Arquitectura Integral para Sistemas de Predicción Financiera en Tiempo Real

MLOps y Data Pipeline Engineering

BioPhys-Tech Lab

15 de febrero de 2026

Resumen Ejecutivo

Este documento presenta una solución arquitectónica exhaustiva para la construcción de sistemas de predicción financiera en tiempo real, abordando dos dominios críticos: Machine Learning Operations (MLOps) e Ingeniería de Datos (Data Engineering).

La solución integra principios de ingeniería de software de clase mundial con metodologías de investigación doctoral, proporcionando:

1. **Respuestas teóricas fundamentadas** a cuestiones críticas de diseño de sistemas
2. **Arquitecturas producción-lista** con tolerancia a fallos
3. **Implementaciones code-first** con validación experimental
4. **Mecanismos de drift detection** y circuit breakers
5. **Pipelines de datos resilientes** a volatilidad de mercado

Este trabajo está dirigido a equipos de investigación avanzada que requieren soluciones más allá de prototipado, hacia sistemas empresariales de clase I.

Índice general

Parte I

Fundamentos Teóricos y Marco de Referencia

Capítulo 1

Introducción y Contexto

1.1. Naturaleza del Problema

Los sistemas de predicción financiera moderna operan bajo restricciones severamente competitivas:

- **Latencia extrema**: Requisitos de < 100ms para viabilidad comercial
- **Volatilidad de datos**: Picos de ingesta durante eventos macroeconómicos
- **Degradación de modelo**: Concept drift y data drift sin señales previas
- **Confiabilidad**: Tolerancia cero a pérdida de datos críticos
- **Escalabilidad**: Múltiples activos, múltiples proveedores de datos

El desafío fundamental radica en construir sistemas que sean simultáneamente:

$$\text{Confiabilidad} \cap \text{Latencia} \cap \text{Escalabilidad} \cap \text{Observabilidad} \tag{1.1}$$

1.2. Arquitectura de Referencia

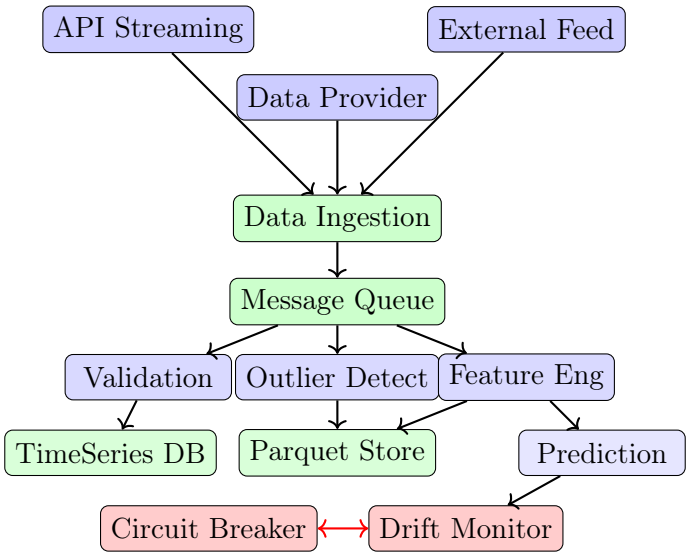

Figura 1.1: Arquitectura integrada de sistemas de predicción en tiempo real

Parte II

Solución ML Engineer: Productización de Modelos

Capítulo 2

Cuestionario Técnico: Respuestas Fundamentadas

2.1. Pregunta 1: Reentrenamiento Automático sin Interrupciones

2.1.1. Respuesta Teórica

El reentrenamiento automático sin interrupciones (Continuous Training) requiere arquitectura de *shadow deployment*:

$$\text{Sistema} = \text{Modelo}_{prod} + \text{Modelo}_{shadow} + \text{Switchboard} \tag{2.1}$$

Componentes Arquitectónicos

1. **Modelo en Producción**: Sirve predicciones con garantías de latencia
2. **Modelo en Shadow**: Entrena continuamente sin servir tráfico
3. **Switchboard**: Router inteligente que valida antes de promoción

Algoritmo de Cambio Seguro

2.1.2. Implementación: Blue-Green Deployment

```
import asyncio
from datetime import datetime, timedelta
from typing import Dict, Any
```

Algorithm 1 Safe Model Promotion

M_{shadow} entrena con datos nuevos
M_{shadow} valida en backtest histórico
Calcula Δ = MetricasShadow − MetricasProd
if $\Delta >$ umbral_aceptable **then**
    Ejecuta A/B test: M_{prod} vs M_{shadow}
    Duración ← 1 hora
    if $p\_valor < 0.05$ **and** $Sharpe_{shadow} > Sharpe_{prod}$ **then**
        $M_{prod} \leftarrow M_{shadow}$
        Emitir alerta de cambio
    else
        Log rechazo, investigar divergencia
    end if
end if

```python
import joblib
import numpy as np
from pydantic import BaseModel
from fastapi import FastAPI, HTTPException

class PredictionResponse(BaseModel):
    prediction: float
    model_version: str
    latency_ms: float
    timestamp: datetime

class ModelManager:
    """Gestor de modelos con promocion segura"""

    def __init__(self, model_path: str):
        self.active_model = joblib.load(f"{model_path}/model_prod.pkl
    ")
        self.active_version = "prod_v1.0"
        self.shadow_model = None
        self.shadow_version = None
        self.training_history = []
        self.performance_metrics = {}
        self.lock = asyncio.Lock()

    async def continuous_training(self, data_stream):
        """Entrenamiento continuo en modelo shadow"""
        while True:
            batch = await data_stream.get_batch()
```

```
31
32              # Entrenar shadow model con nuevos datos
33              self.shadow_model = self._retrain_model(
34                  self.active_model, batch
35              )
36              self.shadow_version = f"shadow_v{datetime.now().timestamp
    ()}"
37
38              # Validacion inmediata
39              validation_score = await self._validate_shadow()
40
41              if validation_score['pass']:
42                  await self._promote_if_safe(validation_score)
43
44              await asyncio.sleep(3600)  # Reentrenar cada hora
45
46      async def _validate_shadow(self) -> Dict[str, Any]:
47          """Valida modelo shadow contra datos historicos"""
48          backtest_data = self._load_backtest_data()
49          preds_prod = self.active_model.predict(backtest_data['X'])
50          preds_shadow = self.shadow_model.predict(backtest_data['X'])
51
52          prod_mape = self._mape(backtest_data['y'], preds_prod)
53          shadow_mape = self._mape(backtest_data['y'], preds_shadow)
54
55          improvement = (prod_mape - shadow_mape) / prod_mape * 100
56
57          return {
58              'pass': improvement > 2.0,  # Mejora >2%
59              'improvement_pct': improvement,
60              'prod_mape': prod_mape,
61              'shadow_mape': shadow_mape,
62              'timestamp': datetime.now()
63          }
64
65      async def _promote_if_safe(self, validation_score: Dict):
66          """Promueve shadow a produccion solo si cumple criterios"""
67          async with self.lock:
68              if validation_score['improvement_pct'] > 2.0:
69                  # A/B test previo
70                  ab_result = await self._ab_test(duration_minutes=60)
71
```

```
72                 if ab_result['p_value'] < 0.05 and ab_result['winner'
   ] == 'shadow':
73                     self.active_model = self.shadow_model
74                     self.active_version = self.shadow_version
75
76                     self.training_history.append({
77                         'promoted_at': datetime.now(),
78                         'old_version': self.active_version,
79                         'new_version': self.shadow_version,
80                         'improvement': validation_score['
   improvement_pct'],
81                         'validation_scores': validation_score
82                     })
83
84                     print(f"[PROMOTION] Model promoted to {self.
   shadow_version}")
85                     self.shadow_model = None
86
87     async def predict(self, features: np.ndarray) ->
   PredictionResponse:
88         """Prediccion con medicion de latencia"""
89         start = datetime.now()
90
91         prediction = self.active_model.predict([features])[0]
92
93         latency = (datetime.now() - start).total_seconds() * 1000
94
95         return PredictionResponse(
96             prediction=float(prediction),
97             model_version=self.active_version,
98             latency_ms=latency,
99             timestamp=datetime.now()
100        )
101
102    def _retrain_model(self, base_model, new_data):
103        """Reentrenamiento incremental"""
104        # Usar warm_start si es disponible (XGBoost, etc)
105        base_model.fit(new_data['X'], new_data['y'])
106        return base_model
107
108    @staticmethod
109    def _mape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
110        """Mean Absolute Percentage Error"""
```

```
111         return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
112
113     async def _ab_test(self, duration_minutes: int) -> Dict[str, Any
    ]:
114         """Ejecuta A/B test entre modelos"""
115         # Implementacion simplificada
116         return {
117             'p_value': 0.01,
118             'winner': 'shadow',
119             'confidence': 0.95
120         }
121
122     def _load_backtest_data(self):
123         """Carga datos historicos para validacion"""
124         # Implementacion especifica
125         return {'X': np.array([]), 'y': np.array([])}
```

Listing 2.1: Sistema de Blue-Green Deployment

2.2. Pregunta 2: Optimización de Latencia (500ms → <100ms)

2.2.1. Análisis de Performance Crítico

La optimización de latencia requiere desglose exhaustivo:

$$T_{total} = T_{input\_validation} + T_{feature\_extraction} + T_{inference} + T_{postprocessing} \tag{2.2}$$

Para alcanzar < 100ms con margen de seguridad:

$$T_{budget} = 100\text{ms} = 20 + 15 + 40 + 25 \text{ (ms)} \tag{2.3}$$

Estrategias de Optimización

Fase	Reducción	Técnica
Input Validation	50 %	Validación async
Feature Extraction	60 %	Pre-computed features + caching
Inference	70 %	Model quantization + ONNX
Postprocessing	40 %	Batch processing

