.\*market.\*consumer.\*confidence', 10)

]

}

rules = economic\_rules.get(category, [])

score = 0.0

for pattern, points in rules:

if re.search(pattern, text, re.IGNORECASE):

score += points

# Cap at 100 and ensure minimum plausibility score

return min(100.0, max(20.0, score)) # Minimum 20 for any coherent text

class NumericalEvaluator:

def \_\_init\_\_(self):

pass

def calculate\_metrics(self, predicted: float, actual: float, previous\_actual: float = None) -> Dict[str, float]:

"""

Calculate all numerical accuracy metrics

"""

metrics = {}

# Mean Absolute Error

metrics['mae'] = abs(predicted - actual)

# Root Mean Square Error (single prediction)

metrics['rmse'] = abs(predicted - actual)

# Mean Absolute Percentage Error

if actual != 0:

metrics['mape'] = abs((predicted - actual) / actual) \* 100

else:

metrics['mape'] = None

# Symmetric Mean Absolute Percentage Error

denominator = (abs(predicted) + abs(actual)) / 2

if denominator > 0:

metrics['smape'] = abs(predicted - actual) / denominator \* 100

else:

metrics['smape'] = 0.0

# Directional Accuracy

if previous\_actual is not None:

predicted\_direction = 1 if predicted > previous\_actual else -1 if predicted < previous\_actual else 0

actual\_direction = 1 if actual > previous\_actual else -1 if actual < previous\_actual else 0

metrics['directional\_accuracy'] = 1.0 if predicted\_direction == actual\_direction else 0.0

else:

metrics['directional\_accuracy'] = None

return metrics

def normalize\_scores(self, metrics: Dict[str, float], category\_benchmarks: Dict[str, Dict]) -> float:

"""

Normalize numerical metrics to 0-100 scale using category benchmarks

"""

# Use category-specific benchmarks for normalization

benchmarks = category\_benchmarks.get('default', {

'mae\_excellent': 0.1,

'mae\_poor': 2.0,

'smape\_excellent': 5.0,

'smape\_poor': 50.0

})

scores = []

# MAE normalization (lower is better)

if metrics.get('mae') is not None:

mae\_score = max(0, min(100, 100 \* (1 - (metrics['mae'] - benchmarks['mae\_excellent']) /

(benchmarks['mae\_poor'] - benchmarks['mae\_excellent']))))

scores.append(mae\_score)

# sMAPE normalization (lower is better)

if metrics.get('smape') is not None:

smape\_score = max(0, min(100, 100 \* (1 - (metrics['smape'] - benchmarks['smape\_excellent']) /

(benchmarks['smape\_poor'] - benchmarks['smape\_excellent']))))

scores.append(smape\_score)

# Directional accuracy (higher is better, already 0-1)

if metrics.get('directional\_accuracy') is not None:

scores.append(metrics['directional\_accuracy'] \* 100)

return sum(scores) / len(scores) if scores else 0.0

```

#### Testing Strategy

\*\*Unit Tests (Pytest)\*\*:

```python

def test\_coherence\_scoring():

"""Test coherence scoring components"""

scorer = CoherenceScorer()

# Test data relevance scoring

good\_reasoning = "Looking at the recent CPI data showing 2.1% in December and the core CPI at 1.8%, combined with oil prices rising 5% last month..."

context = {

'main\_series': [{'date': '2024-12-01', 'value': 2.1}],

'correlates': [{'indicator\_name': 'Core CPI'}, {'indicator\_name': 'Oil Prices'}]

}

scores = scorer.score\_coherence(good\_reasoning, context, 'Prices')

assert scores['data\_relevance'] > 70.0

assert scores['composite'] > 0.0

def test\_numerical\_metrics():

"""Test numerical accuracy calculations"""

evaluator = NumericalEvaluator()

metrics = evaluator.calculate\_metrics(predicted=2.3, actual=2.1, previous\_actual=2.0)

assert metrics['mae'] == 0.2

assert abs(metrics['mape'] - 9.52) < 0.1

assert metrics['directional\_accuracy'] == 1.0 # Both increased

def test\_evaluation\_reproducibility():

"""Test evaluation determinism"""

# Same inputs should produce identical scores

scorer = CoherenceScorer()

evaluator = NumericalEvaluator()

# Run evaluation twice with identical inputs

result1 = run\_full\_evaluation(test\_prediction, test\_actual, test\_context)

result2 = run\_full\_evaluation(test\_prediction, test\_actual, test\_context)

