🏗️ Arquitectura y Especificaciones Finales - Trading Bot Ecosystem v2.0

📋 Documento de Especificaciones Técnicas

Versión: 2.0
Fecha: Octubre 2025
Estado: ✅ Actualizado - Multi-Timeframe + Rhai Support
Changelog: Multi-timeframe strategies, Rhai scripts, semantic constraints, warmup strategy

---

🎯 Visión General del Proyecto

Descripción

Sistema modular de trading algorítmico escrito en Rust que permite generar, testear y ejecutar miles de estrategias de trading **multi-timeframe** de forma automatizada, con arquitectura cliente-servidor basada en gRPC.

Objetivos Principales

- ✅ Generar 10,000+ estrategias automáticamente usando algoritmos genéticos **con constrainst semánticos**
- ✅ **Multi-timeframe strategies**: Combinar indicadores de diferentes timeframes (ej: EMA 1h + RSI 5m)
- ✅ **Rhai scripts**: Creación manual de estrategias con sintaxis amigable
- ✅ Backtest masivo de estrategias en minutos (modo Polars) vs realista (event-driven)
- ✅ Arquitectura cliente-servidor escalable
- ✅ Interface gráfica nativa moderna y rápida
- ✅ **Warmup inteligente**: Datos históricos automáticos para indicadores que requieren largos períodos

Casos de Uso

Usuario	Workflow	Herramientas
Researcher/Quant	Desarrollo y validación masiva	Generador genético + Backtest Polars
Strategy Developer	Crear estrategias manuales	Rhai scripts + StrategyBuilder
Trader Algorítmico	Ejecución automatizada 24/7	Event-driven backtest + Live runner
Portfolio Manager	Gestión de múltiples estrategias	GUI client + Multi-timeframe analysis

---

Arquitectura del Sistema

Diagrama de Alto Nivel Actualizado

```
CAPA DE CLIENTE

  GUI Client              CLI Client
  (GTK4/Relm4)              (Clap)
  + Rhai Editor           + Script Runner

      gRPC (Tonic)
      Protocol Buffers

CAPA DE SERVIDOR

      API Server (gRPC Services)

   Strategy   Backtest   Optimizer   Live
   Service    Service    Service     Service

CAPA DE LÓGICA (Actualizada)

  Generator    ✨ Converter    Backtest
```

```
│  │ (Genetic +  │  │ (Hub)      │  │ (Dual Mode) │       │
│  │ Semantic)   │  │ AST→Formats │  │ Polars+Event │       │
│  └─────────────┘  └─────────────┘  └─────────────┘       │
│                        │                                 │
│  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐       │
│  │ Optimizer   │  │ Runner Live │  │ Data Manager │      │
│  │ (Grid/GA)   │  │ (Multi-TF + │  │ (Multi-TF +  │      │
│  │        │  │ Warmup)  │  │ Warmup)  │      │
│  └─────────────┘  └─────────────┘  └─────────────┘       │
│                        │                                 │
└──────────────────────────────────────────────────────────
          │
┌────────────────────────▼─────────────────────────────────┐
│        CAPA DE DATOS                │
├──────────────────────────────────────────────────────────
│                        │
│  ┌───────────────────────────────────────────────────────┐
│  │   ✨ Multi-Timeframe Strategy Store (PostgreSQL)   │  │
│  │ - Estrategias generadas (AST format)             │  │
│  │ - Estrategias manuales (Rhai format)             │  │
│  │ - Resultados de backtest                         │  │
│  │ - Similarity scores pre-computados               │  │
│  └───────────────────────────────────────────────────────┘
│
│                        │
│  ┌───────────────────────────────────────────────────────┐
│  │    ✨ Historical Data (Multi-Timeframe Parquet)   │  │
│  │ - Precios OHLCV por timeframe                    │  │
│  │ - Warmup data cache                              │  │
│  │ - Indicator correlation matrix                   │  │
│  └───────────────────────────────────────────────────────┘
│
│                        │
└──────────────────────────────────────────────────────────
```

