

AI-Powered Trading Bot: Enterprise Architecture Documentation

Team 19 | December 2025 **Project Team:** Muhammed İkbāl Ceyhan, Yunus Emre Balcı, Nisa Nur Kaya

1. Introduction

1.1 Project Overview

The **AI-Powered Trading Bot** is a sophisticated, modular algorithmic trading system designed to demonstrate the practical application of enterprise-grade **Design Patterns** in a high-concurrency financial environment.

Unlike simple script-based bots, this application uses a loosely coupled architecture that allows for:

- **Hot-swapping trading strategies** without restarting the application.
- **Dynamic risk management** layers that can be wrapped around any strategy.
- **Real-time event-driven processing** of market data.
- **Simulation and Live modes** for safe testing and execution.

1.2 System Requirements

1.2.1 Functional Requirements

- **Real-time Data Stream:** The system must fetch price updates at least once per second (1Hz).
- **Dynamic Configuration:** Users must be able to change active strategies via a web dashboard.
- **Risk Management:** The system must automatically trigger a "Panic Sell" if prices drop >5% in 10 seconds.
- **Visual Dashboard:** A web interface must display live prices, active strategy, and portfolio balance.

1.2.2 Non-Functional Requirements

- **Latency:** Signal generation processing time must be under 50ms.
- **Reliability:** The system must handle API failures gracefully.
- **Extensibility:** New strategies must be addable without modifying the Bot class.

1.3 Technology Stack

Component	Technology	Version	Description
Backend Language	Java	21 (LTS)	Core business logic and multi-threading.
Frontend Framework	Angular	21.0.0	Single Page Application (SPA).
Market Data API	Binance API	v3	Crypto market data source.

Component	Technology	Version	Description
HTTP Server	com.sun.net.httpserver	Native	Lightweight REST API server.
Build Tool	Maven	3.8+	Dependency management.

2. System Architecture

The system operates on an **Event-Driven Layered Architecture**. The core logic is decoupled from data sources and execution mechanisms, ensuring high testability and flexibility.

2.2 Layer Descriptions

2.2.1 Service Layer

Responsible for raw I/O operations. `BinanceService` communicates with the exchange, while `ApiService` handles dashboard requests.

2.2.2 Core Layer

Contains the central coordination logic. The `Bot` class acts as the orchestrator, receiving data from `PriceSubject`.

2.2.3 Logic Layer

Pure functional logic that transforms `Candle` data into `Signal` (BUY/SELL). Contains `TradingStrategies` and `StrategyDecorator`.

2.2.4 Execution Layer

Encapsulates side-effects such as placing orders, updating the wallet, and logging to CSV via `OrderCommand`.

2.2.5 Presentation Layer

The Angular frontend that visualizes the internal state for the human operator.

2.2.6 Data Models

Key data structures used across layers:

- **Candle:** `open`, `high`, `low`, `close`, `volume`, `timestamp`.
- **Order:** `symbol`, `side` (BUY/SELL), `quantity`.
- **Signal:** Enum used for trading decisions.
 - `STRONG_BUY`: 98% Balance "All In"
 - `BUY`: 40% Balance standard entry
 - `HOLD`: No action
 - `SELL`: 50% Position exit
 - `STRONG_SELL`: 100% Position exit (Panic Sell)
- **WalletBalance:** Holds USDT and BTC amounts from Binance.

3. Design Patterns Implementation

This project implements six fundamental GoF (Gang of Four) design patterns. Below is a detailed breakdown of why each pattern was chosen (Challenge), how it solves the problem (Solution), and the specific implementation in this project.

3.1 Strategy Pattern

Challenge: Hardcoding trading logic (e.g., "if RSI > 70 then sell") directly into the Bot class makes the system rigid. We need to support multiple trading algorithms (RSI, MACD, SMA) and be able to switch between them at runtime without stopping the bot or modifying the core code.

Solution: The **Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows the trading logic to vary independently from the Bot that uses it.

Implementation in Project:

- **Interface:** `interfaces.TradingStrategy` - Defines the common contract (`generateSignal`).
- **Context:** `Bot.StrategySelector` & `Bot.BotConfig` - The selector dynamically chooses the best strategy based on market conditions (ADX).
 - **Cooldown:** Prevents rapid strategy switching (5 candles).
 - **Hysteresis:** ADX < 20 (Reversion), ADX > 25 (Trend), else Risk Management.
- **Concrete Strategy:**
 - `TradingStrategies.SmaCrossover`: Simple Moving Average Crossover (Golden/Death Cross).
 - `TradingStrategies.RsiStrategy`: Dynamic thresholds based on trend (Uptrend: Buy < 45, Sell > 80).
 - `TradingStrategies.MacdStrategy`: Histogram reversal and signal line crossovers.
 - `TradingStrategies.AdxStrategy`: Trend strength analysis.
 - `TradingStrategies.TrendFollowing`: Simple price action comparison.
 - `TradingStrategies.DefaultStrategy`: ATR-based risk management (%8 drop protection).

