

π-Bot: Plan de Mejora y Optimización

Resumen Ejecutivo

Este documento detalla el plan de mejora para el sistema π-Bot, un bot de trading algorítmico intradía basado en Machine Learning. El análisis identifica fortalezas actuales y propone un roadmap estructurado para transformar el prototipo actual en un sistema robusto de producción.

Estado Actual del Sistema

Arquitectura Existente

El sistema sigue un pipeline de 4 fases:

- **DAT**: Descarga de datos desde IBKR → Parquet
- **RSH**: Research y análisis de modelos/escenarios
- **TRN**: Entrenamiento de modelos ML
- **OPS**: Operativa en papel (paper trading)

Fortalezas Identificadas

- Arquitectura modular clara y bien separada
- Uso de MLflow para tracking de experimentos
- Método Triple Barrera para etiquetado sofisticado
- Time Series Cross-Validation implementado
- Gestión de riesgo dinámica basada en ATR
- Pipeline de features técnicos robusto
- Dashboard Streamlit funcional

Problemas Críticos Detectados

● Crítico - Resolver Inmediatamente

1. **Configuración fragmentada**: Settings dispersos entre múltiples archivos
2. **Manejo de datos inconsistente**: Mezcla CSV/Parquet con paths hardcodeados
3. **Logging heterogéneo**: Combinación de print() y logging estructurado
4. **Gestión de errores reactiva**: Falta prevención proactiva de fallos

● Alta Prioridad - Próximas 2 Semanas

1. **Validación de datos insuficiente**: Control de calidad sistemático limitado
2. **Testing ausente**: No hay tests unitarios ni de integración

3. Reconexiones IBKR no robustas: Fallos en conectividad no manejados

4. Model drift no detectado: Degradación de modelos sin alertas

● Media Prioridad - Próximo Mes

1. Backtesting básico: Falta análisis de sensibilidad y Monte Carlo

2. Feature selection manual: No hay selección automatizada

3. A/B testing ausente: No hay comparación de estrategias

4. Monitoring limitado: Métricas real-time insuficientes

Plan de Mejora por Fases

FASE 1: Fundamentos (1-2 semanas)

1.1 Unificación del Sistema de Configuración

Objetivo: Centralizar toda la configuración en archivos YAML estructurados

yaml

```

# config/settings.yaml
data:
  storage_backend: "parquet"
  base_path: "data/"
  quality_checks:
    min_coverage_pct: 80
    max_missing_days: 5

models:
  default_params:
    tp_multiplier: 3.0
    sl_multiplier: 2.0
    time_limit_candles: 16

  optimization:
    cv_splits: 5
    test_size: 500

trading:
  capital_per_trade: 1000.0
  max_positions: 10
  risk_limits:
    max_daily_loss: 500.0
    max_drawdown: 0.05

logging:
  level: "INFO"
  structured: true
  correlation_id: true

```

Entregables:

- Archivo de configuración centralizado
- ConfigManager class para acceso unificado
- Migración de todos los settings hardcodeados
- Validación de configuración al inicio

1.2 Sistema de Logging Estructurado

Objetivo: Implementar logging consistente y trazabilidad completa

python

```

# utils/logging_enhanced.py
import structlog

def setup_logging():
    structlog.configure(
        processors=[
            structlog.stdlib.filter_by_level,
            structlog.stdlib.add_logger_name,
            structlog.stdlib.add_log_level,
            structlog.stdlib.PositionalArgumentsFormatter(),
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.processors.StackInfoRenderer(),
            structlog.processors.format_exc_info,
            structlog.processors.JSONRenderer()
        ]
    )

# Uso: logger.info("trade_executed", ticker="AAPL", pnl=150.0, correlation_id="abc123")

```

Entregables:

- Sistema de logging estructurado
- Correlation IDs para trazabilidad
- Migración de todos los print() statements
- Log aggregation y alerting básico

1.3 Validación de Datos Robusta

Objetivo: Implementar control de calidad sistemático de datos

python

```

# utils/data_quality.py
class DataQualityManager:
    ... def validate_ohlc(self, df: pd.DataFrame) -> DataQualityReport:
        """Valida calidad de datos OHLC"""
        checks = [
            self._check_temporal_coverage(),
            self._check_price_consistency(),
            self._check_volume_anomalies(),
            self._check_missing_data(),
        ]
        return DataQualityReport(checks)

    ... def detect_feature_drift(self, current: pd.DataFrame,
                                baseline: pd.DataFrame) -> DriftReport:
        """Detecta drift en features"""
        pass