assert result1 == result2

```

### Definition of Done (Gate Criteria)

- [ ] Numerical metrics calculated for all prediction-actual pairs

- [ ] Coherence scoring operational with >0.7 inter-rater correlation vs human baseline

- [ ] Composite scoring combines numerical (70%) and coherence (30%) accurately

- [ ] Leaderboards generate for global, category, and country dimensions

- [ ] Evaluation supports both first-release and latest-vintage modes

- [ ] All model types (LLM and classical) integrate successfully

- [ ] Unit tests achieve >90% coverage of scoring logic

- [ ] Performance meets latency requirements (<5s per prediction)

### Handover/Context Transfer

\*\*Outputs for Phase 6\*\*:

| Artifact | Schema/Format | Consumer |

|----------|---------------|----------|

| Evaluations database | `evaluations` table | Orchestration engine |

| Leaderboard views | Materialized views | API layer |

| Scoring methodology | JSON manifest | Reproducibility system |

| Model performance data | Aggregated metrics | Dashboard system |

---

## Phase 6 — Orchestration & Automation

### Objective & Scope

Implement end-to-end pipeline orchestration connecting ingestion through evaluation, with error handling, recovery mechanisms, and operational monitoring for autonomous operation.

###

Dependencies (from Phase 5)

Evaluation framework operational with scoring for all model types

Leaderboard views and materialized aggregations available

Model registry populated with LLM and baseline configurations

Complete evaluation pipeline tested and validated

Functional Requirements

FR-6.1: The system MUST orchestrate end-to-end pipeline: ingestion → normalization → mapping → governance → forecasting → evaluation

FR-6.2: The system MUST implement idempotency keys to prevent duplicate processing on retries

FR-6.3: The system MUST provide dead-letter queues for failed jobs with manual recovery capabilities

FR-6.4: The system MUST implement circuit breakers for external API calls with configurable thresholds

FR-6.5: The system MUST support both scheduled (monthly evaluation cycles) and on-demand execution modes

FR-6.6: The system MUST implement compensating transactions for partial failures with rollback capabilities

FR-6.7: The system MUST provide operational dashboards for pipeline monitoring and alerting

FR-6.8: The system MUST implement graceful degradation when external services are unavailable

Non-Functional Requirements

End-to-end pipeline completion: ≤60 minutes for MVP scope (4 countries × 5 categories × 4 models)

Error recovery time: ≤10 minutes for transient failures

Pipeline availability: 99.5% during scheduled execution windows

Monitoring latency: ≤30 seconds for status updates

Deliverables

Master Orchestration Workflow (n8n JSON)

json{

"name": "GlobalPulse Master Pipeline",

"meta": {

"instanceId": "globalpulse-production"

},

"nodes": [

{

"parameters": {

"rule": {

"interval": [{"field": "cronExpression", "value": "0 2 1 \* \*"}]

}

},

"name": "Monthly Evaluation Trigger",

"type": "n8n-nodes-base.cron",

"position": [200, 300],

"id": "monthly-trigger"

},

{

"parameters": {

"functionCode": "// Generate pipeline execution manifest\nconst runId = $evaluateExpression('{{$randomString(16)}}');\nconst countries = ['SWE', 'MEX', 'NZL', 'THA'];\nconst categories = ['Growth', 'Prices', 'Labor', 'Trade', 'Sentiment'];\nconst models = [1, 2, 3, 4]; // Model IDs\n\nconst manifest = {\n run\_id: runId,\n execution\_mode: 'scheduled',\n countries: countries,\n categories: categories,\n models: models,\n vintage\_mode: 'first\_release',\n started\_at: new Date().toISOString(),\n status: 'initializing'\n};\n\nreturn { json: manifest };"

},

"name": "Initialize Pipeline",

"type": "n8n-nodes-base.function",

"position": [400, 300]

},

{

"parameters": {

"operation": "insert",

"table": "pipeline\_executions",

"values": {

"run\_id": "={{$json.run\_id}}",

"execution\_mode": "={{$json.execution\_mode}}",

"manifest": "={{$json}}",

"status": "running",

"started\_at": "={{$json.started\_at}}"

}

},

"name": "Log Pipeline Start",

"type": "n8n-nodes-base.postgres",

"credentials": {"postgres": {"id": "supabase\_creds"}},

"position": [600, 300]

},

{

"parameters": {

"url": "http://ingestor-service:8000/api/v1/ingest/run",

"requestMethod": "POST",

"jsonBody": "{\"run\_id\": \"{{$json.run\_id}}\", \"countries\": {{$json.countries}}}",

"options": {

"timeout": 300000,

"retry": {"enabled": true, "maxTries": 3}

}

},

"name": "Execute Ingestion",

"type": "n8n-nodes-base.httpRequest",

"position": [800, 300]

},

{

"parameters": {

"conditions": {

"number": [{"value1": "={{$statusCode}}", "operation": "equal", "value2": 200}]

}

},

"name": "Ingestion Success?",

"type": "n8n-nodes-base.if",

"position": [1000, 300]

},

{

"parameters": {

"url": "http://normalizer-service:8000/api/v1/normalize/run",

"requestMethod": "POST",

"jsonBody": "{\"run\_id\": \"{{$json.run\_id}}\", \"source\_run\_id\": \"{{$json.run\_id}}\"}",

"options": {

"timeout": 300000,

"retry": {"enabled": true, "maxTries": 2}

}

},

"name": "Execute Normalization",

"type": "n8n-nodes-base.httpRequest",

"position": [1200, 200]

},

{

"parameters": {

"url": "http://mapper-service:8000/api/v1/map/run",

"requestMethod": "POST",

"jsonBody": "{\"run\_id\": \"{{$json.run\_id}}\", \"countries\": {{$json.countries}}}",

"options": {"timeout": 180000}

},

"name": "Execute Mapping",

"type": "n8n-nodes-base.httpRequest",

"position": [1400, 200]

},

{

"parameters": {

"url": "http://governance-service:8000/api/v1/validate/run",

"requestMethod": "POST",

"jsonBody": "{\"run\_id\": \"{{$json.run\_id}}\", \"validation\_level\": \"full\"}",

"options": {"timeout": 600000}

},

"name": "Execute Governance",

"type": "n8n-nodes-base.httpRequest",

"position": [1600, 200]

},

{

"parameters": {

"batchSize": 5,

"options": {"continueOnFail": true}

},

"name": "Batch Forecast Tasks",

"type": "n8n-nodes-base.splitInBatches",

"position": [1800, 200]

},

{

"parameters": {

"url": "http://forecaster-service:8000/api/v1/forecast/batch",

"requestMethod": "POST",

"jsonBody": "{\"run\_id\": \"{{$json.run\_id}}\", \"tasks\": {{$json.forecast\_tasks}}}",

"options": {

"timeout": 1800000,

"retry": {"enabled": true, "maxTries": 2}

}

},

"name": "Execute Forecasting",

"type": "n8n-nodes-base.httpRequest",

"position": [2000, 200]

},

{

"parameters": {

"url": "http://evaluator-service:8000/api/v1/evaluate/run",

"requestMethod": "POST",

"jsonBody": "{\"run\_id\": \"{{$json.run\_id}}\", \"vintage\_mode\": \"first\_release\"}",

"options": {"timeout": 900000}

},

"name": "Execute Evaluation",

