Ai Engineered Prompt for Developing an Autotrading Algorithm in Ninjascript

A comprehensive AI prompt designed to guide the development of a robust NinjaScript automated trading strategy for NinjaTrader Desktop, incorporating order flow, market depth, and best practices for stability and debugging, based on the provided documents.

AI Prompt: NinjaScript Order Flow & Market Depth Autotrading Algorithm Project

Objective: Develop a robust and well-architected NinjaScript automated trading strategy for NinjaTrader 8 Desktop. This strategy will leverage order flow and market depth analysis for its core logic. The project must prioritize stability, comprehensive error handling, detailed logging for easy debugging, and include a Heads-Up Display (HUD) for real-time monitoring.

I. Core Trading Logic & Methodology:

- **Strategy Type:** Automated Trading Strategy (ATS) for futures markets (specify target instrument, e.g., ES, NQ, CL).
- **Primary Data Sources:**
    - **Order Flow Data:** Utilize NinjaTrader's volumetric bar data (footprint charts) or derive order flow insights from Level 1

tick data.
    - **Market Depth (Level 2 Data):** Actively process OnMarketDepth() events to maintain a real-time view of the order book. The strategy should analyze liquidity zones, order book imbalances, absorption, and exhaustion.
- **Specific Order Flow Concepts to Consider (Illustrative - AI to select/refine or suggest based on effectiveness):**
    - **Stacked Imbalances:** Identify and react to three or more consecutive bid/ask imbalances (e.g., 300% threshold). Consider automated zone detection and pullback entries.
    - **Exhaustion Prints:** Detect very low volume at swing highs/lows as potential reversal signals.
    - **Volume Profile Analysis:** Identify High-Volume Nodes (HVNs) and Low-Volume Nodes (LVNs) for support/resistance.
    - **Delta Analysis:** Analyze the difference between aggressive buy and sell orders. Look for delta divergences.
    - **Absorption:** Identify scenarios where strong selling/buying pressure is absorbed by passive orders without significant price movement.
- **Entry/Exit Conditions:** Must be clearly defined, objective, and quantifiable based on the chosen order flow/market depth phenomena.
- **Order Management:**
    - Utilize NinjaScript order submission methods (e.g., EnterLong(), EnterShort(), SubmitOrderUnmanaged()).
    - Implement robust stop-loss and profit-target mechanisms (e.g., SetStopLoss(), SetProfitTarget()).
    - Manage order lifecycle through OnOrderUpdate(), OnExecution(), and OnPositionUpdate().

    - Consider using market orders for entries where speed is critical, as suggested for stacked imbalances.

II. NinjaScript Architecture & Best Practices:

- **Foundation:** The strategy will be a C# class inheriting from NinjaTrader.NinjaScript.Strategies.Strategy.
- **Event-Driven Model:**
    - **OnStateChange():** Critical for managing the script's lifecycle.
        - State.SetDefaults: Initialize default input parameters, plot definitions, and core strategy properties (e.g., Calculate, EntriesPerDirection, IsOverlay).
        - State.Configure: Add any necessary additional data series (AddDataSeries()).
        - State.DataLoaded: Initialize all indicators (e.g., SMA(), EMA()) and custom Series<T> objects. Access Instrument details. This is the earliest point to safely access historical data.
        - State.Historical: For UI interactions if needed.
        - State.Realtime: Handle reconciliation of historical and live Order objects.
        - State.Terminated: Dispose of any custom unmanaged resources (e.g., StreamWriter, custom timers). Set large objects to null.
    - **OnBarUpdate():** Primary logic for bar-based calculations and trading decisions. Adhere to the specified Calculate mode (e.g., Calculate.OnBarClose, Calculate.OnEachTick).
        - If using Calculate.OnEachTick, use IsFirstTickOfBar to control logic that should only run once per bar.
    - **OnMarketData(MarketDataEventArgs e):** For tick-level responsiveness if required by the strategy beyond

OnBarUpdate().
    - **OnMarketDepth(MarketDepthEventArgs e):** Core for Level 2 data processing.
        - Maintain local bid/ask book representations (e.g., using SortedDictionary<double, long>).
        - Process e.Operation (Add, Update, Remove) and e.MarketDataType (Bid, Ask) correctly.
        - Handle e.IsReset by clearing local order book structures.
- **Readability & Maintainability:**
    - Use descriptive names for variables, methods, and properties.
    - Employ judicious commenting, especially for complex logic.
    - Organize code with helper methods for modularity and reusability.
    - Avoid magic numbers; use named constants or input parameters.
- **Performance Considerations:**
    - Minimize computations in frequently called methods like OnBarUpdate() and OnMarketDepth().
    - Cache indicator references initialized in State.DataLoaded.
    - Avoid redundant calculations.
    - Never use Thread.Sleep().
    - Be mindful of barsAgo indexing outside of core data event handlers.

III. Error Handling & Robustness:

- **Null Object Error Prevention:**
    - **DILIGENTLY IMPLEMENT NULL CHECKS** before accessing any member of objects that might not be initialized. This includes:
        - Instrument object (especially in OnStateChange() or if no

instrument is selected).
        - Bars and Bars.Instrument.
        - ChartControl (if performing UI operations).
        - Indicator instances (e.g., mySMA == null). Check before accessing their values, especially if conditionally initialized.
    - Initialize indicator variables properly in State.DataLoaded.
    - Check early and often, especially in OnBarUpdate().
    - Log descriptive messages when a null check fails.
- **General Error Handling:**
    - Use try-catch blocks judiciously for specific, anticipated exceptions (e.g., file I/O, division by zero). Avoid overly broad try-catch blocks.
    - Use safe casting (as keyword followed by a null check) to prevent InvalidCastException.
    - Handle double comparisons with tolerance (e.g., using Instrument.MasterInstrument.TickSize or ApproxCompare()).
- **Error Recovery:**
    - Design the strategy to recover gracefully from non-fatal errors where possible.
    - Log errors extensively to aid in diagnosing issues that could lead to unexpected behavior or financial loss.