```

Entregables:

- Data quality checks automatizados
- Feature drift detection
- Alertas automáticas por calidad de datos
- Dashboard de health de datos

FASE 2: Optimización del Pipeline (2-3 semanas)

2.1 Mejora del Módulo de Datos (DAT)

Objetivos:

- Gestión inteligente de reconexiones IBKR
- Paralelización real de descargas
- Compresión automática de históricos
- Monitoring de data freshness

python

```

# modules/data_manager_v2.py
class IBKRConnectionManager:
    ... def __init__(self, max_retries=5, backoff_factor=2):
        self.connection_pool = []
        self.retry_policy = RetryPolicy(max_retries, backoff_factor)
    ...
    ... async def download_with_retry(self, ticker: str) -> DownloadResult:
        """Descarga con reconexión automática"""
        pass

class DataCompressionManager:
    ... def compress_historical_data(self, older_than_days=90):
        """Comprime datos históricos automáticamente"""
        pass

```

Entregables:

- Connection pooling para IBKR
- Retry logic inteligente con backoff exponencial
- Compresión automática de datos antiguos
- Paralelización real (no solo threading)
- Monitoring de data freshness por ticker

2.2 Research Avanzado (RSH)

Objetivos:

- Optimización de hiperparámetros con Optuna
- Feature selection automatizada
- Early stopping en CV
- Análisis de estabilidad temporal

python

```

# modules/research_v2.py
class HyperparameterOptimizer:
... def __init__(self, backend="optuna"):
...     self.optimizer = optuna.create_study()

... def optimize(self, ticker: str, objective_func: Callable) -> OptimizationResult:
...     """Optimización bayesiana de hiperparámetros"""
...     pass

class FeatureSelector:
...     def select_features(self, X: pd.DataFrame, y: pd.Series) -> List[str]:
...         """Selección automática de features"""
...         methods = [
...             self._recursive_elimination(),
...             self._mutual_info_selection(),
...             self._stability_selection()
...         ]
...         return self._ensemble_selection(methods)

```

Entregables:

- Integración con Optuna para HPO
- Feature selection automatizada multi-método
- Early stopping en validación cruzada
- Análisis de estabilidad temporal de modelos
- Ensemble de métodos de selección

2.3 Entrenamiento Inteligente (TRN)

Objetivos:

- Auto-reentrenamiento basado en métricas
- Model versioning automático
- Drift detection en producción

python

```

# modules/training_v2.py
class ModelLifecycleManager:
    ... def should_retrain(self, ticker: str, current_performance: Dict) -> bool:
        """Decide si re-entrenar basándose en métricas de degradación"""
        drift_detected = self.drift_detector.detect_drift(ticker)
        performance_degraded = current_performance['sharpe'] < self.baseline_sharpe * 0.8
        return drift_detected or performance_degraded

    ... def deploy_model(self, model: Any, ticker: str) -> ModelVersion:
        """Deploy con versionado automático"""
        version = self._create_version()
        self._run_safety_checks(model)
        self._deploy_with_canary(model, version)
        return version

```

Entregables:

- Auto-reentrenamiento trigger system
- Model versioning con metadata completo
- Canary deployment de modelos
- Drift detection real-time
- Model registry centralizado

FASE 3: Operaciones Robustas (2-3 semanas)

3.1 Live Trading Mejorado

Objetivos:

- State persistence para recovery
- Risk management multi-nivel
- Real-time monitoring

python

```

# modules/live_trading_v2.py
class TradingStateManager:
    ... def __init__(self):
        self.state_store = StateStore("redis://localhost:6379")

    ...
    ... def save_state(self, state: TradingState):
        """Persiste estado para recovery automático"""
        pass

    ...
    ... def recover_from_crash(self) -> TradingState:
        """Recovery automático post-crash"""
        pass

class MultiLevelRiskManager:
    ... def check_position_risk(self, position: Position) -> RiskCheck:
        """Risk check a nivel posición"""
        pass

    ...
    ... def check_portfolio_risk(self, portfolio: Portfolio) -> RiskCheck:
        """Risk check a nivel portfolio"""
        pass