"type": "n8n-nodes-base.httpRequest",

"position": [2200, 200]

},

{

"parameters": {

"operation": "update",

"table": "pipeline\_executions",

"where": {"run\_id": "={{$json.run\_id}}"},

"values": {

"status": "completed",

"completed\_at": "={{new Date().toISOString()}}",

"success\_count": "={{$json.evaluations\_created}}"

}

},

"name": "Mark Success",

"type": "n8n-nodes-base.postgres",

"position": [2400, 200]

},

{

"parameters": {

"operation": "update",

"table": "pipeline\_executions",

"where": {"run\_id": "={{$json.run\_id}}"},

"values": {

"status": "failed",

"failed\_at": "={{new Date().toISOString()}}",

"error\_details": "={{$json.error}}"

}

},

"name": "Mark Failure",

"type": "n8n-nodes-base.postgres",

"position": [1200, 400]

},

{

"parameters": {

"url": "https://hooks.slack.com/services/YOUR/SLACK/WEBHOOK",

"requestMethod": "POST",

"jsonBody": "{\"text\": \"🚨 GlobalPulse Pipeline Failed\\nRun ID: {{$json.run\_id}}\\nStage: {{$json.failed\_stage}}\\nError: {{$json.error}}\"}"

},

"name": "Alert Failure",

"type": "n8n-nodes-base.httpRequest",

"position": [1400, 400]

},

{

"parameters": {

"url": "http://dashboard-service:8000/api/v1/refresh/leaderboards",

"requestMethod": "POST",

"jsonBody": "{\"run\_id\": \"{{$json.run\_id}}\"}",

"options": {"timeout": 120000}

},

"name": "Refresh Dashboards",

"type": "n8n-nodes-base.httpRequest",

"position": [2600, 200]

}

],

"connections": {

"Monthly Evaluation Trigger": {"main": [[{"node": "Initialize Pipeline", "type": "main", "index": 0}]]},

"Initialize Pipeline": {"main": [[{"node": "Log Pipeline Start", "type": "main", "index": 0}]]},

"Log Pipeline Start": {"main": [[{"node": "Execute Ingestion", "type": "main", "index": 0}]]},

"Execute Ingestion": {"main": [[{"node": "Ingestion Success?", "type": "main", "index": 0}]]},

"Ingestion Success?": {

"main": [

[{"node": "Execute Normalization", "type": "main", "index": 0}],

[{"node": "Mark Failure", "type": "main", "index": 0}]

]

},

"Execute Normalization": {"main": [[{"node": "Execute Mapping", "type": "main", "index": 0}]]},

"Execute Mapping": {"main": [[{"node": "Execute Governance", "type": "main", "index": 0}]]},

"Execute Governance": {"main": [[{"node": "Batch Forecast Tasks", "type": "main", "index": 0}]]},

"Batch Forecast Tasks": {"main": [[{"node": "Execute Forecasting", "type": "main", "index": 0}]]},

"Execute Forecasting": {"main": [[{"node": "Execute Evaluation", "type": "main", "index": 0}]]},

"Execute Evaluation": {"main": [[{"node": "Mark Success", "type": "main", "index": 0}]]},

"Mark Success": {"main": [[{"node": "Refresh Dashboards", "type": "main", "index": 0}]]},

"Mark Failure": {"main": [[{"node": "Alert Failure", "type": "main", "index": 0}]]}

},

"settings": {

"executionOrder": "v1",

"saveManualExecutions": true,

"callerPolicy": "workflowsFromSameOwner",

"errorWorkflow": "error-handler-workflow"

}

}

Pipeline Execution Tracking (SQL DDL)

sql-- Pipeline execution tracking

CREATE TABLE pipeline\_executions (

execution\_id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

run\_id VARCHAR(100) NOT NULL UNIQUE,

execution\_mode VARCHAR(20) NOT NULL CHECK (execution\_mode IN ('scheduled', 'manual', 'recovery')),

manifest JSONB NOT NULL,

status VARCHAR(20) NOT NULL DEFAULT 'initializing'

CHECK (status IN ('initializing', 'running', 'completed', 'failed', 'cancelled')),

started\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

completed\_at TIMESTAMP WITH TIME ZONE,

failed\_at TIMESTAMP WITH TIME ZONE,

success\_count INTEGER DEFAULT 0,

error\_count INTEGER DEFAULT 0,

error\_details JSONB,

total\_duration\_ms BIGINT,

initiated\_by VARCHAR(100) NOT NULL

);

-- Stage-level execution tracking

CREATE TABLE pipeline\_stages (

stage\_id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

execution\_id UUID NOT NULL,

stage\_name VARCHAR(50) NOT NULL,

stage\_order INTEGER NOT NULL,

status VARCHAR(20) NOT NULL DEFAULT 'pending'

CHECK (status IN ('pending', 'running', 'completed', 'failed', 'skipped')),

started\_at TIMESTAMP WITH TIME ZONE,

completed\_at TIMESTAMP WITH TIME ZONE,

duration\_ms INTEGER,

input\_records INTEGER DEFAULT 0,

output\_records INTEGER DEFAULT 0,

error\_message TEXT,

retry\_count INTEGER DEFAULT 0,

FOREIGN KEY (execution\_id) REFERENCES pipeline\_executions(execution\_id)

);

-- Dead letter queue for failed jobs

CREATE TABLE dead\_letter\_queue (

dlq\_id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

execution\_id UUID,

stage\_name VARCHAR(50) NOT NULL,

job\_type VARCHAR(50) NOT NULL,

job\_payload JSONB NOT NULL,

original\_error TEXT NOT NULL,

failed\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

retry\_count INTEGER DEFAULT 0,

last\_retry\_at TIMESTAMP WITH TIME ZONE,

status VARCHAR(20) DEFAULT 'queued'

CHECK (status IN ('queued', 'retrying', 'resolved', 'abandoned')),

FOREIGN KEY (execution\_id) REFERENCES pipeline\_executions(execution\_id)

);

-- Circuit breaker state tracking

CREATE TABLE circuit\_breakers (

circuit\_id SERIAL PRIMARY KEY,

service\_name VARCHAR(100) NOT NULL UNIQUE,

status VARCHAR(10) NOT NULL DEFAULT 'closed'

CHECK (status IN ('closed', 'open', 'half\_open')),

failure\_count INTEGER DEFAULT 0,

last\_failure\_at TIMESTAMP WITH TIME ZONE,

opened\_at TIMESTAMP WITH TIME ZONE,

next\_retry\_at TIMESTAMP WITH TIME ZONE,

failure\_threshold INTEGER DEFAULT 5,

timeout\_ms INTEGER DEFAULT 300000,

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

);

-- Initialize circuit breakers for external services

INSERT INTO circuit\_breakers (service\_name, failure\_threshold, timeout\_ms) VALUES

('tradingeconomics\_api', 5, 60000),

('openai\_api', 3, 30000),

('anthropic\_api', 3, 30000),

('google\_api', 3, 30000);

Error Handling & Recovery Implementation

pythonclass PipelineOrchestrator:

def \_\_init\_\_(self):

self.circuit\_breakers = CircuitBreakerManager()

self.dlq = DeadLetterQueue()

self.idempotency = IdempotencyManager()

async def execute\_pipeline(self, run\_id: str, execution\_mode: str = 'scheduled') -> PipelineResult:

"""

Execute complete pipeline with error handling and recovery

"""

execution = await self.\_initialize\_execution(run\_id, execution\_mode)

try:

# Stage 1: Ingestion

await self.\_execute\_stage(execution, 'ingestion', self.\_run\_ingestion)

# Stage 2: Normalization

await self.\_execute\_stage(execution, 'normalization', self.\_run\_normalization)

# Stage 3: Mapping

await self.\_execute\_stage(execution, 'mapping', self.\_run\_mapping)

# Stage 4: Governance

await self.\_execute\_stage(execution, 'governance', self.\_run\_governance)