📦 Estructura de Módulos (Crates) - Actualizada

Core Dependencies Actualizadas

```
┌── core (types & traits)
├── indicators (100% dinámico + metadata)
├── data (multi-timeframe context)
├── strategy-store (multi-format storage)
│
├── strategy-generator (genetic + semantic constraints)
├── ✨ strategy-converter (HUB central)
│   ├── inputs/ (rhai_parser, json_dsl, freqtrade)
│   ├── outputs/ (polars_query, event_driven, rhai_runtime)
│   └── similarity/ (correlation calculator)
│
├── backtest-engine (dual: polars + event-driven)
├── optimizer (grid/genetic/bayesian)
├── runner-live (multi-TF + warmup)
├── data-manager (multi-TF downloaders)
│
├── api-proto (gRPC definitions)
├── api-server (gRPC services)
├── api-client (gRPC client)
├── cli-client (terminal interface)
└── gui-client (GTK4 + Rhai editor)
```

🔧 Stack Tecnológico - Actualizaciones

Nuevas Adiciones

Componente	Tecnología	Versión	Justificación
Scripting	Rhai	1.19+	Estrategias manuales, sandboxed, hot reload
Correlación	nalgebra	0.32+	Cálculo de correlación Pearson entre indicadores
Multi-TF Sync	Custom	-	Sincronización de timeframes con forward-fill
Script Editor	GTK4 SourceView	5.0+	Syntax highlighting para Rhai

Mantenidos

- **Lenguaje**: Rust 2024 edition
- **Comunicación**: gRPC (Tonic) + Protocol Buffers

- **Data Processing**: Polars 0.41+ para backtest vectorizado
- **GUI**: GTK4/Relm4 para cliente nativo
- **Database**: SQLite (dev) → PostgreSQL (prod)

🧬 Especificación Multi-Timeframe

1. Concepto: Timeframe Relativo Simple

Filosofía: En lugar de timeframes absolutos complejos, usar **categorías semánticas**:

rust

```rust
pub enum TimeframeCategory {
    Current,  // Timeframe principal de la estrategia
    Medium,   // 3-5x el timeframe principal
    High,     // 12-24x el timeframe principal
}
```

2. Mapping Automático por Timeframe Principal

Principal	Current	Medium	High	Realismo
1m	1m	5m	1h	✅ Scalping + Context
5m	5m	15m	1h	✅ Day trading común
15m	15m	1h	4h	✅ Swing trading
1h	1h	4h	1d	✅ Position trading
4h	4h	1d	1w	✅ Long-term
1d	1d	1w	1M	✅ Investment

3. Lógica de Evaluación

rust

```rust
// Principio fundamental: Higher timeframes = vela cerrada anterior
// Current timeframe = vela actual en evaluación

impl MultiTimeframeStrategy {
    pub fn evaluate(&self, current_time: i64) -> Signal {
        for condition in &self.entry_conditions {
            let value = match condition.indicator.timeframe_category {
                TimeframeCategory::Current => {
                    // 🟢 Usar vela ACTUAL
                    self.get_current_indicator_value(&condition.indicator, current_time)
                }
                TimeframeCategory::Medium | TimeframeCategory::High => {
                    // 🔴 Usar última vela CERRADA de timeframe superior
                    self.get_last_closed_indicator_value(&condition.indicator, current_time)
                }
            };

            if !self.evaluate_condition(condition, value) {
                return Signal::Hold;
            }
        }

        Signal::Buy // Todas las condiciones cumplidas
    }
}
```

4. Ejemplo Práctico

Strategy Timeline (Primary = 5m):

```
14:00 ■■■■■■■■■■■■ 1h closed → EMA(200) = 42,150
14:05 ▓▓▓▓ 5m eval → RSI = 28.5, EMA_1h = 42,150 ✅ SIGNAL
14:10 ▓▓▓▓ 5m eval → RSI = 31.2, EMA_1h = 42,150 ❌ No signal
14:15 ▓▓▓▓ 5m eval → RSI = 29.8, EMA_1h = 42,150 ✅ SIGNAL
15:00 ■■■■■■■■■■■■ 1h closed → EMA(200) = 42,200 (updated)
```

📝 Especificación de Rhai Scripts

1. Sintaxis Propuesta

rust

```rust
// ==========================================
// Ejemplo: Golden Cross + RSI Filter
// ==========================================

// Configuración de timeframes
strategy_timeframe("5m");  // Timeframe principal

// Definición de indicadores con categorías
let ema_short = indicator("ema", [50], "current");  // EMA 50 en 5m
let ema_long = indicator("ema", [200], "medium");   // EMA 200 en 15m
let rsi = indicator("rsi", [14], "current");        // RSI 14 en 5m
let volume_avg = indicator("sma", [20], "high");    // Volume SMA en 1h

// Condiciones de entrada
entry_rules("and", [
    crosses_above(ema_short, ema_long),   // Golden cross
    rsi < 50.0,                           // RSI no sobbrecomprado
    volume() > volume_avg * 1.5           // Volumen alto
]);

// Condiciones de salida
exit_rules("or", [
    crosses_below(ema_short, ema_long),   // Death cross
    rsi > 70.0,                           // RSI sobrecomprado
    price() < ema_long * 0.98             // Stop loss 2%
]);

// Metadata opcional
set_name("Golden Cross RSI Filter");
set_description("EMA cross with RSI filter and volume confirmation");
```