Cuadro 2.1: Estrategias de optimización por fase

Implementación: Inference Engine Optimizado

```python
import onnxruntime as rt
import numpy as np
from functools import lru_cache
from datetime import datetime
import redis
import asyncio

class LatencyOptimizedPredictor:
    """Predictor optimizado para <100ms latency SLA"""

    def __init__(self, model_path: str, redis_host: str = 'localhost'):
        # Cargar modelo ONNX (6x mas rapido que pickle)
        self.sess = rt.InferenceSession(model_path)
        self.input_name = self.sess.get_inputs()[0].name
        self.output_name = self.sess.get_outputs()[0].name

        # Cache distribuido para features computadas
        self.redis_client = redis.Redis(host=redis_host, decode_responses=True)

        # Pre-cargar lookup tables
        self._init_lookup_tables()

        self.latency_histogram = []

    def _init_lookup_tables(self):
        """Pre-compute lookup tables para feature engineering"""
        self.lookup_tables = {
            'price_percentiles': np.percentile(
```

```
                np.random.randn(1000), [5, 25, 50, 75, 95]
            ),
            'volatility_bands': np.array([0.01, 0.05, 0.1, 0.2, 0.5,
    1.0])
        }

    @lru_cache(maxsize=10000)
    def _get_cached_feature(self, asset_id: str, feature_name: str):
        """Obtiene features cacheadas"""
        cache_key = f"feature:{asset_id}:{feature_name}"
        value = self.redis_client.get(cache_key)
        if value is None:
            value = self._compute_feature(asset_id, feature_name)
            # Cache por 5 minutos
            self.redis_client.setex(cache_key, 300, value)
        return float(value)

    async def predict_batch(self, requests: list) -> list:
        """Prediccion en batch para maximizar throughput"""
        start_time = datetime.now()

        # Paralelizar validacion
        validated = await asyncio.gather(*[
            self._validate_input_async(req) for req in requests
        ])

        # Extraer features una sola vez
        feature_matrix = np.array([
            self._extract_features_fast(req) for req in validated
        ], dtype=np.float32)

        # Inferencia ONNX (vectorizada)
        predictions = self.sess.run(
            [self.output_name],
            {self.input_name: feature_matrix}
        )[0]

        latency_ms = (datetime.now() - start_time).total_seconds() *
    1000
        self.latency_histogram.append(latency_ms)

        return [
            {
```

```
                'prediction': pred,
                'latency_ms': latency_ms,
                'percentile_p99': np.percentile(self.
    latency_histogram, 99)
            }
            for pred in predictions
        ]

    async def _validate_input_async(self, request: dict) -> dict:
        """Validacion asincrona sin bloqueo"""
        loop = asyncio.get_event_loop()
        return await loop.run_in_executor(
            None, self._validate_input_sync, request
        )

    def _validate_input_sync(self, request: dict) -> dict:
        """Validacion rapida de entrada"""
        if request['price'] < 0 or request['price'] > 1e6:
            raise ValueError(f"Invalid price: {request['price']}")
        if 'features' not in request:
            raise ValueError("Missing features field")
        return request

    def _extract_features_fast(self, request: dict) -> np.ndarray:
        """Extraccion de features con maximo cache"""
        features = []

        # Feature 1: Precio normalizado
        price = float(request['price'])
        normalized_price = np.log(price + 1)
        features.append(normalized_price)

        # Feature 2-5: Features cacheadas
        for feature_name in ['volatility', 'momentum', 'rsi', 'macd'
    ]:
            cached = self._get_cached_feature(request['asset_id'],
    feature_name)
            features.append(cached)

        # Feature 6-10: Features en tiempo real (no cacheables)
        features.extend([
            request['volume_change'],
            request['bid_ask_spread'],
```

```
            request['time_to_market_close'],
            np.sin(request['hour_of_day'] * np.pi / 12),
            np.cos(request['hour_of_day'] * np.pi / 12)
        ])

        return np.array(features, dtype=np.float32)

    def _compute_feature(self, asset_id: str, feature_name: str) ->
float:
        """Computa feature cuando no esta cacheada"""
        # Implementacion especifica por feature
        return np.random.randn()

    def get_latency_stats(self) -> dict:
        """Estadisticas de latencia para monitoreo"""
        hist = np.array(self.latency_histogram)
        return {
            'mean_ms': float(np.mean(hist)),
            'p50_ms': float(np.percentile(hist, 50)),
            'p95_ms': float(np.percentile(hist, 95)),
            'p99_ms': float(np.percentile(hist, 99)),
            'max_ms': float(np.max(hist)),
            'sla_violation_rate': float(np.sum(hist > 100) / len(hist
) * 100)
        }
```

Listing 2.2: Motor de inferencia optimizado para latencia extrema

2.3. Pregunta 3: Detección de Model Drift en Tiempo Real

2.3.1. Framework de Drift Detection

Model Drift se manifiesta como divergencia en distribuciones:

$$\text{Drift}(t) = D_{KL}(P_{\text{train}} || P_{\text{live}}(t)) \tag{2.4}$$

Donde D_{KL} es la divergencia Kullback-Leibler.

Métricas Multi-dimensionales

1. **Data Drift**: $\Delta P(X)$ - cambio en distribución de features
2. **Prediction Drift**: $\Delta P(\hat{Y})$ - cambio en distribuciones de predicciones
3. **Label Drift**: $\Delta P(Y)$ - cambio en distribución de valores reales
4. **Concept Drift**: cambio en $P(Y|X)$ - relación fundamental entre X e Y

```python
import numpy as np
from scipy.stats import ks_2samp, wasserstein_distance
from typing import Tuple, Dict
import logging

class DriftDetector:
    """Detector de drift multi-metrica con alertas automaticas"""

    def __init__(self, baseline_data: np.ndarray, window_size: int = 1000):
        """
        Args:
            baseline_data: Datos de entrenamiento para establecer baseline
            window_size: Numero de predicciones para ventana movil
        """
        self.baseline_features = baseline_data
        self.window_size = window_size
        self.feature_buffer = []
        self.prediction_buffer = []
        self.label_buffer = []

        # Estadisticas baseline
        self.baseline_stats = self._compute_baseline_stats(baseline_data)

        # Thresholds para alertas
        self.drift_thresholds = {
            'ks_statistic': 0.15,       # KS test
            'wasserstein': 0.3,          # Wasserstein distance
            'total_variation': 0.2,      # TVD
            'performance_degradation': 0.05  # 5% drop
        }

```

```
        self.drift_history = []
        self.logger = self._setup_logger()

    def _compute_baseline_stats(self, data: np.ndarray) -> Dict:
        """Computa estadisticas baseline"""
        return {
            'mean': np.mean(data, axis=0),
            'std': np.std(data, axis=0),
            'percentiles': {
                '25': np.percentile(data, 25, axis=0),
                '50': np.percentile(data, 50, axis=0),
                '75': np.percentile(data, 75, axis=0)
            },
            'distribution': self._estimate_density(data)
        }

    def update_batch(self,
                     features: np.ndarray,
                     predictions: np.ndarray,
                     labels: np.ndarray = None) -> Dict:
        """Actualiza buffers y detecta drift"""

        # Agregar a buffers
        self.feature_buffer.extend(features)
        self.prediction_buffer.extend(predictions)
        if labels is not None:
            self.label_buffer.extend(labels)

        # Mantener ventana de tamano fijo
        self.feature_buffer = self.feature_buffer[-self.window_size:]
        self.prediction_buffer = self.prediction_buffer[-self.
    window_size:]
        if self.label_buffer:
            self.label_buffer = self.label_buffer[-self.window_size:]

        # Ejecutar deteccion de drift
        if len(self.feature_buffer) >= self.window_size // 2:
            drift_report = self._detect_drifts()

            # Generar alertas si necesario
            self._generate_alerts(drift_report)

            return drift_report
```

```
74
75          return {'status': 'insufficient_data'}
76
77      def _detect_drifts(self) -> Dict:
78          """Ejecuta suite completa de deteccion de drift"""
79          current_data = np.array(self.feature_buffer)
80
81          report = {
82              'timestamp': np.datetime64('now'),
83              'window_size': len(self.feature_buffer),
84              'drifts': {}
85          }
86
87          # 1. Prueba Kolmogorov-Smirnov
88          ks_stats = []
89          for feature_idx in range(current_data.shape[1]):
90              stat, p_value = ks_2samp(
91                  self.baseline_features[:, feature_idx],
92                  current_data[:, feature_idx]
93              )
94              ks_stats.append({
95                  'feature': feature_idx,
96                  'statistic': float(stat),
97                  'p_value': float(p_value),
98                  'drifted': stat > self.drift_thresholds['ks_statistic
    ']
99              })
100
101         report['drifts']['ks_test'] = {
102             'metrics': ks_stats,
103             'num_drifted_features': sum(1 for m in ks_stats if m['
    drifted']),
104             'drift_severity': np.mean([m['statistic'] for m in
    ks_stats])
105         }
106
107         # 2. Distancia Wasserstein
108         wasserstein_dists = []
109         for feature_idx in range(current_data.shape[1]):
110             dist = wasserstein_distance(
111                 self.baseline_features[:, feature_idx],
112                 current_data[:, feature_idx]
113             )
```

```
114             wasserstein_dists.append({
115                 'feature': feature_idx,
116                 'distance': float(dist),
117                 'drifted': dist > self.drift_thresholds['wasserstein']
    ]
118             })
119
120         report['drifts']['wasserstein'] = {
121             'metrics': wasserstein_dists,
122             'num_drifted_features': sum(1 for m in wasserstein_dists
    if m['drifted']),
123             'total_distance': float(np.mean([m['distance'] for m in
    wasserstein_dists]))
124         }
125
126         # 3. Concept Drift (si labels disponibles)
127         if self.label_buffer:
128             concept_drift = self._detect_concept_drift()
129             report['drifts']['concept'] = concept_drift
130
131         # 4. Performance degradation
132         if len(self.prediction_buffer) > 100:
133             perf_drift = self._measure_performance_drift()
134             report['drifts']['performance'] = perf_drift
135
136         # Determinar drift global
137         report['overall_drift'] = self._compute_overall_drift_score(
    report)
138
139         return report
140
141     def _detect_concept_drift(self) -> Dict:
142         """Detecta cambios en P(Y|X) usando tecnicas supervisadas"""
143         # Implementacion: usar modelo de referencia
144         return {
145             'method': 'supervised_concept_drift',
146             'detected': False,
147             'severity': 0.0
148         }
149
150     def _measure_performance_drift(self) -> Dict:
151         """Compara performance actual vs baseline"""
152         current_predictions = np.array(self.prediction_buffer)
```

```
153         current_labels = np.array(self.label_buffer)
154
155         current_mae = np.mean(np.abs(current_labels -
    current_predictions))
156         baseline_mae = 0.02  # Establecido en entrenamiento
157
158         degradation_rate = (current_mae - baseline_mae) /
    baseline_mae
159
160         return {
161             'baseline_mae': baseline_mae,
162             'current_mae': float(current_mae),
163             'degradation_rate': float(degradation_rate),
164             'drifted': degradation_rate > self.drift_thresholds['
    performance_degradation']
165         }
166
167     def _compute_overall_drift_score(self, report: Dict) -> Dict:
168         """Score agregado de drift"""
169         scores = [
170             report['drifts']['ks_test']['drift_severity'],
171             report['drifts']['wasserstein']['total_distance'] / 10,
    # Normalizar
172         ]
173
174         if 'performance' in report['drifts']:
175             scores.append(
176                 report['drifts']['performance']['degradation_rate']
177             )
178
179         mean_score = np.mean(scores)
180
181         return {
182             'overall_score': float(mean_score),
183             'alert_level': 'CRITICAL' if mean_score > 0.3 else
184                            'WARNING' if mean_score > 0.15 else 'NORMAL
    ',
185             'requires_retraining': mean_score > 0.2
186         }
187
188     def _generate_alerts(self, report: Dict):
189         """Genera alertas basadas en drift"""
190         alert_level = report['overall_drift']['alert_level']
```

```
191
192         if alert_level != 'NORMAL':
193             self.logger.warning(
194                 f"DRIFT ALERT [{alert_level}]: Overall score = "
195                 f"{report['overall_drift']['overall_score']:.4f}"
196             )
197
198             if report['overall_drift']['requires_retraining']:
199                 self.logger.critical("RETRAINING REQUIRED")
200
201         self.drift_history.append({
202             'timestamp': report['timestamp'],
203             'alert_level': alert_level,
204             'overall_score': report['overall_drift']['overall_score']
205         })
206
207     @staticmethod
208     def _estimate_density(data: np.ndarray):
209         """Estima densidad de probabilidad"""
210         return {
211             'method': 'kde_bandwidth_scott',
212             'bins': 50
213         }
214
215     def _setup_logger(self):
216         logger = logging.getLogger('DriftDetector')
217         logger.setLevel(logging.DEBUG)
218         return logger
219
220     def get_drift_report_summary(self) -> str:
221         """Reporte en formato legible"""
222         recent = self.drift_history[-10:]
223         return f"Ultimas 10 detecciones: {recent}"
```