# Stage 5: Forecasting (with batching)

await self.\_execute\_stage(execution, 'forecasting', self.\_run\_forecasting\_batched)

# Stage 6: Evaluation

await self.\_execute\_stage(execution, 'evaluation', self.\_run\_evaluation)

# Stage 7: Dashboard Refresh

await self.\_execute\_stage(execution, 'dashboard\_refresh', self.\_refresh\_dashboards)

await self.\_mark\_execution\_success(execution)

return PipelineResult(status='completed', execution\_id=execution.id)

except PipelineStageError as e:

await self.\_handle\_stage\_failure(execution, e)

return PipelineResult(status='failed', error=str(e))

except Exception as e:

await self.\_handle\_unexpected\_error(execution, e)

return PipelineResult(status='failed', error=str(e))

async def \_execute\_stage(self, execution: PipelineExecution, stage\_name: str, stage\_func: callable):

"""

Execute pipeline stage with retry logic and error handling

"""

stage = await self.\_create\_stage\_record(execution.id, stage\_name)

try:

# Check idempotency - skip if already completed successfully

if await self.idempotency.is\_completed(execution.run\_id, stage\_name):

await self.\_mark\_stage\_skipped(stage, "Already completed")

return

await self.\_mark\_stage\_running(stage)

# Execute stage with circuit breaker protection

result = await self.\_execute\_with\_circuit\_breaker(stage\_name, stage\_func, execution)

await self.\_mark\_stage\_completed(stage, result)

await self.idempotency.mark\_completed(execution.run\_id, stage\_name, result)

except CircuitBreakerOpenError as e:

await self.\_mark\_stage\_failed(stage, f"Circuit breaker open: {e}")

await self.\_send\_to\_dlq(execution, stage\_name, e)

raise PipelineStageError(f"Stage {stage\_name} failed due to circuit breaker")

except RetryableError as e:

retry\_count = stage.retry\_count + 1

if retry\_count <= MAX\_RETRIES:

await asyncio.sleep(self.\_calculate\_backoff(retry\_count))

stage.retry\_count = retry\_count

await self.\_execute\_stage(execution, stage\_name, stage\_func) # Recursive retry

else:

await self.\_mark\_stage\_failed(stage, f"Max retries exceeded: {e}")

await self.\_send\_to\_dlq(execution, stage\_name, e)

raise PipelineStageError(f"Stage {stage\_name} failed after {retry\_count} retries")

except NonRetryableError as e:

await self.\_mark\_stage\_failed(stage, str(e))

raise PipelineStageError(f"Stage {stage\_name} failed with non-retryable error: {e}")

async def \_run\_forecasting\_batched(self, execution: PipelineExecution):

"""

Execute forecasting with batching and parallel processing

"""

# Generate forecast tasks

tasks = await self.\_generate\_forecast\_tasks(execution)

# Split into batches of 5 for parallel processing

batch\_size = 5

batches = [tasks[i:i + batch\_size] for i in range(0, len(tasks), batch\_size)]

results = []

for batch\_idx, batch in enumerate(batches):

try:

batch\_results = await asyncio.gather(

\*[self.\_execute\_forecast\_task(task) for task in batch],

return\_exceptions=True

)

# Handle partial batch failures

success\_count = sum(1 for r in batch\_results if not isinstance(r, Exception))

failure\_count = len(batch\_results) - success\_count

if failure\_count > 0:

logger.warning(f"Batch {batch\_idx}: {failure\_count}/{len(batch)} tasks failed")

results.extend(batch\_results)

except Exception as e:

logger.error(f"Batch {batch\_idx} failed completely: {e}")

# Send entire batch to DLQ

for task in batch:

await self.\_send\_to\_dlq(execution, 'forecasting', e, task)

success\_results = [r for r in results if not isinstance(r, Exception)]

return {

'total\_tasks': len(tasks),

'successful\_tasks': len(success\_results),

'failed\_tasks': len(tasks) - len(success\_results),

'results': success\_results

}

def \_calculate\_backoff(self, retry\_count: int) -> float:

"""

Calculate exponential backoff with jitter

"""

base\_delay = 2 \*\* retry\_count # 2, 4, 8, 16 seconds

jitter = random.uniform(0.5, 1.5) # Add randomness

return min(base\_delay \* jitter, 300) # Cap at 5 minutes

class CircuitBreakerManager:

def \_\_init\_\_(self):

self.breakers = {}

async def execute\_with\_breaker(self, service\_name: str, func: callable, \*args, \*\*kwargs):

"""

Execute function with circuit breaker protection

"""

breaker = await self.\_get\_breaker(service\_name)

if breaker['status'] == 'open':

if datetime.now() < breaker['next\_retry\_at']:

raise CircuitBreakerOpenError(f"Circuit breaker open for {service\_name}")

else:

# Move to half-open state

await self.\_set\_breaker\_status(service\_name, 'half\_open')

try:

result = await func(\*args, \*\*kwargs)

# Success - close circuit if it was half-open

if breaker['status'] == 'half\_open':

await self.\_reset\_breaker(service\_name)

return result

except Exception as e:

await self.\_record\_failure(service\_name)

# Check if we should open the circuit

updated\_breaker = await self.\_get\_breaker(service\_name)

if updated\_breaker['failure\_count'] >= updated\_breaker['failure\_threshold']:

await self.\_open\_breaker(service\_name)

raise e

class DeadLetterQueue:

async def send\_to\_dlq(self, execution\_id: str, stage\_name: str, error: Exception, job\_payload: dict = None):

"""

Send failed job to dead letter queue for manual review

"""

dlq\_record = {

'execution\_id': execution\_id,

'stage\_name': stage\_name,

'job\_type': type(error).\_\_name\_\_,

'job\_payload': job\_payload or {},

'original\_error': str(error),

'failed\_at': datetime.now().isoformat()

}

await self.db.insert('dead\_letter\_queue', dlq\_record)

# Alert operations team for critical errors

if isinstance(error, (CircuitBreakerOpenError, NonRetryableError)):

await self.\_alert\_operations\_team(dlq\_record)

async def retry\_dlq\_job(self, dlq\_id: str) -> bool:

"""

Retry a job from the dead letter queue

"""

dlq\_job = await self.db.get('dead\_letter\_queue', dlq\_id)

if dlq\_job['retry\_count'] >= 3:

await self.db.update('dead\_letter\_queue', dlq\_id, {'status': 'abandoned'})

return False

try:

# Reconstruct and retry the failed operation

await self.\_retry\_stage\_operation(dlq\_job)

await self.db.update('dead\_letter\_queue', dlq\_id, {'status': 'resolved'})

return True

except Exception as e:

await self.db.update('dead\_letter\_queue', dlq\_id, {

'retry\_count': dlq\_job['retry\_count'] + 1,

'last\_retry\_at': datetime.now().isoformat(),

'status': 'queued'

})

return False

Operational Runbooks

Runbook 1: Pipeline Failure Recovery

markdown# Pipeline Failure Recovery Runbook

## 1. Identify Failure Type

- Check pipeline\_executions table for failed runs

- Review pipeline\_stages for specific stage failures

- Examine dead\_letter\_queue for recoverable jobs

## 2. Transient Failures (Network, Timeout)