2. Funciones Built-in

Función	Descripción	Ejemplo
strategy_timeframe(tf)	Define timeframe principal	strategy_timeframe("5m")
indicator(name, params, category)	Crea indicador	indicator("rsi", [14], "current")
entry_rules(op, conditions)	Define reglas entrada	entry_rules("and", [...])
exit_rules(op, conditions)	Define reglas salida	exit_rules("or", [...])
crosses_above(a, b)	A cruza por encima de B	crosses_above(ema_fast, ema_slow)
crosses_below(a, b)	A cruza por debajo de B	crosses_below(price(), sma)
price()	Precio actual (close)	price() > sma
volume()	Volumen actual	volume() > avg_volume
set_name(name)	Nombre de estrategia	set_name("My Strategy")

3. Validaciones Automáticas

rust

```rust
pub struct RhaiValidator;

impl RhaiValidator {
    pub fn validate_script(script: &str) -> Result<ValidationReport, RhaiError> {
        let mut report = ValidationReport::new();

        // Validar sintaxis Rhai
        let engine = rhai::Engine::new();
        engine.compile(script)?;

        // Validar semántica trading
        report.add_checks(vec![
            Self::check_timeframe_consistency(&script),
            Self::check_indicator_existence(&script),
            Self::check_logical_coherence(&script),
            Self::check_complexity_limits(&script),
        ]);

        Ok(report)
    }

    fn check_timeframe_consistency(script: &str) -> ValidationCheck {
        // Verificar que todos los timeframes sean válidos
        // current/medium/high deben mapear a timeframes reales
    }

    fn check_indicator_existence(script: &str) -> ValidationCheck {
        // Verificar que todos los indicadores existan en registry
    }

    fn check_logical_coherence(script: &str) -> ValidationCheck {
        // Evitar condiciones contradictorias: rsi > 70 AND rsi < 30
    }
}
```

🧠 Generador Masivo con Semantic Constraints

1. Constraints Dinámicos

rust

```rust
pub struct SemanticConstraints {
    // Límites por categoría (dinámico)
    pub max_per_category: HashMap<IndicatorCategory, usize>,

    // Límite de similaridad (correlación)
    pub max_similarity_score: f64,  // 0.0-1.0, default: 0.7

    // Límite de complejidad total
    pub max_complexity_score: f64,

    // Timeframes permitidos
    pub allowed_timeframe_categories: Vec<TimeframeCategory>,
}

impl Default for SemanticConstraints {
    fn default() -> Self {
        Self {
            max_per_category: hashmap! {
                IndicatorCategory::Trend => 2,
                IndicatorCategory::Momentum => 2,
                IndicatorCategory::Volume => 1,
                IndicatorCategory::Volatility => 1,
                IndicatorCategory::CandlestickPatterns => 1,
                IndicatorCategory::SupportResistance => 1,
                IndicatorCategory::Oscillators => 2,
            },
            max_similarity_score: 0.7,  // 70% max correlation
            max_complexity_score: 15.0,
            allowed_timeframe_categories: vec![
                TimeframeCategory::Current,
                TimeframeCategory::Medium,
                TimeframeCategory::High,
            ],
        }
    }
}
```