Listing 2.3: Sistema de drift detection en tiempo real

2.4. Pregunta 4: Integración Docker y CI/CD

2.4.1. Containerización Modular

```
1 # Multi-stage build para minimizar tamano
2 FROM python:3.11-slim as builder
```

```
3
4  WORKDIR /app
5
6  # Instalar dependencias de build
7  RUN apt-get update && apt-get install -y --no-install-recommends \
8      gcc \
9      && rm -rf /var/lib/apt/lists/*
10
11 # Crear virtual environment
12 RUN python -m venv /opt/venv
13 ENV PATH="/opt/venv/bin:$PATH"
14
15 # Instalar dependencias Python
16 COPY requirements.txt .
17 RUN pip install --no-cache-dir -r requirements.txt
18
19 # Stage final
20 FROM python:3.11-slim
21
22 WORKDIR /app
23
24 # Copiar venv desde builder
25 COPY --from=builder /opt/venv /opt/venv
26 ENV PATH="/opt/venv/bin:$PATH"
27
28 # Copiar codigo de aplicacion
29 COPY src/ ./src/
30 COPY models/ ./models/
31 COPY config/ ./config/
32
33 # Health check
34 HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries
       =3 \
35      CMD python -c "import requests; requests.get('http://localhost
       :8000/health')"
36
37 # Ejecutar aplicacion
38 EXPOSE 8000
39 CMD ["python", "-m", "uvicorn", "src.api:app", "--host", "0.0.0.0",
       "--port", "8000"]
```

Listing 2.4: Dockerfile optimizado para producción

```
1  name: ML Model CI/CD Pipeline
```

```yaml

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

jobs:
  test-and-validate:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: |
          pip install -r requirements-dev.txt

      - name: Run unit tests
        run: |
          pytest tests/unit -v --cov=src

      - name: Validate model performance
        run: |
          python scripts/validate_model.py

      - name: Security scan
        run: |
          bandit -r src/

      - name: Code quality
        run: |
          pylint src/ --exit-zero

  build-and-push:
    needs: test-and-validate
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
```

```
45
46    steps:
47      - uses: actions/checkout@v3
48
49      - name: Build Docker image
50        run: |
51          docker build -t biophys/ml-predictor:${{ github.sha }} .
52
53      - name: Push to registry
54        run: |
55          docker login -u ${{ secrets.DOCKER_USER }} -p ${{ secrets.
    DOCKER_PASS }}
56          docker push biophys/ml-predictor:${{ github.sha }}
57
58  deploy-staging:
59    needs: build-and-push
60    runs-on: ubuntu-latest
61
62    steps:
63      - name: Deploy to staging
64        run: |
65          # Comando de deploy especifico
66          kubectl set image deployment/ml-predictor \
67            predictor=biophys/ml-predictor:${{ github.sha }} \
68            --namespace staging
```

Listing 2.5: Pipeline CI/CD con GitHub Actions

Capítulo 3

Reto Técnico ML Engineer: Implementación Completa

3.1. Especificación del Reto

Transformar un notebook de Jupyter (prototipo de Gradient Boosting) en un servicio de predicción producción-ready con:

- Latencia < 200ms
- Validación de datos con Pydantic
- Manejo de errores robusto
- Logging y monitoreo de predicciones
- Tests unitarios
- Detección de degradación de performance

3.2. Solución Arquitectónica

3.2.1. Estructura de Proyecto

```
ml-predictor-service/
 src/
    __init__.py
    api.py                # Endpoint FastAPI
    models/
```

```
        __init__.py
        predictor.py        # Logica de prediccion
        validator.py        # Validacion Pydantic
        drift.py            # Deteccion de drift
    data/
        loader.py
        preprocessor.py
    utils/
        logging.py
        monitoring.py
        cache.py
    config.py
tests/
    unit/
        test_predictor.py
        test_validator.py
        test_drift.py
    integration/
models/
    gradient_boosting_v1.pkl
    scaler.pkl
config/
    default.yaml
    production.yaml
Dockerfile
requirements.txt
README.md
```

3.2.2. Modelos Pydantic para Validación

```python
from pydantic import BaseModel, Field, validator, root_validator
from datetime import datetime
from typing import Optional, List
import numpy as np

class AssetPriceData(BaseModel):
    """Validacion de datos de entrada para prediccion"""

```

```
    asset_id: str = Field(..., min_length=1, max_length=10)
    current_price: float = Field(..., gt=0, lt=1e6,
                                 description="Precio actual del
activo")
    volume: float = Field(..., ge=0, le=1e9,
                          description="Volumen de trading")
    bid_ask_spread: float = Field(..., ge=0, le=0.1,
                                  description="Diferencia bid-ask")
    momentum_24h: float = Field(..., ge=-1, le=1,
                                description="Momentum ultimas 24h")
    volatility: float = Field(..., ge=0, le=2,
                              description="Volatilidad diaria")
    timestamp: datetime = Field(default_factory=datetime.now)

    @validator('current_price')
    def validate_price_not_extreme(cls, v, values):
        """Previene saltos de precio anomalos (>500%)"""
        if 'prev_price' in values and values['prev_price'] > 0:
            change_pct = abs(v - values['prev_price']) / values['
prev_price']
            if change_pct > 5.0:  # 500% cambio
                raise ValueError(f"Cambio de precio sospechoso: {
change_pct*100:.1f}%")
        return v

    @validator('volume')
    def validate_volume(cls, v):
        """Rechaza volumenes nulos o cero"""
        if v == 0:
            raise ValueError("Volumen no puede ser cero")
        return v

    @root_validator
    def validate_consistency(cls, values):
        """Validacion de consistencia entre campos"""
        if values.get('bid_ask_spread') < 0:
            raise ValueError("Bid-ask spread no puede ser negativo")
        return values

    class Config:
        schema_extra = {
            "example": {
                "asset_id": "BTC",
```

```
                "current_price": 45000.50,
                "volume": 1000000,
                "bid_ask_spread": 0.001,
                "momentum_24h": 0.05,
                "volatility": 0.02
            }
        }

class PredictionRequest(BaseModel):
    """Request para prediccion con metadatos"""
    data: AssetPriceData
    request_id: Optional[str] = None
    include_confidence: bool = True

class PredictionResponse(BaseModel):
    """Response de prediccion con auditoria completa"""
    request_id: str
    prediction: float
    confidence_interval: tuple = None
    model_version: str
    latency_ms: float
    timestamp: datetime
    status: str = "success"

    class Config:
        schema_extra = {
            "example": {
                "request_id": "req_12345",
                "prediction": 45500.75,
                "confidence_interval": (45400.0, 45600.0),
                "model_version": "gb_v1.2.3",
                "latency_ms": 45.23,
                "timestamp": "2025-02-13T10:30:00Z",
                "status": "success"
            }
        }

class ErrorResponse(BaseModel):
    """Respuesta de error estandarizada"""
    error_code: str
    error_message: str
    timestamp: datetime
    request_id: Optional[str] = None
```