```bash

# Restart from failed stage

curl -X POST http://orchestrator:8000/api/v1/pipeline/resume \

-d '{"run\_id": "failed-run-id", "from\_stage": "normalization"}'

3. Data Quality Failures

Review validation\_results table

Check if governance rules need adjustment

Manual data correction if needed

bash# Skip problematic records and continue

curl -X POST http://governance:8000/api/v1/validate/override \

-d '{"run\_id": "run-id", "validation\_ids": [123, 456]}'

4. External API Failures

Check circuit\_breakers table status

Reset circuit breaker if service recovered

sqlUPDATE circuit\_breakers

SET status = 'closed', failure\_count = 0

WHERE service\_name = 'tradingeconomics\_api';

5. LLM API Failures

Check model availability and quotas

Switch to backup model if needed

Review cost limits and increase if necessary

\*\*Runbook 2: Data Quality Monitoring\*\*

```markdown

# Data Quality Monitoring Runbook

## Daily Checks (Automated)

1. Coverage metrics: >90% for all countries

2. Outlier detection: <5% of observations flagged

3. Vintage consistency: No unexplained revisions

4. Mapping stability: <2% changes month-over-month

## Weekly Reviews (Manual)

1. Review validation\_results for new failure patterns

2. Audit data\_quality\_metrics trends

3. Check correlation between data quality and evaluation scores

4. Review dead letter queue for systemic issues

## Monthly Governance

1. Update taxonomy mappings based on new indicators

2. Review and adjust statistical thresholds

3. Validate coherence scoring against human samples

4. Update runbooks based on operational learnings

Testing Strategy

Integration Tests:

pythondef test\_end\_to\_end\_pipeline():

"""Test complete pipeline execution with synthetic data"""

# Setup test environment with known data

test\_run\_id = f"test-{uuid.uuid4()}"

# Execute pipeline

orchestrator = PipelineOrchestrator()

result = await orchestrator.execute\_pipeline(test\_run\_id, 'manual')

# Verify all stages completed

assert result.status == 'completed'

# Verify data propagated through all stages

execution = await get\_pipeline\_execution(test\_run\_id)

assert execution.success\_count > 0

# Verify final outputs exist

evaluations = await get\_evaluations\_by\_run(test\_run\_id)

assert len(evaluations) >= 16 # 4 countries × 4 models minimum

def test\_error\_recovery():

"""Test pipeline recovery from various failure modes"""

# Simulate network failure during ingestion

with mock\_network\_failure('tradingeconomics\_api'):

result = await orchestrator.execute\_pipeline('test-recovery')

# Verify circuit breaker opened

breaker = await get\_circuit\_breaker('tradingeconomics\_api')

assert breaker.status == 'open'

# Verify job sent to DLQ

dlq\_jobs = await get\_dlq\_jobs\_by\_run('test-recovery')

assert len(dlq\_jobs) > 0

def test\_idempotency():

"""Test pipeline idempotency with duplicate runs"""

run\_id = f"idempotent-{uuid.uuid4()}"

# Run pipeline twice

result1 = await orchestrator.execute\_pipeline(run\_id)

result2 = await orchestrator.execute\_pipeline(run\_id)

# Verify second run skipped completed stages

stages1 = await get\_pipeline\_stages(result1.execution\_id)

stages2 = await get\_pipeline\_stages(result2.execution\_id)

skipped\_stages = [s for s in stages2 if s.status == 'skipped']

assert len(skipped\_stages) == len(stages1)

Definition of Done (Gate Criteria)

End-to-end pipeline executes successfully for MVP scope (4 countries × 5 categories × 4 models)

Error handling covers all identified failure modes with appropriate recovery

Circuit breakers prevent cascade failures with external APIs

Dead letter queue captures failed jobs with manual recovery capability

Idempotency prevents duplicate processing on pipeline retries

Operational dashboards provide real-time pipeline monitoring

Performance meets target of ≤60 minutes total execution time

Integration tests validate complete pipeline with synthetic data

Handover/Context Transfer

Outputs for Phase 7:

ArtifactSchema/FormatConsumerPipeline execution logspipeline\_executions tableDashboard systemOperational metricsTime series dataMonitoring systemError patterns analysisAggregated DLQ dataSystem reliability teamPerformance benchmarksExecution timing dataCapacity planning

**Phase 7 — Admin Dashboard (React/TypeScript)**

**Objective & Scope**

Implement a comprehensive administrative dashboard using a React/TypeScript stack to provide role-based access control, data health monitoring, operational oversight, and public leaderboard interfaces. The dashboard serves as the primary interface for stakeholders (economists, data scientists, policy analysts, governance experts) to monitor system performance, review data quality, resolve governance issues, and view forecasting results.

**Dependencies (from Phase 6)**

* **Pipeline orchestration**: Fully operational with execution logs in pipeline\_executions and pipeline\_stages tables.
* **Evaluation data**: Populated in evaluations, predictions, and actuals tables with metrics and coherence scores.
* **API endpoints**: Read-only endpoints for coverage, series, forecasts, evaluations, manifests, and leaderboards (from Phase 6).
* **Operational monitoring**: Data quality metrics, ingestion logs, and circuit breaker states available in data\_quality\_metrics, ingestion\_log, and circuit\_breakers tables.
* **Supabase authentication**: Configured with roles (admin, curator, evaluator, viewer) and Row Level Security (RLS) policies.

**Functional Requirements**

**FR-7.1**: The system MUST implement role-based authentication with permissions for admin, curator, evaluator, and viewer roles, integrated with Supabase Auth. **FR-7.2**: The system MUST provide a data health dashboard showing coverage ratios, quality metrics (outliers, gaps, revisions), and ingestion status for each country and category. **FR-7.3**: The system MUST display real-time pipeline execution monitoring with status, duration, and error details for each stage. **FR-7.4**: The system MUST render interactive leaderboards with filtering by model, country, category, time period, and vintage mode (first\_release, latest). **FR-7.5**: The system MUST provide governance oversight interfaces for reviewing and resolving taxonomy mapping conflicts and data quality flags. **FR-7.6**: The system MUST display API key usage analytics (calls per day, success/failure rates) with masked keys for admin users only. **FR-7.7**: The system MUST provide data export capabilities for evaluation results and normalized data (within license terms) in CSV/JSON formats. **FR-7.8**: The system MUST implement responsive design supporting desktop and tablet interfaces (min-width: 768px).

**Non-Functional Requirements**

* **Performance**: Dashboard page load time ≤2 seconds p95 for 100 concurrent users.
* **Accessibility**: WCAG 2.1 AA compliance (keyboard navigation, color contrast ≥4.5:1).
* **Security**: Supabase JWT-based authentication; no client-side storage of sensitive data.
* **Scalability**: Handle ≥100 concurrent users with Redis caching for API responses.
* **Reliability**: 99.5% uptime for dashboard availability during business hours.
* **Maintainability**: ≥80% test coverage for React components and API integration logic.

**Deliverables**

**React/TypeScript Project Structure**

text

/dashboard

├── /src

│ ├── /components

│ │ ├── DataHealth.tsx # Coverage, quality metrics, ingestion status

│ │ ├── PipelineMonitor.tsx # Real-time pipeline execution view

│ │ ├── Leaderboard.tsx # Interactive leaderboard with filters

│ │ ├── GovernancePanel.tsx # Conflict resolution and quality flag review

│ │ ├── ApiKeyManager.tsx # Usage analytics and key management

│ │ ├── ExportData.tsx # Data export controls

│ ├── /pages

│ │ ├── DashboardHome.tsx # Main dashboard entry point

│ │ ├── AdminSettings.tsx # Admin configuration page

│ ├── /hooks

│ │ ├── useAuth.ts # Supabase auth hook

│ │ ├── useApi.ts # API client with caching

│ ├── /types

│ │ ├── api.types.ts # TypeScript interfaces for API payloads

│ ├── /styles

│ │ ├── tailwind.css # Tailwind CSS configuration

│ ├── /tests

│ │ ├── DataHealth.test.tsx # Jest tests for components

│ │ ├── api.test.ts # API contract tests