2. Similarity Calculator (Correlación Real)

rust

```rust
pub struct IndicatorSimilarityCalculator {
    // Cache de correlaciones pre-computadas
    correlation_matrix: HashMap<(String, String), f64>,
    reference_dataset: Vec<Candle>,
}

impl IndicatorSimilarityCalculator {
    pub fn calculate_similarity(&self, ind1: &str, ind2: &str) -> f64 {
        // Buscar en cache primero
        if let Some(cached) = self.get_cached_similarity(ind1, ind2) {
            return cached;
        }

        // Calcular correlación real usando datos históricos
        let values1 = self.calculate_indicator_series(ind1, &self.reference_dataset);
        let values2 = self.calculate_indicator_series(ind2, &self.reference_dataset);

        // Correlación de Pearson
        self.pearson_correlation(&values1, &values2).abs()
    }

    fn pearson_correlation(&self, x: &[f64], y: &[f64]) -> f64 {
        // Implementación estándar de correlación Pearson
        // Resultado: -1.0 a 1.0 (usamos abs() para 0.0 a 1.0)
    }

    pub fn build_correlation_matrix(&mut self) -> Result<(), Error> {
        let all_indicators = registry::all_names();

        for (i, ind1) in all_indicators.iter().enumerate() {
            for ind2 in all_indicators.iter().skip(i + 1) {
                let correlation = self.calculate_raw_correlation(ind1, ind2);
                self.correlation_matrix.insert((ind1.clone(), ind2.clone()), correlation);
            }
        }

        // Persistir cache en disco para futuras ejecuciones
        self.save_cache_to_disk()?;
        Ok(())
```

```
    }
}
```

3. Generador Inteligente

rust

```rust
impl RandomGenerator {
    pub fn generate_with_constraints(&self, constraints: &SemanticConstraints) -> StrategyAST {
        let mut selected_indicators = Vec::new();
        let mut category_count = HashMap::new();
        let mut complexity_score = 0.0;

        // Generate indicators respecting all constraints
        while selected_indicators.len() < self.max_indicators {
            let candidate = self.random_indicator_with_timeframe();
            let metadata = registry::get(&candidate.name).unwrap();

            // Check category limit
            let current_count = category_count.get(&metadata.category).unwrap_or(&0);
            let max_allowed = constraints.max_per_category.get(&metadata.category).unwrap_or(&999);

            if current_count >= max_allowed {
                continue;
            }

            // Check complexity limit
            if complexity_score + metadata.complexity > constraints.max_complexity_score {
                continue;
            }

            // Check similarity (most important)
            if self.is_too_similar(&candidate, &selected_indicators, constraints.max_similarity_score) {
                continue;
            }

            // Add indicator
            selected_indicators.push(candidate);
            *category_count.entry(metadata.category).or_insert(0) += 1;
            complexity_score += metadata.complexity;
        }

        self.build_strategy_from_indicators(selected_indicators)
    }

    fn random_indicator_with_timeframe(&self) -> IndicatorWithTimeframe {
```

```
        let base_indicator = self.random_indicator();
        let timeframe_category = self.random_timeframe_category();

        IndicatorWithTimeframe {
            indicator: base_indicator,
            timeframe_category,
        }
    }

    fn is_too_similar(&self, candidate: &IndicatorWithTimeframe, existing: &[IndicatorWithTimeframe], threshold: f64
        existing.iter().any(|existing_ind| {
            let similarity = self.similarity_calculator.calculate_similarity(
                &candidate.indicator.name,
                &existing_ind.indicator.name,
            );

            similarity > threshold
        })
    }
}
```

⚡ Backtest Engine Dual Mode

1. Arquitectura Dual

rust

```rust
pub enum BacktestMode {
    VectorizedMassive,  // Polars: 10,000+ strategies, fast approximation
    EventDrivenRealistic, // Event-by-event: 100 strategies, realistic simulation
}

pub struct DualBacktestEngine {
    polars_engine: PolarsBacktestEngine,
    event_engine: EventDrivenBacktestEngine,
    data_manager: MultiTimeframeDataManager,
}

impl DualBacktestEngine {
    pub async fn run_batch(&self, strategies: Vec<StrategyAST>, mode: BacktestMode) -> Vec<BacktestResult> {
        match mode {
            BacktestMode::VectorizedMassive => {
                // Convertir strategies a Polars queries
                let polars_queries: Vec<_> = strategies
                    .iter()
                    .map(|s| self.strategy_converter.to_polars_query(s))
                    .collect();

                // Ejecutar en batch paralelo
                self.polars_engine.run_batch_parallel(polars_queries).await
            }

            BacktestMode::EventDrivenRealistic => {
                // Convertir strategies a event-driven format
                let event_strategies: Vec<_> = strategies
                    .iter()
                    .map(|s| self.strategy_converter.to_event_driven(s))
                    .collect();

                // Ejecutar secuencialmente con simulación completa
                let mut results = Vec::new();
                for strategy in event_strategies {
                    let result = self.event_engine.run_single(strategy).await?;
                    results.push(result);
                }
                results
```

```
            }
        }
    }
}
```