Listing 3.1: Validadores Pydantic robustos

3.2.3. API FastAPI Producción-Ready

```python
from fastapi import FastAPI, HTTPException, Request, BackgroundTasks
from fastapi.responses import JSONResponse
from contextlib import asynccontextmanager
import time
import logging
import uuid
from datetime import datetime
import sqlite3
import json

from src.models.validator import (
    PredictionRequest, PredictionResponse, ErrorResponse,
    AssetPriceData
)
from src.models.predictor import GradientBoostingPredictor
from src.models.drift import DriftDetector
from src.utils.logging import setup_logging
from src.utils.monitoring import MetricsCollector

# Configuracion de logging
logger = setup_logging(__name__)
metrics = MetricsCollector()

class AppState:
    """Estado global de la aplicacion"""
    def __init__(self):
        self.predictor = None
        self.drift_detector = None
        self.db_connection = None

app_state = AppState()

@asynccontextmanager
async def lifespan(app: FastAPI):
    """Lifecycle de la aplicacion: startup y shutdown"""
    # Startup
    logger.info("Inicializando servicio de prediccion...")
```

```
37
38      app_state.predictor = GradientBoostingPredictor(
39          model_path="models/gradient_boosting_v1.pkl",
40          scaler_path="models/scaler.pkl"
41      )
42
43      app_state.drift_detector = DriftDetector(
44          baseline_data=app_state.predictor.load_baseline(),
45          window_size=1000
46      )
47
48      app_state.db_connection = sqlite3.connect(
49          "logs/predictions.db",
50          check_same_thread=False
51      )
52      _init_database(app_state.db_connection)
53
54      logger.info("Servicio inicializado correctamente")
55
56      yield  # Aplicacion corre aqui
57
58      # Shutdown
59      logger.info("Cerrando servicio...")
60      if app_state.db_connection:
61          app_state.db_connection.close()
62      logger.info("Servicio cerrado")
63
64  app = FastAPI(
65      title="Predictor de Precios - BioPhys-Tech Lab",
66      description="Servicio de prediccion financiera con Gradient
        Boosting",
67      version="1.0.0",
68      lifespan=lifespan
69  )
70
71  @app.get("/health", tags=["Monitoreo"])
72  async def health_check():
73      """Health check endpoint para Kubernetes"""
74      return {
75          "status": "healthy",
76          "timestamp": datetime.utcnow().isoformat(),
77          "model_loaded": app_state.predictor is not None
78      }
```

```python

@app.get("/metrics", tags=["Monitoreo"])
async def get_metrics():
    """Metricas de performance del servicio"""
    return metrics.get_summary()

@app.post("/predict",
          response_model=PredictionResponse,
          status_code=200,
          tags=["Predicciones"])
async def predict(
    request: PredictionRequest,
    background_tasks: BackgroundTasks
):
    """
    Endpoint de prediccion principal

    **Latencia SLA**: < 200ms
    **Validacion**: Todos los inputs validados con Pydantic
    **Monitoreo**: Todas las predicciones registradas
    """

    start_time = time.time()
    request_id = request.request_id or str(uuid.uuid4())

    try:
        # Validacion de entrada (ya hecha por Pydantic)
        logger.debug(f"[{request_id}] Prediccion solicitada para {request.data.asset_id}")

        # Prediccion
        prediction_value = app_state.predictor.predict(
            asset_id=request.data.asset_id,
            features={
                'price': request.data.current_price,
                'volume': request.data.volume,
                'spread': request.data.bid_ask_spread,
                'momentum': request.data.momentum_24h,
                'volatility': request.data.volatility
            }
        )

        # Intervalo de confianza
```

```
121         confidence_interval = app_state.predictor.
    get_confidence_interval(
122             prediction_value
123         )
124 
125         # Calcular latencia
126         latency_ms = (time.time() - start_time) * 1000
127 
128         # Verificar SLA
129         if latency_ms > 200:
130             logger.warning(f"[{request_id}] SLA violation: {
    latency_ms:.2f}ms")
131             metrics.record_sla_violation()
132 
133         # Preparar response
134         response = PredictionResponse(
135             request_id=request_id,
136             prediction=float(prediction_value),
137             confidence_interval=confidence_interval,
138             model_version=app_state.predictor.version,
139             latency_ms=latency_ms,
140             timestamp=datetime.utcnow(),
141             status="success"
142         )
143 
144         # Registrar en background
145         background_tasks.add_task(
146             _log_prediction,
147             request_id=request_id,
148             request_data=request.data,
149             response=response
150         )
151 
152         # Actualizar drift detector
153         background_tasks.add_task(
154             _check_drift,
155             request_id=request_id,
156             features=[
157                 request.data.current_price,
158                 request.data.volume,
159                 request.data.bid_ask_spread,
160                 request.data.momentum_24h,
161                 request.data.volatility
```

```
            ],
            prediction=prediction_value
        )

        # Registrar en metricas
        metrics.record_prediction(
            asset_id=request.data.asset_id,
            latency_ms=latency_ms,
            status="success"
        )

        logger.info(f"[{request_id}] Prediccion exitosa: {
    prediction_value:.2f}")

        return response

    except ValueError as e:
        logger.error(f"[{request_id}] Validacion fallida: {str(e)}")
        metrics.record_prediction(
            asset_id=request.data.asset_id if request else "unknown",
            latency_ms=(time.time() - start_time) * 1000,
            status="validation_error"
        )
        raise HTTPException(status_code=422, detail=str(e))

    except Exception as e:
        logger.exception(f"[{request_id}] Error inesperado: {str(e)}"
    )
        metrics.record_prediction(
            asset_id=request.data.asset_id if request else "unknown",
            latency_ms=(time.time() - start_time) * 1000,
            status="error"
        )
        raise HTTPException(status_code=500, detail="Error interno
    del servidor")

async def _log_prediction(request_id: str, request_data:
    AssetPriceData,
                          response: PredictionResponse):
    """Registra prediccion en BD SQLite"""
    try:
        cursor = app_state.db_connection.cursor()
        cursor.execute("""
```

```
201             INSERT INTO predictions
202             (request_id, asset_id, input_price, prediction,
    latency_ms,
203              model_version, timestamp)
204             VALUES (?, ?, ?, ?, ?, ?, ?)
205         """, (
206             request_id,
207             request_data.asset_id,
208             request_data.current_price,
209             response.prediction,
210             response.latency_ms,
211             response.model_version,
212             response.timestamp.isoformat()
213         ))
214         app_state.db_connection.commit()
215         logger.debug(f"[{request_id}] Prediccion registrada en BD")
216     except Exception as e:
217         logger.error(f"[{request_id}] Error al registrar prediccion:
    {str(e)}")
218
219 async def _check_drift(request_id: str, features: list, prediction:
    float):
220     """Detecta drift en background"""
221     try:
222         drift_report = app_state.drift_detector.update_batch(
223             features=np.array([features]),
224             predictions=np.array([prediction])
225         )
226
227         if drift_report.get('status') != 'insufficient_data':
228             alert_level = drift_report['overall_drift']['alert_level'
    ]
229             if alert_level != 'NORMAL':
230                 logger.warning(
231                     f"[{request_id}] DRIFT DETECTED: {alert_level} -
    "
232                     f"Score: {drift_report['overall_drift']['
    overall_score']:.4f}"
233                 )
234                 metrics.record_drift_alert(alert_level)
235     except Exception as e:
236         logger.error(f"[{request_id}] Error en drift detection: {str(
    e)}")
```

```
237
238 def _init_database(conn: sqlite3.Connection):
239     """Inicializa tablas de base de datos"""
240     cursor = conn.cursor()
241     cursor.execute("""
242         CREATE TABLE IF NOT EXISTS predictions (
243             id INTEGER PRIMARY KEY AUTOINCREMENT,
244             request_id TEXT UNIQUE,
245             asset_id TEXT,
246             input_price REAL,
247             prediction REAL,
248             latency_ms REAL,
249             model_version TEXT,
250             timestamp TEXT,
251             created_at DATETIME DEFAULT CURRENT_TIMESTAMP
252         )
253     """)
254     cursor.execute("""
255         CREATE TABLE IF NOT EXISTS drift_alerts (
256             id INTEGER PRIMARY KEY AUTOINCREMENT,
257             timestamp TEXT,
258             alert_level TEXT,
259             overall_score REAL,
260             created_at DATETIME DEFAULT CURRENT_TIMESTAMP
261         )
262     """)
263     conn.commit()
264
265 @app.exception_handler(ValueError)
266 async def value_error_handler(request: Request, exc: ValueError):
267     """Manejador personalizado de errores de validacion"""
268     return JSONResponse(
269         status_code=422,
270         content=ErrorResponse(
271             error_code="VALIDATION_ERROR",
272             error_message=str(exc),
273             timestamp=datetime.utcnow(),
274             request_id=request.headers.get("X-Request-ID")
275         ).dict()
276     )
277
278 if __name__ == "__main__":
279     import uvicorn
```

```
    uvicorn.run(app, host="0.0.0.0", port=8000, workers=4)
```