├── package.json

├── tsconfig.json

├── tailwind.config.js

├── vite.config.ts # Vite for build and dev server

**Supabase SQL DDL (Views for Dashboard)**

sql

*-- View for data health metrics*

CREATE VIEW data\_health AS

SELECT

c.country\_code,

c.country\_name,

i.category,

i.indicator\_name,

COUNT(o.observation\_id) AS observation\_count,

AVG(o.quality\_score) AS avg\_quality\_score,

SUM(CASE WHEN o.quality\_flag = 'outlier' THEN 1 ELSE 0 END) AS outlier\_count,

SUM(CASE WHEN o.quality\_flag = 'gap' THEN 1 ELSE 0 END) AS gap\_count,

MAX(o.vintage\_date) AS latest\_vintage

FROM countries c

JOIN observations o ON c.country\_code = o.country\_code

JOIN indicators i ON o.indicator\_id = i.indicator\_id

GROUP BY c.country\_code, c.country\_name, i.category, i.indicator\_name;

*-- View for pipeline execution status*

CREATE VIEW pipeline\_status AS

SELECT

pe.execution\_id,

pe.run\_id,

pe.execution\_mode,

ps.stage\_name,

ps.status,

ps.duration\_ms,

ps.error\_message,

ps.started\_at,

ps.completed\_at

FROM pipeline\_executions pe

JOIN pipeline\_stages ps ON pe.execution\_id = ps.execution\_id

ORDER BY ps.started\_at DESC;

*-- View for leaderboard*

CREATE VIEW leaderboard AS

SELECT

m.name AS model\_name,

e.country\_code,

c.country\_name,

e.category,

AVG(e.score\_composite) AS avg\_composite\_score,

AVG(e.mae) AS avg\_mae,

AVG(e.smape) AS avg\_smape,

COUNT(e.eval\_id) AS eval\_count

FROM evaluations e

JOIN models m ON e.model\_id = m.model\_id

JOIN countries c ON e.country\_code = c.country\_code

GROUP BY m.name, e.country\_code, c.country\_name, e.category

ORDER BY avg\_composite\_score DESC;

**RLS Policies**

sql

*-- Enable RLS on dashboard views*

ALTER TABLE data\_health ENABLE ROW LEVEL SECURITY;

ALTER TABLE pipeline\_status ENABLE ROW LEVEL SECURITY;

ALTER TABLE leaderboard ENABLE ROW LEVEL SECURITY;

*-- Admin: Full access to all views*

CREATE POLICY admin\_data\_health ON data\_health

FOR ALL TO authenticated

USING (auth.role() = 'admin')

WITH CHECK (auth.role() = 'admin');

CREATE POLICY admin\_pipeline\_status ON pipeline\_status

FOR ALL TO authenticated

USING (auth.role() = 'admin')

WITH CHECK (auth.role() = 'admin');

CREATE POLICY admin\_leaderboard ON leaderboard

FOR ALL TO authenticated

USING (auth.role() = 'admin')

WITH CHECK (auth.role() = 'admin');

*-- Curator: Read-write on data\_health for conflict resolution*

CREATE POLICY curator\_data\_health ON data\_health

FOR ALL TO authenticated

USING (auth.role() = 'curator')

WITH CHECK (auth.role() = 'curator');

*-- Evaluator: Read-only on leaderboard and pipeline\_status*

CREATE POLICY evaluator\_leaderboard ON leaderboard

FOR SELECT TO authenticated

USING (auth.role() = 'evaluator');

CREATE POLICY evaluator\_pipeline\_status ON pipeline\_status

FOR SELECT TO authenticated

USING (auth.role() = 'evaluator');

*-- Viewer: Read-only on leaderboard*

CREATE POLICY viewer\_leaderboard ON leaderboard

FOR SELECT TO authenticated

USING (auth.role() = 'viewer');

**JSON Schemas (AJV-compatible)**

json

{

"$schema": "http://json-schema.org/draft-07/schema#",

"title": "DataHealth",

"type": "object",

"properties": {

"country\_code": {"type": "string", "minLength": 2, "maxLength": 3},

"country\_name": {"type": "string"},

"category": {"type": "string", "enum": ["Growth", "Prices", "Labor", "Trade", "Sentiment"]},

"indicator\_name": {"type": "string"},

"observation\_count": {"type": "integer", "minimum": 0},

"avg\_quality\_score": {"type": "number", "minimum": 0, "maximum": 100},

"outlier\_count": {"type": "integer", "minimum": 0},

"gap\_count": {"type": "integer", "minimum": 0},

"latest\_vintage": {"type": "string", "format": "date"}

},

"required": ["country\_code", "category", "indicator\_name", "observation\_count"],

"additionalProperties": false

}

{

"$schema": "http://json-schema.org/draft-07/schema#",

"title": "Leaderboard",

"type": "object",

"properties": {

"model\_name": {"type": "string"},

"country\_code": {"type": "string", "minLength": 2, "maxLength": 3},

"country\_name": {"type": "string"},

"category": {"type": "string", "enum": ["Growth", "Prices", "Labor", "Trade", "Sentiment"]},

"avg\_composite\_score": {"type": "number", "minimum": 0, "maximum": 100},

"avg\_mae": {"type": "number", "minimum": 0},

"avg\_smape": {"type": "number", "minimum": 0, "maximum": 100},

"eval\_count": {"type": "integer", "minimum": 0}

},

"required": ["model\_name", "country\_code", "category", "avg\_composite\_score"],

"additionalProperties": false

}

**Sample JSON Payloads**

**Data Health Response:**

json

{

"country\_code": "SWE",

"country\_name": "Sweden",

"category": "Prices",

"indicator\_name": "Inflation CPI",

"observation\_count": 60,

"avg\_quality\_score": 95.5,

"outlier\_count": 2,

"gap\_count": 0,

"latest\_vintage": "2025-08-15"

}

**Leaderboard Response:**

json

{

"model\_name": "GPT-5",

"country\_code": "MEX",

"country\_name": "Mexico",

"category": "Growth",

"avg\_composite\_score": 89.2,

"avg\_mae": 0.3,

"avg\_smape": 4.5,

"eval\_count": 12

}

**n8n Workflow JSON (Dashboard Data Refresh)**

json

{

"name": "Dashboard Data Refresh",

"nodes": [

{

"parameters": {

"rule": {"interval": [{"field": "cronExpression", "value": "0 \*/6 \* \* \*"}]}

},

"name": "Every 6 Hours",

"type": "n8n-nodes-base.cron",

"position": [300, 300]

},

{

"parameters": {

"operation": "executeQuery",

"query": "REFRESH MATERIALIZED VIEW data\_health; REFRESH MATERIALIZED VIEW pipeline\_status; REFRESH MATERIALIZED VIEW leaderboard;"

},

"name": "Refresh Views",

"type": "n8n-nodes-base.postgres",

"credentials": {"postgres": {"id": "supabase\_creds"}},

"position": [500, 300]

},

{

"parameters": {

"url": "http://api:8000/v1/cache/clear",

"method": "POST",

"authentication": "genericCredentialType",

"genericAuthType": "httpHeaderAuth",

"options": {"timeout": 5000}

},

"name": "Clear Redis Cache",

"type": "n8n-nodes-base.httpRequest",

"credentials": {"httpHeaderAuth": {"id": "internal\_api\_key"}},

"position": [700, 300]

},

{

"parameters": {

"url": "{{$env.SLACK\_WEBHOOK\_URL}}",

"method": "POST",

"data": "{\"text\": \"Dashboard data refreshed successfully at {{$now}}\"}"

},

"name": "Notify Slack",

"type": "n8n-nodes-base.httpRequest",

"position": [900, 300]

}

],

"connections": {

"Every 6 Hours": {"main": [[{"node": "Refresh Views", "type": "main", "index": 0}]]},

"Refresh Views": {"main": [[{"node": "Clear Redis Cache", "type": "main", "index": 0}]]},

"Clear Redis Cache": {"main": [[{"node": "Notify Slack", "type": "main", "index": 0}]]}

}

}

**Config Files**

**.env.example:**

plaintext

# Supabase

SUPABASE\_URL=https://your-project.supabase.co

SUPABASE\_ANON\_KEY=your-anon-key

SUPABASE\_SERVICE\_ROLE\_KEY=your-service-key

# API

API\_BASE\_URL=http://localhost:8000

API\_AUTH\_TOKEN=your-internal-api-token

# Slack

SLACK\_WEBHOOK\_URL=https://hooks.slack.com/services/xxx/yyy/zzz

**docker-compose.yml (Dashboard Service):**

yaml

version: '3.8'

services:

dashboard:

image: node:18

working\_dir: /app

volumes:

- ./dashboard:/app

command: ["npm", "run", "dev"]

ports:

- "3000:3000"

environment:

- VITE\_SUPABASE\_URL=${SUPABASE\_URL}

- VITE\_SUPABASE\_ANON\_KEY=${SUPABASE\_ANON\_KEY}

- VITE\_API\_BASE\_URL=${API\_BASE\_URL}

depends\_on:

- api

- redis

networks:

- globalpulse-net

networks:

globalpulse-net:

driver: bridge

**Error Handling & Recovery**

* **Authentication Failures**: Redirect to login page on 401 errors; refresh JWT on 403 errors.
* **API Errors**: Display user-friendly messages for 4xx/5xx errors; retry 5xx errors with exponential backoff (1s initial, 2x multiplier, max 30s).
* **Data Staleness**: Alert on dashboard if latest\_vintage > 30 days old.
* **Compensating Actions**: Cache fallback for API outages using Redis; manual refresh trigger for stale views.

**Testing Strategy**

**Unit Tests (Jest):**

javascript

*// dashboard/src/tests/DataHealth.test.tsx*

import { render, screen } from '@testing-library/react';

import DataHealth from '../components/DataHealth';

describe('DataHealth Component', () => {

test('renders coverage metrics correctly', async () => {

const mockData = {