2. Polars Vectorized Engine

rust

```rust
pub struct PolarsBacktestEngine {
    // Optimizado para throughput masivo
}

impl PolarsBacktestEngine {
    pub async fn run_batch_parallel(&self, queries: Vec<PolarsBatchQuery>) -> Vec<BacktestResult> {
        use rayon::prelude::*;

        // Paralelizar por chunks
        queries.par_chunks(100)
            .flat_map(|chunk| {
                // Cada chunk se ejecuta en Polars
                self.execute_polars_chunk(chunk)
            })
            .collect()
    }

    fn execute_polars_chunk(&self, queries: &[PolarsBatchQuery]) -> Vec<BacktestResult> {
        // Convertir estrategias a operations Polars
        // Ejecutar vectorizado con lazy evaluation
        // Retornar métricas básicas rápido
    }
}
```

3. Event-Driven Engine

rust

```rust
pub struct EventDrivenBacktestEngine {
    // Optimizado para realismo
    order_book: SimulatedOrderBook,
    slippage_model: SlippageModel,
    commission_model: CommissionModel,
}

impl EventDrivenBacktestEngine {
    pub async fn run_single(&self, strategy: EventDrivenStrategy) -> BacktestResult {
        let mut portfolio = Portfolio::new(10000.0);
        let mut trades = Vec::new();

        for candle in &self.data_manager.get_candles() {
            // Evaluar estrategia multi-timeframe
            let signal = strategy.evaluate_at_time(candle.timestamp, &self.data_manager);

            // Simular ejecución realista
            if let Some(order) = self.signal_to_order(signal, &portfolio) {
                let execution = self.order_book.simulate_execution(order, candle);
                let trade = self.apply_execution(execution, &mut portfolio);
                trades.push(trade);
            }
        }

        BacktestResult::from_trades(trades)
    }
}
```

💾 Warmup Strategy para Live Trading

1. Límites Realistas por Timeframe

rust

```rust
pub struct WarmupLimits {
    pub max_days_in_memory: u32,    // Máximo en RAM/streaming
    pub max_days_download: u32,     // Máximo descarga histórica
}

impl WarmupLimits {
    pub fn for_timeframe(tf: TimeFrame) -> Self {
        match tf {
            TimeFrame::M1 => Self { max_days_in_memory: 1, max_days_download: 1 },
            TimeFrame::M5 => Self { max_days_in_memory: 3, max_days_download: 3 },
            TimeFrame::M15 => Self { max_days_in_memory: 7, max_days_download: 14 },
            TimeFrame::M30 => Self { max_days_in_memory: 2, max_days_download: 30 },
            TimeFrame::H1 => Self { max_days_in_memory: 2, max_days_download: 60 },
            TimeFrame::H4 => Self { max_days_in_memory: 2, max_days_download: 240 },
            TimeFrame::D1 => Self { max_days_in_memory: 2, max_days_download: 365 },
        }
    }
}
```

2. Warmup Decision Logic

rust

```rust
pub enum WarmupAction {
    StreamingOnly,       // Para periods cortos (< 2 días)
    HistoricalDownload,  // Para periods medios (2-365 días)
    StrategyNotViable,   // Para periods largos (> 365 días)
}

impl LiveDataManager {
    pub async fn plan_warmup(&self, strategy: &StrategyAST) -> Result<WarmupPlan, Error> {
        let required_timeframes = strategy.required_timeframes();
        let mut plan = WarmupPlan::new();

        for tf in required_timeframes {
            let max_period = self.calculate_max_indicator_period(strategy, tf);
            let required_days = self.periods_to_days(max_period, tf);
            let limits = WarmupLimits::for_timeframe(tf);

            let action = if required_days <= limits.max_days_in_memory {
                WarmupAction::StreamingOnly
            } else if required_days <= limits.max_days_download {
                WarmupAction::HistoricalDownload
            } else {
                WarmupAction::StrategyNotViable
            };

            plan.add_timeframe(tf, action, required_days);
        }

        Ok(plan)
    }

    fn calculate_max_indicator_period(&self, strategy: &StrategyAST, tf: TimeFrame) -> usize {
        let mut max_period = 0;

        // Buscar en todas las condiciones
        for condition in strategy.all_conditions() {
            if condition.indicator.timeframe_matches(tf) {
                let meta = registry::get(&condition.indicator.name).unwrap();

                // Calcular period basado en parámetros
```

```
                // ej: EMA(200) = 200 periods, SMA(50) = 50 periods
                let period = self.extract_period_from_params(&condition.indicator, meta);
                max_period = max_period.max(period);
            }
        }

        // Factor de seguridad para warmup (1.5x)
        (max_period as f64 * 1.5) as usize
    }
}
```