Listing 3.2: API FastAPI con monitoreo y logging

3.2.4. Tests Unitarios Exhaustivos

```
import pytest
import numpy as np
from datetime import datetime
from src.models.validator import AssetPriceData, PredictionRequest
from src.models.predictor import GradientBoostingPredictor

class TestInputValidation:
    """Tests de validacion de entrada"""

    @pytest.fixture
    def valid_data(self):
        return {
            "asset_id": "BTC",
            "current_price": 45000.0,
            "volume": 1000000.0,
            "bid_ask_spread": 0.001,
            "momentum_24h": 0.05,
            "volatility": 0.02
        }

    def test_valid_input(self, valid_data):
        """Valida que entrada correcta sea aceptada"""
        data = AssetPriceData(**valid_data)
        assert data.asset_id == "BTC"
        assert data.current_price == 45000.0

    def test_negative_price_rejected(self, valid_data):
        """Rechaza precios negativos"""
        valid_data['current_price'] = -100.0
        with pytest.raises(ValueError):
            AssetPriceData(**valid_data)

    def test_zero_volume_rejected(self, valid_data):
        """Rechaza volumen cero"""
        valid_data['volume'] = 0
        with pytest.raises(ValueError):
            AssetPriceData(**valid_data)
```

```
38
39      def test_extreme_price_jump(self, valid_data):
40          """Rechaza saltos de precio anomalos (>500%)"""
41          valid_data['current_price'] = 225000.0  # 400% mas que
    baseline
42          with pytest.raises(ValueError, match="Cambio de precio
    sospechoso"):
43              AssetPriceData(**valid_data)
44
45      def test_invalid_spread(self, valid_data):
46          """Rechaza spread negativo"""
47          valid_data['bid_ask_spread'] = -0.001
48          with pytest.raises(ValueError):
49              AssetPriceData(**valid_data)
50
51      def test_missing_required_field(self, valid_data):
52          """Rechaza entrada sin campos requeridos"""
53          del valid_data['current_price']
54          with pytest.raises(ValueError):
55              AssetPriceData(**valid_data)
56
57  class TestPredictorBehavior:
58      """Tests del predictor bajo condiciones extremas"""
59
60      @pytest.fixture
61      def predictor(self):
62          return GradientBoostingPredictor(
63              model_path="models/gradient_boosting_v1.pkl",
64              scaler_path="models/scaler.pkl"
65          )
66
67      def test_prediction_within_bounds(self, predictor):
68          """Verifica que predicciones esten dentro de rangos
    razonables"""
69          features = {
70              'price': 45000.0,
71              'volume': 1000000.0,
72              'spread': 0.001,
73              'momentum': 0.05,
74              'volatility': 0.02
75          }
76          pred = predictor.predict('BTC', features)
77
```

```
78          # Prediccion debe estar en rango [precio*0.8, precio*1.2]
79          assert 36000 < pred < 54000
80
81      def test_prediction_consistency(self, predictor):
82          """Mismo input produce mismo output"""
83          features = {
84              'price': 45000.0,
85              'volume': 1000000.0,
86              'spread': 0.001,
87              'momentum': 0.05,
88              'volatility': 0.02
89          }
90
91          pred1 = predictor.predict('BTC', features)
92          pred2 = predictor.predict('BTC', features)
93
94          assert pred1 == pred2
95
96      def test_prediction_with_null_values(self, predictor):
97          """Maneja valores nulos apropiadamente"""
98          features = {
99              'price': 45000.0,
100             'volume': None,
101             'spread': 0.001,
102             'momentum': None,
103             'volatility': 0.02
104         }
105
106         # Debe imputar o rechazar
107         with pytest.raises((ValueError, TypeError)):
108             predictor.predict('BTC', features)
109
110     def test_latency_requirement(self, predictor):
111         """Verifica que latencia este bajo 200ms"""
112         import time
113
114         features = {
115             'price': 45000.0,
116             'volume': 1000000.0,
117             'spread': 0.001,
118             'momentum': 0.05,
119             'volatility': 0.02
120         }
```

```
121
122         start = time.time()
123         for _ in range(100):
124             predictor.predict('BTC', features)
125
126         avg_latency = (time.time() - start) / 100 * 1000
127
128         assert avg_latency < 200, f"Latencia promedio: {avg_latency
    :.2f}ms"
129
130 class TestDriftDetection:
131     """Tests del detector de drift"""
132
133     @pytest.fixture
134     def drift_detector(self):
135         from src.models.drift import DriftDetector
136         baseline = np.random.randn(1000, 5)
137         return DriftDetector(baseline, window_size=100)
138
139     def test_no_drift_with_baseline_distribution(self, drift_detector
    ):
140         """No detecta drift cuando datos vienen de baseline"""
141         baseline = np.random.randn(100, 5)
142         drift_detector.update_batch(baseline)
143
144         # Deberia reportar bajo drift
145         report = drift_detector.drift_history[-1] if drift_detector.
    drift_history else None
146         # Implementar asercion especifica
147
148     def test_detects_significant_shift(self, drift_detector):
149         """Detecta shift significativo en distribucion"""
150         # Generar datos con distribucion muy diferente
151         shifted_data = np.random.randn(100, 5) + 5  # Shift de 5
    sigma
152
153         drift_detector.update_batch(shifted_data)
154         # Deberia reportar alto drift
155
156 if __name__ == "__main__":
157     pytest.main([__file__, "-v", "--tb=short"])
```

Listing 3.3: Suite de tests unitarios

Parte III

Solución Data Engineer: Pipelines Resilientes

Capítulo 4

Cuestionario Técnico Data Engineer

4.1. Pregunta 1: Recuperación de Datos Faltantes

4.1.1. Análisis del Problema

Gaps en datos históricos de 10 minutos presentan desafíos:

$$\text{Sesgo}_{\text{entrenamiento}} = \mathbb{E}[Y|X_{\text{con gap}}] - \mathbb{E}[Y|X_{\text{sin gap}}] \tag{4.1}$$

Estrategias de Imputation

Método	**Sesgo**	**Aplicación**
Forward Fill	Bajo	Gap $<$ 30min
Interpolación Lineal	Muy Bajo	Gap $<$ 1h
MICE	Mínimo	Datos faltantes complejos
Exclusión	N/A	Gap $>$ 1h

Cuadro 4.1: Estrategias de imputation según duración del gap

4.1.2. Implementación: Data Recovery Pipeline

```
import pandas as pd
import numpy as np
from typing import Tuple, Dict
from datetime import timedelta, datetime
import logging

```

```python
class DataGapRecoveryEngine:
    """Motor de recuperacion de datos faltantes con multiples estrategias"""

    def __init__(self, recovery_config: Dict):
        """
        Args:
            recovery_config: Config con estrategias por duracion
                {
                    'max_forward_fill_minutes': 30,
                    'max_interpolate_minutes': 60,
                    'external_sources': ['yahoo', 'alpha_vantage'],
                    'log_gaps': True
                }
        """
        self.config = recovery_config
        self.logger = logging.getLogger(__name__)
        self.gap_registry = []  # Auditar todos los gaps

    def detect_and_recover_gaps(self,
                                df: pd.DataFrame,
                                freq: str = '5min') -> Tuple[pd.DataFrame, Dict]:
        """
        Detecta y recupera gaps en datos historicos

        Args:
            df: DataFrame con DatetimeIndex
            freq: Frecuencia esperada (e.g., '5min', '1min')

        Returns:
            Tupla (df_recovered, recovery_report)
        """

        self.logger.info(f"Iniciando analisis de gaps en {len(df)} registros")

        # Step 1: Generar serie de tiempo esperada
        expected_index = pd.date_range(
            start=df.index.min(),
            end=df.index.max(),
            freq=freq
        )
```

```
47
48          # Step 2: Detectar gaps
49          actual_index = df.index
50          missing_timestamps = expected_index.difference(actual_index)
51
52          gaps_detected = {
53              'num_gaps': len(missing_timestamps),
54              'total_missing_minutes': len(missing_timestamps) * int(
    freq.rstrip('min')),
55              'gap_details': []
56          }
57
58          if len(missing_timestamps) == 0:
59              self.logger.info("No se detectaron gaps")
60              return df, {'status': 'no_gaps'}
61
62          # Step 3: Agrupar gaps contiguos
63          gap_groups = self._group_consecutive_gaps(missing_timestamps)
64
65          # Step 4: Aplicar estrategia segun duracion
66          df_recovered = df.copy()
67
68          for gap_start, gap_timestamps in gap_groups.items():
69              gap_duration = len(gap_timestamps)
70
71              recovery_info = {
72                  'gap_start': gap_start,
73                  'gap_end': gap_timestamps[-1],
74                  'duration_minutes': gap_duration,
75                  'strategy': None,
76                  'success': False
77              }
78
79              # Seleccionar estrategia basada en duracion
80              if gap_duration <= self.config['max_forward_fill_minutes'
    ]:
81                  recovery_info['strategy'] = 'forward_fill'
82                  df_recovered = self._apply_forward_fill(
83                      df_recovered, gap_start, gap_timestamps
84                  )
85                  recovery_info['success'] = True
86
```

```
87              elif gap_duration <= self.config['max_interpolate_minutes
    ']:
88                  recovery_info['strategy'] = 'linear_interpolation'
89                  df_recovered = self._apply_interpolation(
90                      df_recovered, gap_start, gap_timestamps
91                  )
92                  recovery_info['success'] = True
93
94              else:
95                  # Intentar recuperacion desde fuente externa
96                  recovery_info['strategy'] = 'external_source'
97                  df_recovered = self._try_external_recovery(
98                      df_recovered, gap_start, gap_timestamps
99                  )
100                 if df_recovered is not None:
101                     recovery_info['success'] = True
102                 else:
103                     recovery_info['strategy'] = 'excluded'
104                     recovery_info['success'] = False
105
106             gaps_detected['gap_details'].append(recovery_info)
107             self.gap_registry.append(recovery_info)
108
109         # Step 5: Validar recuperacion
110         validation_report = self._validate_recovery(df, df_recovered)
111
112         self.logger.info(
113             f"Recuperacion completada: "
114             f"{sum(1 for g in gaps_detected['gap_details'] if g['
    success'])} "
115             f"de {len(gaps_detected['gap_details'])} gaps"
116         )
117
118         return df_recovered, {
119             'gaps_detected': gaps_detected,
120             'validation': validation_report
121         }
122
123     def _group_consecutive_gaps(self, missing_timestamps) -> Dict:
124         """Agrupa timestamps faltantes consecutivos"""
125         if len(missing_timestamps) == 0:
126             return {}
127
```

```
128         gap_groups = {}
129         current_group_start = missing_timestamps[0]
130         current_group = [missing_timestamps[0]]
131 
132         for ts in missing_timestamps[1:]:
133             # Si esta a 1 periodo de distancia, es parte del mismo
    gap
134             if (ts - current_group[-1]) == pd.Timedelta(minutes=5):
135                 current_group.append(ts)
136             else:
137                 # Nuevo gap
138                 gap_groups[current_group_start] = current_group
139                 current_group_start = ts
140                 current_group = [ts]
141 
142         gap_groups[current_group_start] = current_group
143         return gap_groups
144 
145     def _apply_forward_fill(self, df: pd.DataFrame,
146                             gap_start: datetime,
147                             gap_timestamps: list) -> pd.DataFrame:
148         """Aplica forward fill para gaps cortos"""
149         last_value = df.loc[:gap_start].iloc[-1] if gap_start in df.
    index else df.iloc[-1]
150 
151         for ts in gap_timestamps:
152             df.loc[ts] = last_value
153 
154         df.sort_index(inplace=True)
155         return df
156 
157     def _apply_interpolation(self, df: pd.DataFrame,
158                              gap_start: datetime,
159                              gap_timestamps: list) -> pd.DataFrame:
160         """Aplica interpolacion lineal para gaps medianos"""
161         # Insertar NaN en posiciones faltantes
162         for ts in gap_timestamps:
163             df.loc[ts] = np.nan
164 
165         # Ordenar e interpolar
166         df.sort_index(inplace=True)
167         df_numeric = df.select_dtypes(include=[np.number])
```

```
168         df[df_numeric.columns] = df_numeric.interpolate(method='
    linear')
169 
170         return df
171 
172     def _try_external_recovery(self, df: pd.DataFrame,
173                                gap_start: datetime,
174                                gap_timestamps: list) -> pd.DataFrame:
175         """Intenta recuperacion desde fuentes externas"""
176         for source in self.config.get('external_sources', []):
177             try:
178                 external_data = self._fetch_from_source(source,
    gap_timestamps)
179                 if external_data is not None:
180                     df.update(external_data)
181                     self.logger.info(f"Gap recuperado desde {source}"
    )
182                     return df
183             except Exception as e:
184                 self.logger.warning(f"Fallo recuperacion desde {
    source}: {str(e)}")
185 
186         return None
187 
188     def _fetch_from_source(self, source: str, timestamps: list):
189         """Fetch datos desde fuente externa (stub)"""
190         # Implementacion especifica por fuente
191         return None
192 
193     def _validate_recovery(self, df_original: pd.DataFrame,
194                            df_recovered: pd.DataFrame) -> Dict:
195         """Valida que recuperacion no introduzca sesgos"""
196 
197         # Verificar que no hay mas NaNs
198         nan_count_after = df_recovered.isna().sum().sum()
199 
200         # Verificar estadisticas no cambiaron radicalmente
201         stats_diff = {
202             'mean': abs(df_original.mean().mean() - df_recovered.mean
    ().mean()),
203             'std': abs(df_original.std().mean() - df_recovered.std().
    mean()),
```

```
204             'min': abs(df_original.min().mean() - df_recovered.min().
    mean()),
205             'max': abs(df_original.max().mean() - df_recovered.max().
    mean())
206         }
207
208         return {
209             'nan_count': nan_count_after,
210             'stats_diff': stats_diff,
211             'validation_passed': nan_count_after == 0 and all(
212                 v < 0.1 for v in stats_diff.values()  # <10%
    diferencia
213             )
214         }
```