Code Showcase:

```
// 1. The Strategy Interface
public interface TradingStrategy {
    Signal generateSignal(List<Candle> candles); // Uses Candle history
    String getName();
}

// 2. A Concrete Strategy (SMA Crossover)
public class SmaCrossover implements TradingStrategy {
    // ... fields for periods (e.g. 50, 200) ...
    @Override
    public Signal generateSignal(List<Candle> candles) {
        if (candles.size() < longPeriod) return Signal.HOLD;

        // Calculate SMA for current and previous candles
        double shortSmaCurrent = calculateSma(candles, shortPeriod, candles.size()
- 1);
        double longSmaCurrent = calculateSma(candles, longPeriod, candles.size() -
```

```

1);
    double shortSmaPrev = calculateSma(candles, shortPeriod, candles.size() -
2);
    double longSmaPrev = calculateSma(candles, longPeriod, candles.size() -
2);

    // Golden Cross (Short crosses above Long)
    if (shortSmaPrev <= longSmaPrev && shortSmaCurrent > longSmaCurrent) {
        return Signal.BUY;
    }

    // Death Cross (Short crosses below Long)
    if (shortSmaPrev >= longSmaPrev && shortSmaCurrent < longSmaCurrent) {
        return Signal.SELL;
    }

    return Signal.HOLD;
}
}

```

3.2 Observer Pattern

Challenge: The bot operates in a real-time market. Polling the API (constantly asking "Is there a new price?") is inefficient and introduces unnecessary latency. The Bot needs to be notified immediately when a new price candle closes.

Solution: The **Observer Pattern** creates a subscription mechanism. The Data Source (Subject) automatically notifies all registered Listeners (Observers) whenever its state changes.

Implementation in Project:

- **Subject Interface:** `interfaces.Subject` - Defines `register(Observer)`, `remove(Observer)`, `notifyObservers()`.
- **Observer Interface:** `interfaces.Observer` - Defines `priceUpdated(Candle)`.
- **Subject Implementation:** `PriceObservers.PriceSubject` - Manages the list of observers and notifies them with the new `Candle`.
- **Observer Implementation:** `PriceObservers.PriceListener` - Waits for updates and triggers the bot.

Code Showcase:

```

// 1. The Subject (Broadcaster)
public class PriceSubject implements Subject {
    List<Observer> observers = new ArrayList<>();
    models.Candle candle; // State

    @Override
    public void notifyObservers() {
        // Pushes the update to all subscribers
        for(int i = 0; i < observers.size(); i++){

```

```

        observers.get(i).priceUpdated(candle);
    }
}

public void setPrice(models.Candle candle){
    this.candle = candle;
    notifyObservers();
}
}

// 2. The Observer (Listener)
public class PriceListener implements Observer {
    Bot bot = new Bot();
    @Override
    public void priceUpdated(Candle candle) {
        // Reacts immediately to the new data
        bot.trade(candle);
    }
}

```

3.3 Decorator Pattern

Challenge: We need to apply global risk management rules, such as "Crash Protection" (panic sell if price drops significantly), to *any* active strategy. We don't want to rewrite the logic inside every single strategy class (RSI, SMA, MACD, etc.).

Solution: The **Decorator Pattern** allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class. We "wrap" the strategy in a protective layer that intercepts the signal.

Implementation in Project:

- **Abstract Decorator:** `StrategyDecorator.StrategyDecorator` - Implements `TradingStrategy`.
- **Concrete Decorator:**
 - `StrategyDecorator.CrashProtection`: Panic sell if price drops > X% (default 2%) in 5 candles. Includes RSI check (don't sell if already oversold < 25).
 - `StrategyDecorator.HighRisk`: Aggressive logic. Buys on Bullish Divergence or Breakout with Volume. Sells on Pump (>1.5% vs EMA20).
 - `StrategyDecorator.LowRisk`: Conservative logic. Buys only if Trend is UP + Volume + MACD positive. Sells on support break.