3. Ejemplos de Warmup

Estrategia	Indicador Crítico	Timeframe	Períodos	Días	Acción
Scalping 1m	RSI(14) 1m	1m	14	0.01	✅ Streaming
Day Trading	EMA(50) 5m + RSI(14) 5m	5m	50	0.17	✅ Streaming
Swing	EMA(200) 1h + SMA(20) 4h	1h	200	8.3	✅ Download
Position	EMA(200) 1d	1d	200	200	✅ Download
Long-term	EMA(500) 1d	1d	500	500	❌ Not viable

🌐 Especificación gRPC Services - Actualizadas

1. StrategyService (Extendido)

protobuf

```protobuf
service StrategyService {
  // CRUD básico
  rpc List(ListRequest) returns (ListResponse);
  rpc Get(GetRequest) returns (Strategy);
  rpc Delete(DeleteRequest) returns (google.protobuf.Empty);

  // ✨ Nuevas operaciones
  rpc ParseRhai(ParseRhaiRequest) returns (ParseRhaiResponse);
  rpc ValidateStrategy(ValidateRequest) returns (ValidationResponse);
  rpc ConvertFormat(ConvertRequest) returns (ConvertResponse);

  // Operaciones pesadas (streaming)
  rpc Generate(GenerateRequest) returns (stream ProgressUpdate);
  rpc Export(ExportRequest) returns (stream ExportChunk);
}

// ✨ Nuevos mensajes
message ParseRhaiRequest {
  string rhai_script = 1;
  string target_timeframe = 2;  // "5m", "1h", etc.
}

message ParseRhaiResponse {
  oneof result {
    StrategyAST strategy_ast = 1;
    ValidationError error = 2;
  }
}

message ConvertRequest {
  StrategyAST strategy = 1;
  ConvertFormat target_format = 2;
}

enum ConvertFormat {
  POLARS_QUERY = 0;
  EVENT_DRIVEN = 1;
  RHAI_SCRIPT = 2;
```

```
    FREQTRADE = 3;
}
```

2. BacktestService (Dual Mode)

protobuf

```protobuf
service BacktestService {
  // ✨ Modo dual
  rpc RunMassive(MassiveBacktestRequest) returns (stream BacktestProgress);
  rpc RunRealistic(RealisticBacktestRequest) returns (stream BacktestProgress);

  // Multi-timeframe support
  rpc RunMultiTimeframe(MultiTimeframeBacktestRequest) returns (stream BacktestProgress);

  // Warmup planning
  rpc PlanWarmup(WarmupPlanRequest) returns (WarmupPlanResponse);
}

message MassiveBacktestRequest {
  repeated int64 strategy_ids = 1;
  string dataset = 2;
  TimeRange time_range = 3;
  PolarsConfig polars_config = 4;  // Paralelización, chunks
}

message RealisticBacktestRequest {
  repeated int64 strategy_ids = 1;  // Max 100 strategies
  string dataset = 2;
  TimeRange time_range = 3;
  SimulationConfig simulation = 4;  // Slippage, commission, etc.
}

message WarmupPlanRequest {
  StrategyAST strategy = 1;
  repeated string required_timeframes = 2;
}

message WarmupPlanResponse {
  repeated WarmupAction actions = 1;
  bool is_viable = 2;
  string viability_reason = 3;
}

message WarmupAction {
  string timeframe = 1;
```

```
  WarmupType type = 2;
  uint32 required_days = 3;
}

enum WarmupType {
  STREAMING_ONLY = 0;
  HISTORICAL_DOWNLOAD = 1;
  NOT_VIABLE = 2;
}
```

📊 Performance Benchmarks - Actualizados

1. Multi-Timeframe Performance

Operación	Input	Target	Notas
Rhai parsing	100 líneas script	< 50ms	Sintaxis validation
Strategy conversion	AST → Polars	< 100ms	Multi-TF compilation
Massive backtest	10,000 strategies	< 60 min	Polars vectorized
Realistic backtest	100 strategies	< 30 min	Event-driven simulation
Warmup planning	Multi-TF strategy	< 5ms	Decision logic
Historical download	1 year 1h data	< 2 min	Exchange API limits
Correlation calculation	100x100 indicators	< 10 min	One-time precompute