IV. Debugging & Logging:

- **Verbose Print and Log Statements:**
    - **Print() Method:** Use extensively for debugging and informational messages during development and real-time execution. Output to NinjaScript Output window.
        - Include timestamps, variable states, and execution flow markers.

        - Example: Print(Time[0] + " | MyStrategy | OnBarUpdate | Close[0]: " + Close[0] + " | Condition XYZ met.");
    - **Log() Method:** Use for significant strategy events that should be recorded in the main NinjaTrader Log tab and log file.
        - Use NinjaTrader.Cbi.LogLevel (e.g., Information, Warning, Error, Alert).
        - LogLevel.Alert will also generate a pop-up. Use judiciously to avoid excessive pop-ups.
        - Be mindful of memory consumption with excessive Log() calls.
    - **Caller Information Attributes (for Custom Logging Helper):**
        - Implement a static helper class (e.g., LogHelper.PrintWithInfo()) that uses [CallerMemberName], [CallerFilePath], and [CallerLineNumber] attributes.
        - This will automatically include the calling method name, file path (use System.IO.Path.GetFileName() for conciseness), and line number in log messages, significantly aiding debugging.
- **Visual Studio Debugging:**
    - Ensure "Debug Mode" is enabled in NinjaTrader's NinjaScript Editor and the script is recompiled.
    - Attach the Visual Studio debugger to the NinjaTrader.exe (or NinjaTrader64.exe) process.
    - Set breakpoints, step through code (F10, F11), inspect variables in Locals/Watch windows, and use the Call Stack.
    - If code is changed in Visual Studio, save and **recompile in NinjaTrader** before re-attaching the debugger.

    - After debugging, uncheck "Debug Mode" in NinjaTrader and recompile for optimal performance.

V. Heads-Up Display (HUD):

- **Objective:** Provide a lightweight, on-chart or separate minimal window HUD for real-time monitoring.
- **Implementation:**
    - If on-chart, create as a NinjaScript Indicator that adds/modifies WPF elements on ChartControl.ChartPanel.
    - If a separate window, create as a NinjaScript AddOn launching a small, borderless, always-on-top WPF window.
- **Information to Display (Examples - AI to select/refine):**
    - Current Strategy State (e.g., "Long Active", "Seeking Entry", "Flat").
    - Active Order Status (e.g., "Stop Pending at X", "Limit Buy at Y").
    - Unrealized P&L for the current trade/instrument.
    - Key Level 2 insights (e.g., best bid/ask size, immediate imbalance for top N levels).
    - Critical alerts generated by the strategy.
- **Data Updates:**
    - HUD script to subscribe to relevant NinjaTrader events (OnMarketData, OnMarketDepth, OnOrderUpdate, OnPositionUpdate, AccountItemUpdate).
    - UI updates must use Dispatcher.InvokeAsync() or Dispatcher.BeginInvoke() as events often fire on non-UI threads.
- **WPF Controls:** Use TextBlock, Border, Grid for layout and data display.

VI. Development Environment & Tools:

- **Primary IDE:** Microsoft Visual Studio (Community Edition is sufficient).
  - Ensure ".NET desktop development" workload is installed.
  - Target .NET Framework 4.8.
  - Project type: C# Class Library.
  - Reference NinjaTrader DLLs (NinjaTrader.Core.dll, NinjaTrader.Gui.dll, NinjaTrader.NinjaScript.dll).
  - Configure a post-build event to copy the compiled DLL to the NinjaTrader bin\Custom directory.
- **Version Control:** Git, with a remote repository on GitHub (private).
  - Use feature branching for development (e.g., feature/core-logic, feature/hud-implementation).
  - Commit frequently with clear, descriptive messages.
  - Utilize a proper .gitignore file for Visual Studio C# projects.
- **(Optional) AI Pair Programmer:** GitHub Copilot within Visual Studio.
  - Use for boilerplate code generation, suggesting logic, explaining code, and assisting with debugging.
  - **CRITICAL:** Human oversight and validation of ALL AI-generated code is mandatory. Verify against official NinjaTrader documentation.

VII. Backtesting & Forward-Testing Considerations:

- **Level 2 Data Limitation:** Be acutely aware that OnMarketDepth() is **NOT** called during standard historical backtesting in Strategy Analyzer.
- **Testing Approach for Level 2 Logic:**
  - Extensive **forward-testing** (paper trading) on a live data

connection is essential.
    - Utilize NinjaTrader's Market Replay feature, understanding its data fidelity limitations for Level 2.
    - Standard backtesting will primarily validate logic that doesn't depend on OnMarketDepth().
- **General Testing:** Rigorously test across various instruments, timeframes, and historical periods.

Final Directive: The AI should prioritize creating a well-structured, commented, and robust codebase that is easy to understand, debug, and maintain, adhering to the specific architectural and operational nuances of the NinjaTrader 8 platform. The emphasis is on a production-quality autotrader.