```

Entregables:

- State persistence con Redis/SQLite
- Recovery automático post-crash
- Risk management jerárquico
- Circuit breakers automáticos
- Real-time position monitoring

3.2 Sistema de Alertas

Objetivos:

- Alertas de degradación de modelo
- Monitoring de slippage real vs esperado
- Detección de anomalías en mercado

python

```

# modules/alerting.py
class AlertingSystem:
    ... def __init__(self, channels=["slack", "email", "dashboard"]):
        self.channels = self._setup_channels(channels)

    ...
    ... def check_model_degradation(self, ticker: str):
        """Monitorea degradación de modelo"""
        pass

    ...
    ... def check_market_anomalies(self):
        """Detecta anomalías de mercado"""
        pass

```

Entregables:

- Sistema de alertas multi-canal
- Model degradation monitoring
- Slippage analysis real-time
- Market anomaly detection
- Performance threshold alerts

3.3 Backtesting Avanzado

Objetivos:

- Walk-forward analysis automatizado
- Monte Carlo simulation
- Sensitivity analysis

```

python

# modules/backtesting_v2.py
class AdvancedBacktester:
    ... def walk_forward_analysis(self, strategy: Strategy,
                                window_size: int = 252) -> WFAResults:
        """Walk-forward analysis automatizado"""
        pass

    ...
    ... def monte_carlo_simulation(self, strategy: Strategy,
                                 n_simulations: int = 1000) -> MCResults:
        """Monte Carlo simulation de estrategia"""
        pass

```

Entregables:

- Walk-forward analysis engine
- Monte Carlo simulation
- Sensitivity analysis automatizado
- Out-of-sample validation estricto
- Backtesting comparison framework

FASE 4: Productización (1-2 semanas)

4.1 Testing Comprehensivo

Estructura de testing:

```
tests/
├── unit/..... # Funciones individuales
│   ├── test_features.py
│   ├── test_labeling.py
│   └── test_metrics.py
├── integration/..... # Módulos completos
│   ├── test_data_pipeline.py
│   ├── test_training_pipeline.py
│   └── test_trading_pipeline.py
└── system/..... # End-to-end
    ├── test_full_backtest.py
    └── test_live_simulation.py
    └── performance/..... # Load testing
        ├── test_data_throughput.py
        └── test_model_latency.py
```

Entregables:

- Test suite comprehensivo (>80% coverage)
- CI/CD pipeline con GitHub Actions
- Performance benchmarks automatizados
- Integration testing con datos reales

4.2 Deployment & Monitoring

```
yaml
```

```
# docker-compose.yml
version: '3.8'
services:
  phibot-data:
    build: ./services/data
    environment:
      - REDIS_URL=redis://redis:6379

  phibot-training:
    build: ./services/training
    depends_on: [phibot-data]

  phibot-trading:
    build: ./services/trading
    depends_on: [phibot-training]

  redis:
    image: redis:alpine
```

Entregables:

- Containerización completa con Docker
- Docker Compose para desarrollo local
- Health checks automatizados
- Backup strategies implementadas
- Monitoring con Prometheus/Grafana

4.3 Dashboard Avanzado

Objetivos:

- Real-time P&L tracking
- Model performance metrics
- Risk exposure monitoring

```
python
```

```
# dashboard/advanced_dashboard.py
class AdvancedDashboard:
    ... def __init__(self):
        self.real_time_data = StreamingData()
        self.model_metrics = ModelMetrics()

    ...
    ... def render_pnl_dashboard(self):
        """Dashboard de P&L en tiempo real"""
        pass

    ...
    ... def render_risk_dashboard(self):
        """Dashboard de exposición al riesgo"""
        pass
```

Entregables:

- Real-time P&L dashboard
 - Model performance tracking
 - Risk exposure visualization
 - Trade attribution analysis
 - Mobile-responsive interface
-