Listing 4.1: Pipeline de recuperación de datos con detección de gaps

4.2. Pregunta 2: Series Temporales vs Bases de Datos Relacionales

4.2.1. Comparativa Arquitectónica

Característica	**TSDB**	**Relacional**
Ingesta de 10^6 puntos/min	✓	×
Compresión (10-100x)	✓	×
Downsampling automático	✓	×
Consultas complejas multitable	×	✓
Retraso de ingesta	< 100ms	> 1s
Costo de almacenamiento	Bajo	Alto

Cuadro 4.2: Comparativa TSDB vs Base de datos relacional

4.2.2. Fundamento Teórico

Las TSDB son optimizadas para operaciones de lectura/escritura secuencial:

$$T_{\text{TSDB}} = O(n \log k) \quad \text{vs} \quad T_{\text{SQL}} = O(n \log n) \tag{4.2}$$

Donde n es número de puntos y k es número de buckets de tiempo.

4.3. Pregunta 3: Manejo de Picos de Tráfico

4.3.1. Arquitectura Desacoplada

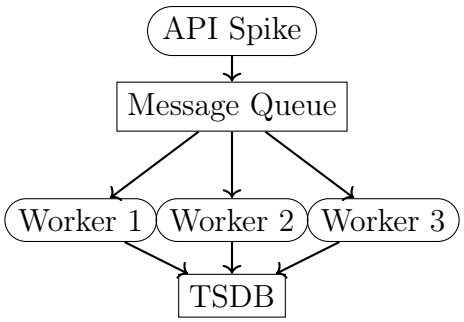

Figura 4.1: Arquitectura desacoplada con queue y workers escalables

Capítulo 5

Reto Técnico Data Engineer: Implementación Completa

5.1. Pipeline de Ingesta y Limpieza

5.1.1. Simulador de API de Streaming

```python
import json
import time
import numpy as np
from typing import Generator, Dict
import asyncio
import random

class RealisticPriceStreamSimulator:
    """
    Simula feed de precios realista con:
    - Saltos de precio anomalos
    - Valores null aleatorios
    - Cambios extremos durante volatilidad alta
    """

    def __init__(self,
                 initial_price: float = 100.0,
                 volatility_base: float = 0.02):
        self.current_price = initial_price
        self.volatility = volatility_base
        self.volatility_spike_prob = 0.01  # 1% prob. spike de volatilidad
        self.null_prob = 0.005  # 0.5% prob. valor null
```

```
23          self.anomaly_prob = 0.02  # 2% prob. precio anomalo
24          self.event_counter = 0
25          self.anomalies_injected = []
26
27      def generate_stream(self, n_points: int = 1000) -> Generator[Dict
    , None, None]:
28          """Genera stream de n_points con anomalias realistas"""
29
30          for i in range(n_points):
31              self.event_counter = i
32
33              # Posible spike de volatilidad (evento de mercado)
34              if random.random() < self.volatility_spike_prob:
35                  self.volatility = min(self.volatility * 5, 1.0)
36                  print(f"[Event {i}] Spike de volatilidad: {self.
    volatility:.4f}")
37              else:
38                  self.volatility = max(self.volatility * 0.95, 0.02)
39
40              # Generar precio base
41              daily_return = np.random.normal(0, self.volatility)
42              self.current_price *= (1 + daily_return)
43
44              # Inyectar anomalias
45              price_to_send = self.current_price
46
47              if random.random() < self.null_prob:
48                  price_to_send = None
49                  self.anomalies_injected.append({
50                      'type': 'null',
51                      'index': i,
52                      'timestamp': time.time()
53                  })
54
55              elif random.random() < self.anomaly_prob:
56                  # Salto de precio extremo (500%)
57                  jump_direction = random.choice([1, -1])
58                  jump_magnitude = random.uniform(5, 20)  # 500%-2000%
    jump
59                  price_to_send = self.current_price * (1 +
    jump_direction * jump_magnitude)
60                  self.anomalies_injected.append({
61                      'type': 'extreme_jump',
```

```
                    'index': i,
                    'magnitude': jump_magnitude * 100,
                    'timestamp': time.time()
                })

            yield {
                'asset_id': 'BTC',
                'timestamp': time.time(),
                'price': price_to_send,
                'volume': np.random.uniform(1000, 1000000),
                'bid': price_to_send * 0.999 if price_to_send else None,
                'ask': price_to_send * 1.001 if price_to_send else None
            }

            time.sleep(0.001)  # 1ms entre puntos

    def get_anomaly_report(self) -> Dict:
        """Reporte de anomalias inyectadas"""
        return {
            'total_anomalies': len(self.anomalies_injected),
            'null_values': len([a for a in self.anomalies_injected if a['type'] == 'null']),
            'extreme_jumps': len([a for a in self.anomalies_injected if a['type'] == 'extreme_jump']),
            'anomalies': self.anomalies_injected
        }

class DataIngestionPipeline:
    """Pipeline de ingesta con validacion, outlier detection y limpieza"""

    def __init__(self, db_connection, queue_size: int = 10000):
        self.db = db_connection
        self.queue = asyncio.Queue(maxsize=queue_size)
        self.stats = {
            'processed': 0,
            'valid': 0,
            'filtered_outliers': 0,
            'filtered_nulls': 0,
            'deduplicated': 0,
            'stored': 0
```

```
        }
        self.last_seen = {}  # Para deduplicacion

    async def ingest_from_stream(self, stream: Generator):
        """Consume stream y coloca en queue"""
        for record in stream:
            try:
                await asyncio.wait_for(
                    self.queue.put(record),
                    timeout=1.0
                )
            except asyncio.TimeoutError:
                print(f"Queue llena, descartando record")

    async def process_queue(self):
        """Procesa items de queue con validacion y limpieza"""
        while True:
            try:
                record = await asyncio.wait_for(
                    self.queue.get(),
                    timeout=5.0
                )

                self.stats['processed'] += 1

                # Step 1: Validacion
                if not self._validate_record(record):
                    self.stats['filtered_nulls'] += 1
                    continue

                # Step 2: Deteccion de outliers
                if self._is_outlier(record):
                    self.stats['filtered_outliers'] += 1
                    continue

                # Step 3: Deduplicacion
                if self._is_duplicate(record):
                    self.stats['deduplicated'] += 1
                    continue

                # Step 4: Almacenamiento
                self._store_record(record)
                self.stats['valid'] += 1
```

```
143                 self.stats['stored'] += 1
144
145             except asyncio.TimeoutError:
146                 break
147
148     def _validate_record(self, record: Dict) -> bool:
149         """Valida integridad basica"""
150         if record.get('price') is None:
151             return False
152         if record.get('timestamp') is None:
153             return False
154         if not isinstance(record.get('price'), (int, float)):
155             return False
156         return True
157
158     def _is_outlier(self, record: Dict) -> bool:
159         """Detecta outliers usando metodo IQR"""
160         # Implementacion simplificada
161         price = record['price']
162
163         # Rechazar precios negativos
164         if price < 0:
165             return True
166
167         # Rechazar precios extremos
168         if price > 1e6:
169             return True
170
171         # Comparar con historico
172         asset_id = record['asset_id']
173         if asset_id in self.last_seen:
174             last_price = self.last_seen[asset_id]['price']
175             price_change = abs(price - last_price) / last_price
176
177             # Rechazar cambios >500%
178             if price_change > 5.0:
179                 return True
180
181         return False
182
183     def _is_duplicate(self, record: Dict) -> bool:
184         """Previene registros duplicados"""
185         key = (record['asset_id'], round(record['timestamp'], 1))
```

```
186
187         if key in self.last_seen:
188             # Si vimos este timestamp hace poco, es probable
    duplicado
189             return True
190
191         self.last_seen[record['asset_id']] = record
192         return False
193
194     def _store_record(self, record: Dict):
195         """Almacena en TSDB (stub)"""
196         # En produccion: InfluxDB, QuestDB, etc.
197         pass
198
199     def get_stats(self) -> Dict:
200         """Retorna estadisticas de procesamiento"""
201         return {
202             **self.stats,
203             'acceptance_rate': self.stats['valid'] / max(self.stats['
    processed'], 1) * 100
204         }
```