2. Memory Usage

Timeframe	Max Period	Data Size	Memory	Action
1m	RSI(14)	14 candles	< 1KB	✅ Stream
5m	EMA(200)	200 candles	< 10KB	✅ Stream
1h	EMA(200)	200 candles	< 10KB	✅ Stream
1h	EMA(500)	500 candles	< 25KB	✅ Stream
1d	EMA(200)	200 candles	< 10KB	✅ Stream
1d	EMA(500)	500 candles	< 25KB	✅ Download

🧪 Testing Strategy - Multi-Timeframe

1. Unit Tests Extendidos

rust

```rust
#[cfg(test)]
mod multi_timeframe_tests {
    use super::*;

    #[test]
    fn test_rhai_parsing_multi_timeframe() {
        let script = r#"
            strategy_timeframe("5m");
            let ema = indicator("ema", [200], "medium");  // Should map to 15m
            entry_rules("and", [ema > price()]);
        "#;

        let result = RhaiParser::parse(script).unwrap();
        assert_eq!(result.primary_timeframe, TimeFrame::M5);

        let ema_condition = &result.entry_rules.conditions[0];
        assert_eq!(ema_condition.indicator.timeframe_category, TimeframeCategory::Medium);
    }

    #[test]
    fn test_semantic_constraints() {
        let constraints = SemanticConstraints::default();
        let generator = RandomGenerator::with_constraints(constraints);

        let strategy = generator.generate("Test".to_string());

        // Verificar que no hay indicadores muy correlacionados
        let indicators: Vec<_> = strategy.all_indicators().collect();
        for (i, ind1) in indicators.iter().enumerate() {
            for ind2 in indicators.iter().skip(i + 1) {
                let similarity = SIMILARITY_CALCULATOR.calculate_similarity(&ind1.name, &ind2.name);
                assert!(similarity < 0.7, "Indicators {} and {} too similar: {}", ind1.name, ind2.name, similarity);
            }
        }
    }

    #[test]
    fn test_warmup_planning() {
        let strategy = create_test_multi_tf_strategy(); // EMA(200) 1h + RSI(14) 5m
```

```rust
        let plan = LiveDataManager::plan_warmup(&strategy).unwrap();

        assert_eq!(plan.actions.len(), 2);
        assert_eq!(plan.actions[0].timeframe, TimeFrame::M5);
        assert_eq!(plan.actions[0].action, WarmupAction::StreamingOnly);
        assert_eq!(plan.actions[1].timeframe, TimeFrame::H1);
        assert_eq!(plan.actions[1].action, WarmupAction::HistoricalDownload);
    }
}
```

2. Integration Tests

rust

```rust
#[tokio::test]
async fn test_full_multi_timeframe_workflow() {
    // 1. Create Rhai strategy
    let rhai_script = create_sample_multi_tf_script();

    // 2. Parse to AST
    let strategy_ast = RhaiParser::parse(&rhai_script).unwrap();

    // 3. Store in database
    let strategy_id = StrategyRepo::save(strategy_ast, "rhai").await.unwrap();

    // 4. Plan warmup
    let warmup_plan = LiveDataManager::plan_warmup(&strategy_ast).await.unwrap();
    assert!(warmup_plan.is_viable);

    // 5. Convert to backtest format
    let polars_query = StrategyConverter::to_polars_query(&strategy_ast).unwrap();

    // 6. Run backtest
    let result = DualBacktestEngine::run_single(polars_query, BacktestMode::VectorizedMassive).await.unwrap();

    // 7. Verify results
    assert!(result.total_trades > 0);
    assert!(result.sharpe_ratio.is_finite());
}
```

✅ Roadmap Actualizado

Fase 1: Multi-Timeframe Foundation (Semanas 1-2)

- ✅ **Completar** `data/multi_timeframe.rs` con sincronización
- ✅ **Extender** `StrategyAST` para soportar timeframe categories
- ✅ **Implementar** `TimeframeCategory` mapping logic
- ✅ **Testing** multi-timeframe data loading

Fase 2: Rhai Integration (Semanas 3-4)

- ✅ **Crear** `strategy-converter/inputs/rhai_parser.rs`
- ✅ **Implementar** Rhai → StrategyAST conversion
- ✅ **Validación** semántica de scripts Rhai
- ✅ **Testing** Rhai parsing y validation

Fase 3: Semantic Constraints (Semanas 5-6)