Implementación Técnica

Patrones de Diseño Recomendados

Repository Pattern para Datos

```
python
```

```

from abc import ABC, abstractmethod

class DataRepository(ABC):
    @abstractmethod
    def get_ohlc(self, ticker: str, start: datetime, end: datetime) -> pd.DataFrame:
        pass

class ParquetRepository(DataRepository):
    def get_ohlc(self, ticker: str, start: datetime, end: datetime) -> pd.DataFrame:
        # Implementación específica para Parquet
        pass

class CSVRepository(DataRepository):
    def get_ohlc(self, ticker: str, start: datetime, end: datetime) -> pd.DataFrame:
        # Implementación específica para CSV
        pass

```

Factory Pattern para Modelos

```

python

class ModelFactory:
    @staticmethod
    def create_model(model_type: str, params: Dict) -> BaseModel:
        if model_type == "xgb":
            return XGBModel(params)
        elif model_type == "rf":
            return RandomForestModel(params)
        # ...

```

Observer Pattern para Alertas

```

python

class TradingEventObserver:
    def __init__(self):
        self.observers = []

    def notify_trade_executed(self, trade: Trade):
        for observer in self.observers:
            observer.on_trade_executed(trade)

```

Métricas de Calidad

Code Quality Gates

- **Test Coverage:** Mínimo 80%
- **Type Coverage:** Mínimo 90% (mypy)
- **Code Quality:** Score A en SonarQube
- **Documentation:** 100% docstrings en funciones públicas

Performance Benchmarks

- **Data Loading:** < 5 segundos para 1 año de datos 5min
 - **Model Training:** < 30 segundos por ticker
 - **Inference:** < 100ms por predicción
 - **Backtest:** < 60 segundos para 1 año completo
-

Cronograma de Implementación

Semana 1-2: Fundamentos

- Día 1-3: Sistema de configuración unificado
- Día 4-7: Logging estructurado completo
- Día 8-10: Data quality checks
- Día 11-14: Testing básico módulos críticos

Semana 3-5: Optimización Pipeline

- Día 15-21: Mejora módulo datos (DAT)
- Día 22-28: Research avanzado (RSH)
- Día 29-35: Training inteligente (TRN)

Semana 6-8: Operaciones Robustas

- Día 36-42: Live trading mejorado
- Día 43-49: Sistema de alertas
- Día 50-56: Backtesting avanzado

Semana 9-10: Productización

- Día 57-63: Testing comprehensivo
 - Día 64-70: Deployment & Dashboard final
-

Criterios de Éxito

Métricas Técnicas

- **Uptime:** >99.5% durante horas de mercado

- **Data Quality:** >95% cobertura temporal sin gaps
- **Model Performance:** Sharpe ratio >1.5 consistente
- **Latency:** <200ms desde señal hasta orden

Métricas de Negocio

- **Risk Management:** Max drawdown <5%
 - **Trade Execution:** <2% slippage promedio
 - **Operational:** <1 incident crítico por mes
 - **Monitoring:** 100% alertas funcionando
-

Recursos Necesarios

Herramientas Adicionales

- **Optuna:** Optimización bayesiana de hiperparámetros
- **Redis:** State persistence y caching
- **Prometheus/Grafana:** Monitoring y alertas
- **Docker:** Containerización
- **pytest:** Framework de testing

Conocimientos Requeridos

- Patrones de diseño de software
- Testing strategies (unit, integration, system)
- DevOps basics (Docker, CI/CD)
- Time series analysis avanzado
- Risk management cuantitativo

Timeline Realista

- **Tiempo total estimado:** 10 semanas
 - **Esfuerzo requerido:** 3-4 horas diarias promedio
 - **Hitos críticos:** Cada 2 semanas review y ajuste
 - **Buffer recomendado:** 20% tiempo adicional para imprevistos
-

Conclusión

Este plan transforma π-Bot de un prototipo funcional a un sistema de trading robusto listo para

producción. La implementación por fases permite mantener el sistema operativo mientras se mejora progresivamente cada componente.

El enfoque prioriza la estabilidad y observabilidad, elementos críticos para un sistema de trading automatizado. Al final de la implementación, tendrás un sistema que no solo opera de forma autónoma, sino que se automonitorea, se auto-mejora y proporciona transparencia completa de sus decisiones.

Próximo paso recomendado: Comenzar con FASE 1 - Fundamentos, específicamente con la unificación del sistema de configuración, ya que esto facilitará todas las mejoras posteriores.