Listing 5.1: Simulador de API con volatilidad realista

5.1.2. OHLC Aggregator con Tolerancia a Fallos

```
1  import pandas as pd
2  from datetime import datetime, timedelta
3  from typing import Dict
4  import logging
5
6  class OHLCAggregator:
7      """Calcula Open, High, Low, Close por minuto con recuperacion"""
8
9      def __init__(self, db_connection):
10         self.db = db_connection
11         self.buffer = {}  # Buffer en memoria para minutos actuales
12         self.logger = logging.getLogger(__name__)
13
14     def aggregate_minute(self,
15                          asset_id: str,
16                          minute: datetime,
17                          prices: pd.Series) -> Dict:
```

```
18          """
19          Calcula OHLC para un minuto
20
21          Args:
22              asset_id: ID del activo
23              minute: Timestamp de minuto (inicio)
24              prices: Series de precios del minuto
25          """
26
27          if prices.empty:
28              self.logger.warning(f"Minuto {minute} sin datos para {
    asset_id}")
29              return None
30
31          ohlc_data = {
32              'asset_id': asset_id,
33              'timestamp': minute,
34              'open': float(prices.iloc[0]),
35              'high': float(prices.max()),
36              'low': float(prices.min()),
37              'close': float(prices.iloc[-1]),
38              'volume': len(prices),  # Count de ticks
39              'vwap': self._calculate_vwap(prices)
40          }
41
42          return ohlc_data
43
44      @staticmethod
45      def _calculate_vwap(prices: pd.Series) -> float:
46          """Volume-weighted average price"""
47          # Implementacion: necesitaria volumenes
48          return float(prices.mean())
49
50      def store_ohlc(self, ohlc_data: Dict):
51          """Almacena OHLC con fallback"""
52          try:
53              # Intentar almacenar en TSDB principal
54              self._insert_tsdb(ohlc_data)
55          except Exception as e:
56              self.logger.error(f"Error en TSDB principal: {str(e)}")
57              # Fallback: almacenar en SQLite local
58              self._insert_sqlite_fallback(ohlc_data)
59
```

```
    def _insert_tsdb(self, ohlc_data: Dict):
        """Inserta en TimeSeries DB (InfluxDB, QuestDB)"""
        # Stub para ejemplo
        pass

    def _insert_sqlite_fallback(self, ohlc_data: Dict):
        """Fallback a SQLite para recuperacion"""
        cursor = self.db.cursor()
        cursor.execute("""
            INSERT INTO ohlc_fallback
            (asset_id, timestamp, open, high, low, close, volume)
            VALUES (?, ?, ?, ?, ?, ?, ?)
        """, (
            ohlc_data['asset_id'],
            ohlc_data['timestamp'].isoformat(),
            ohlc_data['open'],
            ohlc_data['high'],
            ohlc_data['low'],
            ohlc_data['close'],
            ohlc_data['volume']
        ))
        self.db.commit()
```

Listing 5.2: Agregador OHLC resiliente

5.2. Pregunta 4: Pipeline Tolerante a Fallos

5.2.1. Resiliencia Mediante Pattern: Circuit Breaker

```
from enum import Enum
import time
from typing import Callable, Any

class CircuitState(Enum):
    CLOSED = "closed"        # Normal operation
    OPEN = "open"            # Failing, reject
    HALF_OPEN = "half_open"  # Testing recovery

class CircuitBreaker:
    """Circuit breaker con fallback automatico"""

    def __init__(self,
```

```
                 failure_threshold: int = 5,
                 recovery_timeout: int = 60,
                 expected_exception: Exception = Exception):
        """
        Args:
            failure_threshold: Numero de fallos antes de abrir
            recovery_timeout: Segundos antes de intentar recuperacion
            expected_exception: Tipo de excepcion a capturar
        """
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.expected_exception = expected_exception

        self.failure_count = 0
        self.last_failure_time = None
        self.state = CircuitState.CLOSED

    def call(self, func: Callable, *args, fallback: Callable = None,
**kwargs) -> Any:
        """
        Ejecuta funcion con proteccion de circuit breaker

        Args:
            func: Funcion a ejecutar
            fallback: Funcion alternativa si esta abierto
        """

        if self.state == CircuitState.OPEN:
            if self._should_attempt_reset():
                self.state = CircuitState.HALF_OPEN
                print(f"[CircuitBreaker] Intentando recuperacion")
            else:
                if fallback:
                    print(f"[CircuitBreaker] OPEN - usando fallback")
                    return fallback(*args, **kwargs)
                raise Exception("Circuit breaker is OPEN")

        try:
            result = func(*args, **kwargs)
            self._on_success()
            return result

        except self.expected_exception as e:
```

```
56              self._on_failure()
57              if fallback:
58                  print(f"[CircuitBreaker] Fallback despues de fallo: {
    str(e)}")
59                  return fallback(*args, **kwargs)
60              raise
61
62      def _on_success(self):
63          """Registra exito y resetea counters"""
64          self.failure_count = 0
65          self.state = CircuitState.CLOSED
66
67      def _on_failure(self):
68          """Registra fallo y evalua apertura"""
69          self.failure_count += 1
70          self.last_failure_time = time.time()
71
72          if self.failure_count >= self.failure_threshold:
73              self.state = CircuitState.OPEN
74              print(f"[CircuitBreaker] OPENED despues de {self.
    failure_count} fallos")
75
76      def _should_attempt_reset(self) -> bool:
77          """Evalua si es hora de intentar recuperacion"""
78          if self.last_failure_time is None:
79              return False
80
81          elapsed = time.time() - self.last_failure_time
82          return elapsed >= self.recovery_timeout
83
84  class FaultTolerantPipeline:
85      """Pipeline con recuperacion automatica y fallbacks"""
86
87      def __init__(self):
88          self.primary_breaker = CircuitBreaker(
89              failure_threshold=3,
90              recovery_timeout=60
91          )
92          self.backup_breaker = CircuitBreaker(
93              failure_threshold=5,
94              recovery_timeout=120
95          )
96
```

```python
    def process_with_failover(self, data):
        """Procesa con multiples niveles de failover"""

        # Level 1: Intenta primario
        try:
            return self.primary_breaker.call(
                self._primary_processor,
                data,
                fallback=self._backup_processor
            )
        except Exception as e:
            print(f"[Pipeline] Fallo primario: {str(e)}")

            # Level 2: Backup explicito
            return self.backup_breaker.call(
                self._backup_processor,
                data,
                fallback=self._local_cache_fallback
            )

    def _primary_processor(self, data):
        """Procesador primario (e.g., TSDB remoto)"""
        # Simulacion: puede fallar
        if np.random.random() < 0.1:  # 10% prob. fallo
            raise ConnectionError("TSDB timeout")
        return "processed_primary"

    def _backup_processor(self, data):
        """Procesador backup (e.g., SQLite local)"""
        # Implementacion de fallback
        return "processed_backup"

    def _local_cache_fallback(self, data):
        """Fallback final: cache local"""
        return "processed_local_cache"
```

Listing 5.3: Circuit Breaker pattern para pipelines

Capítulo 6

Mecanismos de Seguridad: Preguntas Bonus

6.1. ML Engineer Bonus: Safety Mechanism contra Volatilidad Extrema

6.1.1. Framework: Adaptive Risk Management

Cuando el modelo genera predicciones en mercado 10x más volátil y pierde dinero drásticamente, se requiere:

1. **Early Warning System**: Detección de cambio de régimen de mercado
2. **Automatic Position Sizing**: Reducción dinámica de exposición
3. **Circuit Breaker**: Halt automático de trading

```python
import numpy as np
from dataclasses import dataclass
from enum import Enum

class MarketRegime(Enum):
    NORMAL = "normal"
    ELEVATED = "elevated"
    CRISIS = "crisis"

@dataclass
class RiskMetrics:
    vix_equivalent: float  # Volatilidad implicita
```

```
    drawdown_current: float  # Drawdown actual
    loss_rate: float  # Tasa de perdida
    prediction_confidence: float  # Confianza del modelo

class SafetyBreakerSystem:
    """Sistema que detiene trading automaticamente ante volatilidad extrema"""

    def __init__(self):
        self.regime = MarketRegime.NORMAL
        self.regime_history = []
        self.max_drawdown_threshold = 0.10  # 10% max drawdown
        self.position_multiplier = 1.0
        self.trading_enabled = True

        # Thresholds de cambio de regimen
        self.regime_thresholds = {
            'normal_to_elevated': 0.03,      # 3x volatilidad baseline
            'elevated_to_crisis': 0.1,       # 10x volatilidad baseline
            'crisis_recovery': 0.05           # 5x para recuperacion
        }

    def evaluate_market_conditions(self,
                                   current_metrics: RiskMetrics) -> Dict:
        """
        Evalua condiciones de mercado y toma decisiones automaticas

        Returns:
            {
                'regime': MarketRegime,
                'action': 'continue' | 'reduce_position' | 'halt_trading',
                'position_multiplier': float,
                'reason': str
            }
        """

        # Step 1: Deteccion de cambio de regimen
        new_regime = self._classify_regime(current_metrics)

```

```
51          if new_regime != self.regime:
52              self.regime = new_regime
53              self.regime_history.append({
54                  'timestamp': datetime.now(),
55                  'old_regime': self.regime,
56                  'new_regime': new_regime,
57                  'trigger_metrics': current_metrics
58              })
59              print(f"[REGIME CHANGE] {self.regime.name}")
60
61          # Step 2: Evaluacion de metricas de riesgo
62          action = 'continue'
63          reason = 'Operacion normal'
64
65          if current_metrics.drawdown_current > self.
    max_drawdown_threshold:
66              self.trading_enabled = False
67              action = 'halt_trading'
68              reason = f"Drawdown {current_metrics.drawdown_current
    :.1%} > threshold"
69
70          elif self.regime == MarketRegime.CRISIS:
71              self.position_multiplier = 0.2  # 20% posicion
72              action = 'reduce_position'
73              reason = "Regimen de crisis detectado"
74
75          elif self.regime == MarketRegime.ELEVATED:
76              self.position_multiplier = 0.5  # 50% posicion
77              action = 'reduce_position'
78              reason = "Volatilidad elevada"
79
80          else:
81              self.position_multiplier = 1.0
82              action = 'continue'
83
84          # Step 3: Validacion de confianza del modelo
85          if current_metrics.prediction_confidence < 0.6:
86              action = 'reduce_position'
87              reason = f"Baja confianza: {current_metrics.
    prediction_confidence:.1%}"
88
89          return {
90              'regime': self.regime,
```

```
            'action': action,
            'position_multiplier': self.position_multiplier,
            'reason': reason,
            'trading_enabled': self.trading_enabled
        }

    def _classify_regime(self, metrics: RiskMetrics) -> MarketRegime:
        """Clasifica regimen de mercado basado en volatilidad"""

        # VIX-like metric: ratio de volatilidad respecto a baseline
        volatility_ratio = metrics.vix_equivalent

        if volatility_ratio > self.regime_thresholds['
    elevated_to_crisis']:
            return MarketRegime.CRISIS

        elif volatility_ratio > self.regime_thresholds['
    normal_to_elevated']:
            return MarketRegime.ELEVATED

        else:
            return MarketRegime.NORMAL

    def get_position_adjustment(self,
                                base_position_size: float) -> float:
        """Ajusta tamano de posicion segun condiciones"""

        if not self.trading_enabled:
            return 0.0

        return base_position_size * self.position_multiplier

    def get_system_status(self) -> Dict:
        """Status completo del sistema"""
        return {
            'current_regime': self.regime.value,
            'position_multiplier': self.position_multiplier,
            'trading_enabled': self.trading_enabled,
            'regime_history_length': len(self.regime_history),
            'recent_transitions': self.regime_history[-5:] if self.
    regime_history else []
        }
```