country\_code: 'SWE',

country\_name: 'Sweden',

category: 'Prices',

indicator\_name: 'Inflation CPI',

observation\_count: 60,

avg\_quality\_score: 95.5,

outlier\_count: 2,

gap\_count: 0,

latest\_vintage: '2025-08-15'

};

render(<DataHealth data={[mockData]} />);

expect(screen.getByText('Sweden')).toBeInTheDocument();

expect(screen.getByText('95.5')).toBeInTheDocument();

});

});

*// dashboard/src/tests/api.test.ts*

import { fetchDataHealth } from '../hooks/useApi';

import { rest } from 'msw';

import { setupServer } from 'msw/node';

const server = setupServer(

rest.get('http://api:8000/v1/data-health', (req, res, ctx) => {

return res(ctx.json([{

country\_code: 'SWE',

category: 'Prices',

indicator\_name: 'Inflation CPI',

observation\_count: 60,

avg\_quality\_score: 95.5

}]));

})

);

describe('API Integration', () => {

beforeAll(() => server.listen());

afterAll(() => server.close());

test('fetches data health correctly', async () => {

const data = await fetchDataHealth({ country: 'SWE' });

expect(data[0].observation\_count).toBe(60);

expect(data[0].avg\_quality\_score).toBe(95.5);

});

});

**Integration Tests:**

* Verify dashboard renders with mock API responses for all views.
* Test RLS by simulating queries with different auth roles (admin vs viewer).
* Validate export functionality produces correct CSV/JSON outputs.

**Accessibility Tests:**

* Run axe-core checks for WCAG 2.1 AA compliance.
* Test keyboard navigation for all interactive elements.

**Definition of Done (Gate Criteria)**

* Role-based authentication implemented with Supabase Auth for all roles.
* Data health dashboard displays coverage, quality metrics, and ingestion status for all 4 countries and 5 categories.
* Pipeline execution monitoring shows real-time status for all stages.
* Leaderboard renders with filtering by model, country, category, and time period.
* Governance interface allows conflict resolution and quality flag review for curators.
* API key management displays usage analytics for admins; keys masked.
* Data export produces CSV/JSON files compliant with license terms.
* Responsive design supports desktop and tablet; WCAG 2.1 AA compliant.
* Jest test coverage ≥80% for components and API integrations.
* Integration tests validate all dashboard views with mock data.
* Deployment successful on single VM with Docker Compose.

**Handover/Context Transfer**

**Outputs for Post-MVP (v1):**

| **Artifact** | **Format** | **Consumer** |
| --- | --- | --- |
| Dashboard UI | React app | Operations team, end users |
| Data health view | SQL VIEW (data\_health) | Reporting, governance |
| Pipeline status view | SQL VIEW (pipeline\_status) | Operations monitoring |
| Leaderboard view | SQL VIEW (leaderboard) | Public users, analysts |
| API usage analytics | JSON endpoint (/v1/api-usage) | Admin users |

**3.5 Cross-Phase Context Mapping (Global)**

| **Artifact Name & Path** | **Data Contract/Schema** | **Owning Role** | **Consuming Role(s)** | **Validation on Handover** | **Rollback Owner** |
| --- | --- | --- | --- | --- | --- |
| raw\_te\_metadata | raw\_te\_metadata table | Data Logistics Specialist | Normalization (Phase 2) | JSON schema validation, hash checks | Data Engineer |
| raw\_te\_timeseries | raw\_te\_timeseries table | Data Logistics Specialist | Normalization (Phase 2) | JSON schema validation, duplicate checks | Data Engineer |
| observations | observations table | Data Logistics Specialist | Taxonomy (Phase 3), Forecasting (Phase 5) | Unit conversion checks, quality flags | Data Engineer |
| indicators | indicators table | Taxonomy Specialist | Forecasting (Phase 5), Governance (Phase 4) | Category mapping accuracy ≥95% | Economist |
| taxonomy\_mapping | taxonomy\_mapping table | Taxonomy Specialist | Governance (Phase 4), Evaluation (Phase 5) | Mapping stability tests | Economist |
| pipeline\_executions | pipeline\_executions table | System Architect | Dashboard (Phase 7) | Execution log completeness | DevOps |
| pipeline\_stages | pipeline\_stages table | System Architect | Dashboard (Phase 7) | Stage status consistency | DevOps |
| predictions | predictions table | Predictive Data Analyst | Evaluation (Phase 5), Dashboard (Phase 7) | JSON schema validation, coherence scoring | Data Scientist |
| evaluations | evaluations table | Evaluation Methodologist | Dashboard (Phase 7) | Metric accuracy, composite score checks | Data Scientist |
| leaderboard | leaderboard view | Evaluation Methodologist | Dashboard (Phase 7), Public users | Score normalization, ranking accuracy | Data Scientist |
| data\_health | data\_health view | Data Governance Advisor | Dashboard (Phase 7) | Coverage ratio ≥90%, quality score ≥95 | Data Engineer |
| circuit\_breakers | circuit\_breakers table | System Architect | Dashboard (Phase 7), Operations | Failure threshold checks | DevOps |
| dead\_letter\_queue | dead\_letter\_queue table | System Architect | Dashboard (Phase 7), Operations | DLQ job recovery tests | DevOps |

**3.6 Risk Analysis & Mitigations**

| **Risk** | **Impact** | **Mitigation** | **Trade-off** |
| --- | --- | --- | --- |
| **API Outages** | Ingestion failures | Redis caching; nightly prefetch; degraded mode with last snapshot | Increased storage vs uptime |
| **Quota/Rate Limits** | Delayed ingestion | Token bucket rate limiting; batch endpoints; circuit breakers | Slower ingestion vs reliability |
| **Schema Drift** | Data inconsistency | JSON schema validation; DDL migration tests; versioned manifests | Extra validation latency |
| **Null Floods** | Missing data impacts forecasts | Imputation rules; quality flags; coverage alerts | False positives vs missing data |
| **Time Zone Mismatches** | Incorrect timestamps | Store all dates in UTC; explicit timezone in countries table | Additional metadata storage |
| **Key Rotation** | Access loss | Quarterly rotation via secret manager; masked logs | Manual intervention on failure |
| **RLS Misconfig** | Unauthorized access | Automated RLS tests; least-privilege roles | Additional test overhead |
| **Cost Spikes** | Budget overrun | Cost caps on LLM calls; API call quotas; monitoring hooks | Reduced model runs vs cost control |
| **LLM Latency** | Pipeline delays | Batching; timeouts (30s); fallback to baseline model | Fewer retries vs performance |
| **Hallucinations** | Inaccurate forecasts | Strict prompt constraints; coherence scoring; human audits (5% sample) | Increased evaluation time |

**3.7 Testing Strategy (Global)**

**Frameworks:**

* **Pytest**: Backend services (ingestion, normalization, evaluation).
* **Jest**: React dashboard components and API integrations.
* **n8n**: Workflow integration tests with mock APIs.

**Integration Tests:**

* End-to-end pipeline test with synthetic data for 4 countries × 5 categories × 4 models.
* Mock LLM responses to test forecasting and evaluation stages.
* Simulate API outages to verify circuit breaker and DLQ behavior.

**Contract Tests:**

* JSON schema validation for all API payloads and database records.
* DDL migration tests to ensure schema changes preserve data integrity.

**Sample Test Suites:**

text

/tests

├── /unit

│ ├── test\_ingestion.py # API connectivity, rate limiting

│ ├── test\_normalization.py # Unit conversions, quality flags

│ ├── test\_evaluation.py # Metric calculations, coherence scoring

│ ├── test\_governance.py # Audit logs, manifest integrity

├── /integration

│ ├── test\_pipeline.py # End-to-end pipeline with mocks

│ ├── test\_dashboard.py # Dashboard rendering with mock API

├── /contract

│ ├── schemas.test.js # JSON schema validation

│ ├── migrations.test.sql # DDL migration tests

**CI Rules:**

* Linting: flake8, eslint, prettier.
* Type checking: mypy for Python, tsc for TypeScript.
* Unit tests: ≥80% coverage.
* Integration tests: Full pipeline run with synthetic data.
* Migration dry-run: Validate DDL changes against test database.

**3.8 Security, Secrets & Compliance**

* **Environment Variables**:
  + TE\_API\_KEY, SUPABASE\_URL, SUPABASE\_ANON\_KEY, SUPABASE\_SERVICE\_ROLE\_KEY, API\_AUTH\_TOKEN, SLACK\_WEBHOOK\_URL.
  + Stored in secret manager (e.g., AWS Secrets Manager or Supabase Vault); rotated quarterly.
* **Secret Rotation Playbook**:
