Okay, this is a common and interesting challenge in trading strategy development! You want to create a "cascading" or "tiered" stop-loss system where the stop adjusts based on how quickly the price moves against your entry *within the entry candle itself or shortly after*.

Here's a conceptual NinjaScript code example and an explanation of how you might approach this.

Important Considerations and Disclaimers:

1. **Intra-bar Granularity:** NinjaTrader, by default, primarily processes logic OnBarUpdate(), which means once per bar on bar close (or on each tick if Calculate = Calculate.OnEachTick is set). For true intra-candle price tracking as you described ("track the price *during* the candle"), you'll need to ensure your script is running with Calculate = Calculate.OnEachTick or Calculate.OnPriceChange. This has performance implications, so use it judiciously.
2. **Order Execution Reality:** There will always be some latency between your script detecting a price, sending an order, and the exchange confirming it. In very fast-moving markets, slippage can still occur.
3. **Error Handling:** The "stop loss is above the market price" error (for a sell stop) or below (for a buy stop) typically happens if the market gaps or moves extremely fast right as you're trying to place the order. The logic below tries to mitigate this by canceling and replacing, but extreme volatility can still be a challenge.
4. **Backtesting Limitations:** Backtesting logic that relies heavily on intra-bar movements and order modifications can sometimes be less accurate than live trading if the backtesting engine

doesn't perfectly simulate tick-by-tick order book dynamics. Ensure you test thoroughly in simulation/live.

5. **NinjaTrader Version:** Syntax might vary slightly between NinjaTrader 7 and NinjaTrader 8. This example will aim for NinjaTrader 8 syntax.
6. **State Management:** You need to carefully manage the "state" of your stop losses (e.g., which one is currently active or pending).

Conceptual NinjaScript Code (NinjaTrader 8)