- ✅ **Calcular** correlation matrix de indicadores
- ✅ **Implementar** `SemanticConstraints` en generator
- ✅ **Testing** constraint enforcement
- ✅ **Optimizar** performance de similarity calculation

Fase 4: Backtest Engine Dual Mode (Semanas 7-10)

- ✅ **Implementar** Polars vectorized engine
- ✅ **Implementar** Event-driven engine
- ✅ **Strategy converter** outputs (AST → executable formats)
- ✅ **Testing** ambos modos de backtest

Fase 5: Warmup Strategy (Semanas 11-12)

- ✅ **Implementar** warmup planning logic
- ✅ **Historical data** downloader para exchanges
- ✅ **Cache management** para datos históricos
- ✅ **Testing** warmup scenarios

Fase 6: API Layer (Semanas 13-14)

- ✅ **Extender** gRPC services con nuevas operaciones
- ✅ **Implementar** server-side handlers
- ✅ **Client integration** para multi-timeframe
- ✅ **Testing** end-to-end workflows

Fase 7: GUI Updates (Semanas 15-16)

- ✅ **Rhai editor** con syntax highlighting
- ✅ **Multi-timeframe** visualization
- ✅ **Strategy builder** visual
- ✅ **Testing** user workflows

📈 Success Metrics - Actualizados

Funcionalidad

Métrica	Target	Status
Rhai scripts parseable	100% valid syntax	🚧 In progress
Multi-TF strategies generated	10,000+ combinations	🚧 In progress
Correlation accuracy	< 5% error vs real data	🚧 In progress
Warmup success rate	> 95% strategies viable	🚧 In progress
Backtest mode coverage	100% strategies both modes	🚧 In progress

Performance

Métrica	Target	Status
Rhai parsing time	< 50ms per script	🚧 In progress
Massive backtest	10,000 strategies < 60min	🚧 In progress
Realistic backtest	100 strategies < 30min	🚧 In progress
Memory usage	< 2GB for full system	🚧 In progress

📚 Documentation Structure - Actualizada

```
docs/
├── README.md                    # Overview + quick start
├── architecture/
│   ├── multi-timeframe.md         # ✨ Multi-TF strategy design
│   ├── rhai-integration.md        # ✨ Rhai scripting guide
│   ├── semantic-constraints.md    # ✨ Constraint system
│   ├── backtest-dual-mode.md      # ✨ Polars vs Event-driven
│   └── warmup-strategy.md         # ✨ Live trading warmup
├── api/
│   ├── grpc-reference.md          # gRPC service reference
│   ├── rhai-api.md                # ✨ Rhai built-in functions
│   └── strategy-ast.md            # ✨ AST format specification
├── guides/
│   ├── creating-rhai-strategies.md  # ✨ Rhai scripting tutorial
│   ├── multi-timeframe-strategies.md # ✨ Multi-TF best practices
│   ├── backtesting-modes.md       # ✨ When to use each mode
│   └── live-trading-setup.md      # ✨ Live trading configuration
└── examples/
    ├── rhai-strategies/           # ✨ Rhai script examples
    ├── multi-timeframe-examples/   # ✨ Multi-TF strategy examples
    └── backtest-comparisons/      # ✨ Polars vs Event comparisons
```

🎯 Conclusión

Esta especificación v2.0 integra completamente:

✅ Multi-Timeframe Support

- Timeframe categories semánticas (current/medium/high)

- Evaluación realista (higher TF = vela cerrada)
- Mapping automático por timeframe principal

✅ Rhai Scripting

- Sintaxis amigable para estrategias manuales
- Validación semántica automática
- Conversión a StrategyAST unificado

✅ Semantic Constraints

- Generador masivo inteligente
- Anti-correlación basada en datos reales
- Escalable a 100+ indicadores

✅ Backtest Dual Mode

- Polars vectorizado: 10,000 strategies masivo
- Event-driven: 100 strategies realista
- Strategy converter como hub central

✅ Warmup Strategy

- Límites realistas por timeframe
- Download automático vs streaming
- Validación de viabilidad de estrategias

La arquitectura mantiene la modularidad original pero agrega las capacidades avanzadas necesarias para un sistema de trading profesional multi-timeframe.

Próximo paso: Implementación fase por fase siguiendo el roadmap actualizado.

Documento preparado por: Trading Bot Team
Última actualización: Octubre 2025
Estado: ✅ **Aprobado para Desarrollo v2.0**