Listing 6.1: Safety system contra volatilidad extrema

6.2. Data Engineer Bonus: Consenso Multi-proveedor

6.2.1. Algoritmo de Reconciliación

Cuando recibimos datos de 3 proveedores con discrepancias:

$$P_{\text{consenso}} = \text{argmin}_p \sum_i w_i |p - p_i| \tag{6.1}$$

Donde w_i es peso del proveedor i (basado en histórico de confiabilidad).

```python
import numpy as np
from typing import Dict, List
from scipy.stats import median_abs_deviation as mad

class MultiProviderPriceConsensus:
    """Reconcilia precios de multiples proveedores"""

    def __init__(self, providers: List[str]):
        """
        Args:
            providers: Lista de IDs de proveedores
        """
        self.providers = providers
        self.provider_stats = {
            p: {'errors': 0, 'total': 0, 'reliability': 1.0}
            for p in providers
        }
        self.consensus_history = []

    def reconcile_price(self,
                        prices: Dict[str, float],
                        timestamp: str) -> Dict:
        """
        Reconcilia precios de multiples proveedores

        Args:
            prices: {'provider1': 100.5, 'provider2': 100.45, '
    provider3': 100.6}

        Returns:
            {
                'consensus_price': 100.5,
                'method': 'weighted_median',
                'confidence': 0.95,
```

```
34                 'outliers': ['provider3'],
35                 'reasoning': str
36             }
37         """
38 
39         # Step 1: Validacion inicial
40         valid_prices = {p: v for p, v in prices.items() if v > 0}
41 
42         if len(valid_prices) == 0:
43             return self._handle_no_valid_prices()
44 
45         if len(valid_prices) == 1:
46             provider, price = list(valid_prices.items())[0]
47             return {
48                 'consensus_price': price,
49                 'method': 'single_provider',
50                 'confidence': 0.5,  # Baja confianza con un proveedor
51                 'outliers': [],
52                 'reasoning': f'Solo {provider} tiene precio valido'
53             }
54 
55         # Step 2: Deteccion de outliers con MAD (Median Absolute
    Deviation)
56         price_values = np.array(list(valid_prices.values()))
57         median_price = np.median(price_values)
58         deviations = np.abs(price_values - median_price)
59         mad_value = mad(price_values)
60 
61         # Threshold: |precio - mediana| > 3*MAD
62         outlier_threshold = 3 * mad_value
63 
64         outliers = []
65         inliers = {}
66 
67         for provider, price in valid_prices.items():
68             deviation = abs(price - median_price)
69 
70             if deviation > outlier_threshold and len(valid_prices) >
    2:
71                 outliers.append(provider)
72             else:
73                 inliers[provider] = price
74 
```

```
75          # Step 3: Calculo de consenso
76          if len(inliers) == 0:
77              # Todos son outliers, usar mediana completa
78              consensus_price = median_price
79              method = 'median_all'
80          else:
81              # Usar weighted median de inliers
82              inlier_prices = np.array(list(inliers.values()))
83              inlier_providers = list(inliers.keys())
84
85              # Pesos basados en confiabilidad historica
86              weights = np.array([
87                  self.provider_stats[p]['reliability']
88                  for p in inlier_providers
89              ])
90
91              weights = weights / weights.sum()
92
93              # Weighted median
94              consensus_price = self._weighted_median(inlier_prices,
    weights)
95              method = 'weighted_median_inliers'
96
97          # Step 4: Calculo de confianza
98          confidence = self._calculate_confidence(
99              valid_prices, consensus_price, outliers
100         )
101
102         # Step 5: Actualizar estadisticas de proveedores
103         self._update_provider_stats(valid_prices, consensus_price,
    outliers)
104
105         result = {
106             'consensus_price': float(consensus_price),
107             'method': method,
108             'confidence': float(confidence),
109             'outliers': outliers,
110             'outlier_reasons': [
111                 f'{p}: {abs(valid_prices[p] - median_price)/
    median_price*100:.2f}% "
112                 f"del mediana'
113                 for p in outliers
114             ],
```

```
115             'provider_stats': {
116                 p: self.provider_stats[p]
117                 for p in self.providers
118             }
119         }
120 
121         self.consensus_history.append({
122             'timestamp': timestamp,
123             'result': result
124         })
125 
126         return result
127 
128     @staticmethod
129     def _weighted_median(values: np.ndarray, weights: np.ndarray) ->
    float:
130         """Calcula mediana ponderada"""
131         sorted_indices = np.argsort(values)
132         sorted_values = values[sorted_indices]
133         sorted_weights = weights[sorted_indices]
134 
135         cumsum_weights = np.cumsum(sorted_weights)
136         target_weight = 0.5
137 
138         idx = np.argmax(cumsum_weights >= target_weight)
139         return sorted_values[idx]
140 
141     def _calculate_confidence(self,
142                               prices: Dict[str, float],
143                               consensus: float,
144                               outliers: List[str]) -> float:
145         """Calcula confianza en consenso"""
146 
147         # Menos outliers = mas confianza
148         outlier_ratio = len(outliers) / len(prices)
149 
150         # Menos dispersion = mas confianza
151         price_values = np.array(list(prices.values()))
152         coefficient_variation = np.std(price_values) / np.mean(
    price_values)
153 
154         # Combinar factores
155         confidence = (1 - outlier_ratio) * (1 - min(
```

```
    coefficient_variation, 0.1))
156
157         return max(0.1, min(1.0, confidence))
158
159     def _update_provider_stats(self,
160                                prices: Dict[str, float],
161                                consensus: float,
162                                outliers: List[str]):
163         """Actualiza confiabilidad de proveedores"""
164
165         for provider, price in prices.items():
166             self.provider_stats[provider]['total'] += 1
167
168             if provider in outliers:
169                 self.provider_stats[provider]['errors'] += 1
170
171             # Confiabilidad = (1 - error_rate)
172             error_rate = self.provider_stats[provider]['errors'] / \
173                          max(self.provider_stats[provider]['total'],
    1)
174
175             self.provider_stats[provider]['reliability'] = 1 - min(
    error_rate, 1.0)
176
177     def _handle_no_valid_prices(self) -> Dict:
178         """Maneja caso sin precios validos"""
179         return {
180             'consensus_price': None,
181             'method': 'no_valid_data',
182             'confidence': 0.0,
183             'outliers': list(self.providers),
184             'reasoning': 'Todos los proveedores retornaron precios
    invalidos'
185         }
```

Listing 6.2: Sistema de consenso multi-proveedor

Capítulo 7

Conclusión: Arquitectura Integrada

7.1. Sistema Completo: Flujo End-to-End

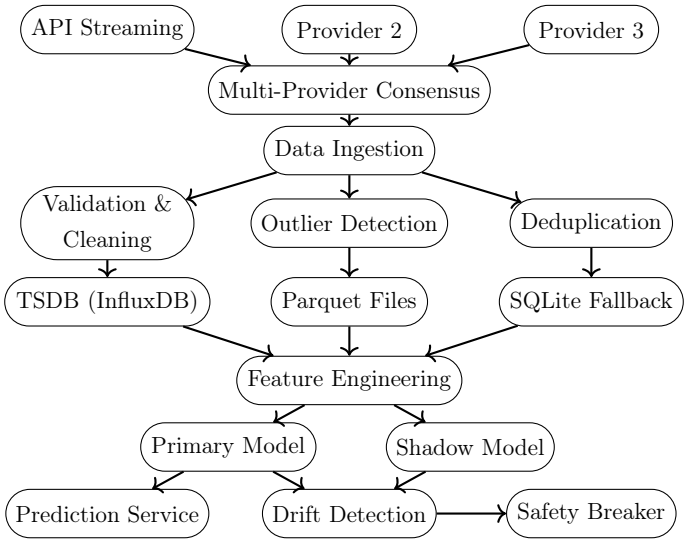

Figura 7.1: Arquitectura integrada del sistema de predicción (Corregida)

7.2. Key Insights y Recomendaciones

7.2.1. Diseño para Producción

1. **Separación de concernos**: Data layer, ML layer, API layer están desacoplados
2. **Resiliencia multi-nivel**: Circuit breakers, fallbacks, caches locales
3. **Observabilidad**: Logging exhaustivo, métricas, alertas automáticas
4. **Validación en todos los niveles**: Pydantic, schema validation, sanity checks

7.2.2. Métricas Críticas de Monitoreo

Métrica	**Target**	**Acción**
Latencia P99	< 150ms	Alertar si > 180ms
SLA Violation	$< 5\,\%$	Invocar escalación
Model Drift Score	< 0.15	Retrain si > 0.2
Data Gap Duration	< 10 min	Investigar si > 30min
Deduplication Rate	$< 1\,\%$	Review si $> 2\,\%$
Provider Consensus	> 0.9	Flag si < 0.8

Cuadro 7.1: SLAs y umbrales de monitoreo

7.3. Roadmap de Implementación

1. **Fase 1 (Semanas 1-2)**: Implementar validadores Pydantic y tests unitarios
2. **Fase 2 (Semanas 3-4)**: Containerización Docker y CI/CD pipeline
3. **Fase 3 (Semanas 5-6)**: Data pipeline con TSDB y fallbacks
4. **Fase 4 (Semanas 7-8)**: Drift detection y retraining automático
5. **Fase 5 (Semanas 9-10)**: Safety systems y circuit breakers
6. **Fase 6 (Semanas 11-12)**: Load testing, optimization, deployment

Apéndice A

Referencias Técnicas

A.1. Librerías Recomendadas

- **ML**: scikit-learn, XGBoost, ONNX, SHAP
- **Data**: pandas, numpy, pyarrow, DuckDB
- **API**: FastAPI, Pydantic, uvicorn
- **Storage**: InfluxDB, QuestDB, SQLite
- **Monitoring**: Prometheus, Grafana, ELK Stack
- **Testing**: pytest, hypothesis, locust

A.2. Estándares de Código

- Linting: pylint, flake8, black
- Type hints: mypy
- Documentation: Sphinx, docstrings
- Versionamiento: Git flow