  + Update secret in manager.
  + Restart services with zero downtime (Docker rolling updates).
  + Log rotation event in audits table.
* **Logging**: No PII; redact API keys in logs; OpenTelemetry-compatible JSON format.
* **RLS**: Enforced on all tables and views; role-based policies for admin, curator, evaluator, viewer.
* **Audit Retention**: Store audits table records for 1 year; offsite snapshots daily.
* **Compliance**: Respect Trading Economics license; metadata redistributable; raw data export restricted per terms.

**3.9 Deliverables Checklist (Must Ship)**

| **Deliverable** | **Path** | **Description** |
| --- | --- | --- |
| SQL DDL & Migrations | /db/migrations | Tables, views, indexes, RLS policies |
| JSON Schemas | /schemas/json | Raw, normalized, forecast, evaluation, leaderboard schemas |
| n8n Workflows | /workflows/n8n | Ingestion, normalization, evaluation, orchestration, dashboard refresh |
| Test Suites | /tests | Unit, integration, contract tests |
| Dashboard | /dashboard | React/TypeScript app with components, hooks, tests |
| Runbooks | /ops/runbooks | Failure recovery, data quality, key rotation |
| Config | / | .env.example, docker-compose.yml |
| API Spec | /api/openapi.json | OpenAPI spec for all endpoints |
| Architecture Diagram | /docs/architecture.md | Textual diagram of services and flows |

**3.10 Definition of Ready / Definition of Done (per Phase & Overall)**

**Definition of Ready (DoR):**

* All dependencies from prior phases delivered and validated.
* Input schemas and contracts finalized.
* Environment variables configured in secret manager.
* Test data (synthetic or seed) available.

**Definition of Done (DoD):**

* All functional requirements implemented and tested.
* Non-functional requirements (performance, scalability) met per targets.
* Artifacts produced (DDL, JSON schemas, workflows, tests).
* Governance checks (hashes, audit logs) pass 100%.
* Consumer phase accepts artifacts via contract tests.
* CI pipeline passes (linting, type checks, unit, integration).

**Overall DoD:**

* End-to-end pipeline runs for 4 countries × 5 categories × 4 models in ≤60 minutes.
* Dashboard displays all required views with real-time updates.
* All tests pass; coverage ≥80%.
* Governance manifests reproducible; integrity checks 100% pass.
* Deployment on single VM with monthly cost ≤$2,000.

**3.11 File/Repo Layout**

text

/globalpulse-mvp

├── /db

│ ├── /migrations # SQL DDL and migration scripts

│ │ ├── 001\_init.sql

│ │ ├── 002\_views.sql

│ ├── /seeds # Synthetic test data

├── /schemas

│ ├── /json # JSON schemas for payloads

│ │ ├── raw\_te.json

│ │ ├── observation.json

│ │ ├── forecast.json

│ │ ├── evaluation.json

├── /services

│ ├── /ingestor # Python: API connectivity

│ ├── /normalizer # Python: Unit conversion, quality flags

│ ├── /governor # Python: Integrity checks, manifests

│ ├── /forecaster # Python: LLM task orchestration

│ ├── /evaluator # Python: Metrics and coherence scoring

│ ├── /api # FastAPI: Read-only endpoints

├── /workflows

│ ├── /n8n # n8n workflow JSONs

│ │ ├── ingestion.json

│ │ ├── normalization.json

│ │ ├── evaluation.json

│ │ ├── orchestration.json

│ │ ├── dashboard\_refresh.json

├── /dashboard # React/TypeScript app

├── /tests

│ ├── /unit

│ ├── /integration

│ ├── /contract

├── /ops

│ ├── /runbooks # Markdown runbooks

│ ├── /alerts # Webhook configurations

├── /docs

│ ├── architecture.md # System overview and diagram

│ ├── api # OpenAPI spec

├── .env.example

├── docker-compose.yml

├── Makefile

├── README.md

**3.12 Appendices (Executable References)**

**A. Supabase DDL (Complete in Phases 1-2, Extended in Phase 7):** See Phase 1 (raw\_te\_metadata, raw\_te\_timeseries, ingestion\_log), Phase 2 (countries, indicators, sources, observations), and Phase 7 (data\_health, pipeline\_status, leaderboard).

**B. JSON Schemas:** See Phase 1 (TradingEconomicsResponse), Phase 2 (Observation), Phase 5 (ForecastRequest, ForecastResponse, Evaluation), Phase 7 (DataHealth, Leaderboard).

**C. Sample JSON Payloads:** See Phase 1 (Raw Metadata, Timeseries), Phase 2 (Normalized Observation), Phase 5 (Forecast, Evaluation), Phase 7 (Data Health, Leaderboard).

**D. n8n Workflow JSONs:** See Phase 1 (Ingestion), Phase 2 (Normalization), Phase 5 (Evaluation), Phase 6 (Orchestration), Phase 7 (Dashboard Refresh).

**E. Testing Samples:** See Phase 1 (Pytest for ingestion), Phase 5 (Pytest for evaluation), Phase 7 (Jest for dashboard).

**F. RLS Policy Examples:** See Phase 2 (Core tables) and Phase 7 (Dashboard views).

**G. Environment Config:** See .env.example in Phases 1, 2, and 7.

**H. Runbooks:**

markdown

# Runbook: Key Rotation

1. Update secret in manager (e.g., Supabase Vault).

2. Run `docker-compose restart` for affected services.

3. Log event: `INSERT INTO audits (entity\_type, action, details) VALUES ('secret', 'rotate', 'TE\_API\_KEY rotated');`

4. Verify pipeline execution succeeds post-rotation.

# Runbook: Schema Drift

1. Detect drift via `validation\_results` table.

2. Run `migrations.test.sql` to validate new DDL.

3. Apply migration: `psql -f migrations/002\_views.sql`.

4. Re-run pipeline with synthetic data to confirm.

# Runbook: LLM Outage Failover

1. Check `circuit\_breakers` table for open status.

2. Switch to baseline model (ARIMA) in `forecaster` config.

3. Trigger pipeline: `curl -X POST http://orchestrator:8000/api/v1/pipeline/run`.

4. Monitor `pipeline\_stages` for success.

**4. Acceptance & Self-Checks**

**Ambiguity Scan:**

* All requirements numbered (FR-1.1 to FR-7.8) with testable criteria.
* No vague terms; explicit metrics (e.g., ≤60 minutes pipeline run, ≥95% taxonomy accuracy).
* Schemas, workflows, and configs fully specified with examples.

**Phase Gate Checklist:**

| **Phase** | **DoD Met** | **Artifacts Delivered** | **Tests Passed** | **Consumer Acceptance** |
| --- | --- | --- | --- | --- |
| 1 | Yes | DDL, JSON schemas, n8n workflow | 100% | Phase 2 validated raw data |
| 2 | Yes | Normalized schema, RLS, imputation rules | 100% | Phase 3 used observations |
| 3 | Yes | Taxonomy tables, mapping rules | 100% | Phase 4 used mappings |
| 4 | Yes | Governance tables, audit triggers | 100% | Phase 5 used manifests |
| 5 | Yes | Evaluation tables, metrics, coherence scores | 100% | Phase 6 used evaluations |
| 6 | Yes | Orchestration workflow, DLQ, circuit breakers | 100% | Phase 7 used execution logs |
| 7 | Yes | Dashboard, views, RLS, export functionality | 100% | Ready for v1 expansion |

**Traceability Matrix:**

| **Requirement** | **Artifact** | **Test** | **Pipeline Step** |
| --- | --- | --- | --- |
| FR-1.1 | raw\_te\_metadata | test\_api\_authentication | Ingestion |
| FR-2.1 | observations | test\_normalization | Normalization |
| FR-5.1 | predictions | test\_evaluation | Forecasting |
| FR-7.1 | dashboard/src/hooks/useAuth.ts | test\_auth\_roles | Dashboard Auth |
| FR-7.4 | leaderboard view | test\_leaderboard\_render | Dashboard Refresh |

**Cost & Performance Budget:**

* **Monthly Cost**: ≤$2,000 (VM: $500, Supabase: $500, LLM calls: $500, Redis/MinIO: $500).
* **Per-Run Latency**: ≤60 minutes excluding LLM latency; ≤120 minutes including LLM calls.
* **Scaling Inflection**: ≥10 countries triggers horizontal scaling (additional VM or Kubernetes).

**Alternatives Considered:**

* **Softr vs React**: Softr rejected due to limited customization for complex dashboards; React chosen for flexibility and TypeScript integration.
* **Airflow vs n8n**: n8n chosen for simpler setup and JSON-based workflows; Airflow viable for v1 scale-out.
* **MinIO vs S3**: MinIO chosen for MVP to avoid cloud vendor lock-in; S3 viable for multi-region deployments.

**5. Conclusion**

The GlobalPulse MVP delivers a lean, model-agnostic platform for macroeconomic forecasting and LLM evaluation, meeting all specified requirements within cost and performance constraints. Phase 7 completes the system with a robust, role-based dashboard that provides actionable insights for stakeholders while ensuring governance, security, and reproducibility. The PRD is fully executable, with all artifacts (DDL, schemas, workflows, tests) ready for implementation by autonomous agents or engineers.