Code Showcase:

```

// 1. The Concrete Decorator
public class CrashProtection extends StrategyDecorator {
    public CrashProtection(TradingStrategy strategy, double dropThreshold) {
        super(strategy); // Wraps the original strategy
    }

    @Override

```

```

public Signal generateSignal(List<Candle> candles) {
    // 1. Check for crash condition (Price Drop)
    Candle current = candles.get(candles.size() - 1);
    Candle old = candles.get(candles.size() - lookback);
    double change = (current.close - old.close) / old.close;

    if (change < -dropThreshold) {
        // Smart Filter: Don't sell if already Oversold (RSI < 25)
        double rsi = calculateRSI(candles, 14);
        if (rsi < 25) {
            return Signal.HOLD; // Market might bounce back
        }
        return Signal.STRONG_SELL; // Override strategy -> Panic Sell!
    }

    // 2. If safe, delegate to the original strategy
    return wrappedStrategy.generateSignal(candles);
}
}

```

3.4 Command Pattern

Challenge: The decision to trade (Logic) should be separated from the actual execution of the trade (API calls, Wallet updates). We also need a way to support "Simulation" vs "Live" execution modes transparently.

Solution: The **Command Pattern** encapsulates a request as an object. This allows us to parameterize the execution and decouple the invoker (Bot) from the receiver (OrderReceiver).

Implementation in Project:

- **Command Interface:** `commands.OrderCommand` - Defines `execute()` and `undo()`.
- **Concrete Command:**
 - `commands.BuyCommand`: Encapsulates buy logic.
 - `commands.SellCommand`: Encapsulates sell logic, symmetrical to BuyCommand.
- **Receiver:** `commands.OrderReceiver` - Handles actual execution (Simulation vs Real).
- **Additional:** The `OrderCommand` interface includes an `undo()` method for potential rollback.

Code Showcase:

```

// 1. The Command Object
public class BuyCommand implements OrderCommand {
    private OrderReceiver receiver;
    private Order order;
    private double currentPrice;

    public BuyCommand(OrderReceiver receiver, Order order, double currentPrice) {
        this.receiver = receiver;
        this.order = order;
        this.currentPrice = currentPrice;
    }
}

```

```

    @Override
    public void execute() {
        // The command passes all necessary context to the receiver
        receiver.placeBuyOrder(order, currentPrice);
    }
    @Override
    public void undo() {
        System.out.println("Undo Buy Command not implemented");
    }
}

// 2. The Receiver (Business Logic)
public class OrderReceiver {
    public void placeBuyOrder(Order order, double currentPrice) {
        if (!BinanceConfig.isConfigured()) {
            // Simulation Logic
            wallet.withdrawUsdt(order.quantity * currentPrice);
        } else {
            // Real Execution Logic
            service.placeOrder(order.symbol, "BUY", order.quantity);
        }
    }
}

```

3.5 Template Method Pattern

Challenge: The lifecycle of a trade tick is strict and must always follow the same order: **Fetch Data** -> **Analyze** -> **Create Order** -> **Execute** -> **Log**. However, the implementation of "Analyze" or "Fetch" might differ. We want to enforce this sequence while allowing flexibility in the steps.

Solution: The **Template Method Pattern** defines the skeleton of an algorithm in a base class, letting subclasses override specific steps without changing the overall structure.

Implementation in Project:

- **Abstract Base Class:** `interfaces.TradingTemplate`
- **Template Method:** `trade(Candle candle)` - Defined as `final` to prevent modification.
- **Concrete Implementation:** `Bot.Bot` extends `TradingTemplate`.

Code Showcase:

```

public abstract class TradingTemplate {

    // The "Template Method" - declared final so it cannot be changed
    public final void trade(Candle candle) {
        List<Candle> candles = fetchData(candle);
        Signal signal = evaluateData(candles); // Specific logic happens here
        Order order = createOrder(signal);
        executeOrder(order);
        logResult(order);
    }
}

```

```
// Abstract methods to be implemented by subclasses
protected abstract Signal evaluateData(List<Candle> candles);
protected abstract void executeOrder(Order order);
protected abstract void logResult(Order order);
}
```

3.6 Singleton Pattern

Challenge: Components like the `Wallet` (which holds money) and `BotConfig` (which holds global settings) must have exactly one instance. Multiple instances could lead to conflicting states (e.g., one wallet saying you have \$100 and another saying \$50).

Solution: The **Singleton Pattern** ensures a class has only one instance and provides a global point of access to it.

Implementation in Project:

- **Singleton Class:** `Bot.BotConfig`
- **Singleton Class:** `models.Wallet`

Code Showcase:

```
public class BotConfig {
    // 1. Private static instance
    private static BotConfig uniqueInstance;
    public TradingStrategy strategy; // Active Strategy

    // 2. Private constructor prevents instantiation
    private BotConfig() {}

    // 3. Global access point
    public static synchronized BotConfig getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new BotConfig();
        }
        return uniqueInstance;
    }
}
```

3.7 Lock Service (Concurrency)

Implementation: `services.LockService` provides a centralized `fileLock` object used in `synchronized` blocks to ensure thread-safe CSV writing across multiple threads.

4. Backend Implementation Details

4.1 Application Entry Point (Program.java)

The application initializes as follows:

1. **Bot Config:** Loads `CrashProtection` wrapping `SmaCrossover` as the initial strategy.
2. **Observer Setup:** Registers `PriceListener` to `PriceSubject`.
3. **Services:** Starts `ApiService` (port 8081) and `BinanceService`.
4. **Main Loop:** Fetches data every 15m (15-minute candles) and pushes updates to the Subject.

4.2 Bot Logic (Bot.java)

- **Quantity Logic:**
 - `STRONG_BUY`: Uses 98% of USDT balance.
 - `BUY`: Uses 40% of USDT.
 - `STRONG_SELL`: Sells 100% of BTC.
 - `SELL`: Sells 50% of BTC.
- **Minimum Order:** Skips orders where value < 5 USDT.
- **Balance Cache:** Caches balance to reduce API calls, updating only on trade execution.

4.3 Binance Service & Security

The `BinanceService` handles authenticated requests using HMAC-SHA256 signatures, manages time synchronization, and handles precision rounding.

4.3.1 Key Features

- **Testnet Support:** Configurable via `.env` or defaults to Binance Testnet.
- **Time Sync:** Calculates server time offset to prevent "Timestamp for this request is outside of the recvWindow" errors.
- **Step Size:** Automatically rounds quantity to the valid symbol step size (e.g., 0.00001 BTC).

4.3.2 HMAC Signature Generation

```
private String hmacSha256(String data, String secret) {
    Mac sha256_HMAC = Mac.getInstance("HmacSHA256");
    SecretKeySpec secret_key = new SecretKeySpec(secret.getBytes(), "HmacSHA256");
    sha256_HMAC.init(secret_key);
    return Hex.encodeHexString(sha256_HMAC.doFinal(data.getBytes()));
}
```

4.4 API Service

Handles HTTP requests for the dashboard with CORS support enabled for all origins. Uses `LockService` for thread-safe reading of the trade history CSV.

4.4.1 Request Handling

```
static class StrategyHandler implements HttpHandler {
    public void handle(HttpExchange t) {
```

```
        if (POST) {
            // Parse JSON body and update Strategy
            BotConfig.getInstance().strategy = newStrategy;
        }
    }
}
```

5. Frontend Implementation

5.1 Angular Architecture

The frontend is built with **Angular 21** using Standalone Components and Signals.

- **Configuration:** Users can select strategies ([SmaCrossover](#), [TrendFollowing](#), [RSI](#), [MACD](#), [ADX](#), [Default](#)) and decorators ([None](#), [CrashProtection](#), [HighRisk](#), [LowRisk](#)).

```
@Injectable({ providedIn: 'root' })
export class TradeService {
    getTrades(): Observable<Trade[]> {
        return this.http.get<Trade[]>(`${this.baseUrl}/trades`);
    }
}
```

5.2 Real-time Dashboard

The dashboard uses **Chart.js** to render live price data, updating dynamically as new price points arrive from the backend API.

6. API Documentation

6.1 REST Endpoints

6.1.1 Trade History API

Method	Endpoint	Description
GET	/api/trades	Returns list of executed trades.

6.1.2 Strategy Management API

Method	Endpoint	Description
GET	/api/strategy	Returns current active strategy name.
POST	/api/strategy	Switches the active strategy.

6.2 Request/Response Examples

POST /api/strategy

```
{
  "strategy": "RsiStrategy",
  "decorator": "CrashProtection"
}
```

7. Conclusion

7.1 Achievements

This project successfully demonstrates that enterprise-grade software architecture principles can be applied to algorithmic trading. The system is robust, flexible, and maintainable.

7.2 Future Enhancements

- **Machine Learning Integration:** Python-Bridge for AI models.
- **Database Migration:** Moving from CSV to PostgreSQL.
- **User Authentication:** JWT auth for Dashboard.

References

1. Gamma, E., et al. (1994). *Design Patterns*. Addison-Wesley.
2. Binance API Docs. (2025).