C#

```
#region Using declarations
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Xml.Serialization;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.Gui;
using NinjaTrader.Gui.Chart;
using NinjaTrader.Gui.SuperDom;
using NinjaTrader.Gui.Tools;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.Globals;
using NinjaTrader.NinjaScript.Indicators;
using NinjaTrader.NinjaScript.DrawingTools;
#endregion

//This namespace holds Strategies in this folder and is required. Do not change
it.
namespace NinjaTrader.NinjaScript.Strategies
{
	public class CascadingStopLoss : Strategy
	{
		private double stopLossLevel1Points = 3.5;
		private double stopLossLevel2Points = 5.0;
		private double stopLossLevel3Points = 10.0;

		private double entryPrice = 0;
		private IOrder entryOrder = null;
		private IOrder stopLossOrder1 = null;
		private IOrder stopLossOrder2 = null;
		private IOrder stopLossOrder3 = null;

		private bool stop1Active = false;
		private bool stop2Active = false;
		private bool stop3Active = false;
		
		private bool stop1Attempted = false;
		private bool stop2Attempted = false;
		private bool stop3Attempted = false;
```

```
                protected override void OnStateChange()
                {
                        if (State == State.SetDefaults)
                        {
                                Description
                = @"Strategy with cascading stop loss levels.";
                                Name
                = "CascadingStopLoss";
                                Calculate
        = Calculate.OnEachTick; // Crucial for intra-bar logic
                                EntriesPerDirection
        = 1;
                                EntryHandling
        = EntryHandling.AllEntries;
                                IsExitOnSessionCloseStrategy
= true;
                                ExitOnSessionCloseSeconds
        = 30;
                                IsFillLimitOnTouch
        = false;
                                MaximumBarsLookBack
                = MaximumBarsLookBack.TwoHundredFiftySix;
                                OrderFillResolution
        = OrderFillResolution.Standard;
                                Slippage
        = 0;
                                StartBehavior
        = StartBehavior.WaitUntilFlat;
                                TimeInForce
                = TimeInForce.Gtc;
                        TraceOrders
                = false;
                        RealtimeErrorHandling
        = RealtimeErrorHandling.StopCancelClose;
                        StopTargetHandling
        = StopTargetHandling.PerEntryExecution;
                        BarsRequiredToTrade
        = 20;
                        IsInstantiatedOnEachOptimizationIteration = true;

                        // Default stop loss levels (can be changed in strategy
properties)
                        StopLossLevel1Points = 3.5;
                        StopLossLevel2Points = 5.0;
                        StopLossLevel3Points = 10.0;
                    }
                    else if (State == State.Configure)
                    {
                    }
                    else if (State == State.DataLoaded)
                    {
                        // Optional: Add indicators here if needed for entry
signals
                    }
                }

                protected override void OnBarUpdate()
                {
                    if (CurrentBar < BarsRequiredToTrade)
                        return;
                // Entry Logic (Example: Enter long on a new bar for
demonstration)
                if (Position.MarketPosition == MarketPosition.Flat &&
entryOrder == null && CurrentBars[0] > 1) // Ensure not first bar of data
                {
                    // Replace this with your actual entry signal
                    if (Close[0] > Open[0] && IsFirstTickOfBar) //
Example entry condition
                    {
                        EnterLong(DefaultQuantity, "MyEntry");
                        entryPrice = Close[0]; // Or execution price if
available immediately
                        Print($"Entry initiated at {entryPrice}. Time:
{Time[0]}");

                        // Reset stop loss states on new entry attempt
                        ResetStopStates();
                    }
                }

                // Manage Stop Losses if in a position
                if (Position.MarketPosition != MarketPosition.Flat &&
entryPrice > 0)
                {
                    ManageCascadingStopLoss();
                }
            }

            protected override void OnExecutionUpdate(Execution execution,
string executionId, double price, int quantity, MarketPosition marketPosition,
string orderId, DateTime time)
                {
                        // Check if the execution is related to our entry order
                        if (entryOrder != null && entryOrder.Name ==
"MyEntry" && execution.Order.OrderId == entryOrder.OrderId)
                        {
                                if (execution.Order.OrderState ==
OrderState.Filled || execution.Order.OrderState ==
OrderState.PartFilled)
                                {
                                        entryPrice = execution.Price; // More
accurate entry price
                                        Print($"Entry filled. Actual entry price:
{entryPrice}. Time: {time}");
                                        // Initial attempt to set the first stop loss
                                        AttemptPlaceStopLoss1();
                                }
                                // Potentially handle if entry order is cancelled or
rejected
                                else if (execution.Order.OrderState ==
OrderState.Cancelled || execution.Order.OrderState ==
OrderState.Rejected)
                                {
                                        Print($"Entry order {entryOrder.Name} was
{execution.Order.OrderState}. Resetting.");
                                        entryOrder = null;
                                        entryPrice = 0;
                                        ResetStopStates();
                                }
                        }
```

```
                    // Check if a stop loss was filled
                    if ((stopLossOrder1 != null &&
execution.Order.OrderId == stopLossOrder1.OrderId &&
execution.Order.OrderState == OrderState.Filled) ||
                       (stopLossOrder2 != null &&
execution.Order.OrderId == stopLossOrder2.OrderId &&
execution.Order.OrderState == OrderState.Filled) ||
                       (stopLossOrder3 != null &&
execution.Order.OrderId == stopLossOrder3.OrderId &&
execution.Order.OrderState == OrderState.Filled))
                    {
                            Print($"Stop loss filled. Exited position. Price:
{execution.Price}, Time: {time}");
                            ResetStopStates();
                            entryPrice = 0; // Reset entry price after exit
                            entryOrder = null; // Allow new entries
                    }
            }

            protected override void OnOrderUpdate(Order order, double
limitPrice, double stopPrice, int quantity, int filled, double averageFillPrice,
OrderState orderState, DateTime time, ErrorCode error, string nativeError)
            {
                    // Catch if an entry order was submitted
            if (order.Name == "MyEntry" && entryOrder == null) // Check
if entryOrder isn't already set by OnExecutionUpdate for a fill
            {
            entryOrder = order;
                            Print($"Entry order submitted: {order.Name}, State:
{orderState}, Time: {time}");
            }
```

```
                        // Handle submitted stop loss orders
                        if (order.Name == "StopLoss1" && stopLossOrder1 ==
null) stopLossOrder1 = order;
                        if (order.Name == "StopLoss2" && stopLossOrder2 ==
null) stopLossOrder2 = order;
                        if (order.Name == "StopLoss3" && stopLossOrder3 ==
null) stopLossOrder3 = order;

                        // Error handling for stop orders specifically if they are
rejected (e.g., price too close)
                        // For a Long position, stop price would be below current
market. Error if trying to place it above.
                        // For a Short position, stop price would be above current
market. Error if trying to place it below.
                        if (orderState == OrderState.Rejected && error ==
ErrorCode.PriceTooCloseToMarket || error ==
ErrorCode.StopPriceViolatesRule150A) // Or other relevant error codes
                        {
                                Print($"Order {order.Name} rejected with error:
{error}. Native: {nativeError}. Order Stop Price: {order.StopPrice}, Current Ask:
{GetCurrentAsk()}, Current Bid: {GetCurrentBid()}");
                                if (Position.MarketPosition ==
MarketPosition.Long)
                                {
                                        if (order.Name == "StopLoss1" &&
stop1Attempted && !stop1Active)
                                        {
                                                Print($"StopLoss1 rejected for long
position (price: {order.StopPrice}). Attempting StopLoss2.");

CancelOrderIfExists(stopLossOrder1);
                                                stopLossOrder1 = null;
                                                stop1Active = false; // Ensure it's marked
inactive
                                                stop1Attempted = true; // It was
attempted
                                                AttemptPlaceStopLoss2(); // Try the
next level
                                        }
                                        else if (order.Name == "StopLoss2" &&
stop2Attempted && !stop2Active)
                                        {
                                                Print($"StopLoss2 rejected for long
position (price: {order.StopPrice}). Attempting StopLoss3.");

CancelOrderIfExists(stopLossOrder2);
                                                stopLossOrder2 = null;
                                                stop2Active = false;
                                                stop2Attempted = true;
                                                AttemptPlaceStopLoss3();
                                        }
                                        else if (order.Name == "StopLoss3" &&
stop3Attempted && !stop3Active)
                                        {
                                                Print($"StopLoss3 rejected for long
position (price: {order.StopPrice}). No further stops to attempt.");
                                                // Potentially exit market or handle as
critical error
                                        }
                                }
                                else if (Position.MarketPosition ==
MarketPosition.Short)
                                {
                                        // Similar logic for short positions
                                        if (order.Name == "StopLoss1" &&
stop1Attempted && !stop1Active)
                                        {
                                                Print($"StopLoss1 rejected for short
position (price: {order.StopPrice}). Attempting StopLoss2.");
                                                
CancelOrderIfExists(stopLossOrder1);
                                                stopLossOrder1 = null;
                                                stop1Active = false;
                                                stop1Attempted = true;
                                                AttemptPlaceStopLoss2();
                                        }
                                        // ... and so on for StopLoss2 and StopLoss3 for
short positions
                                }
                        }
                        else if (orderState == OrderState.Accepted ||
orderState == OrderState.Working)
                        {
                                if (order.Name == "StopLoss1") { stop1Active =
true; Print($"StopLoss1 active at {order.StopPrice}."); }
                                if (order.Name == "StopLoss2") { stop2Active =
true; Print($"StopLoss2 active at {order.StopPrice}."); }
                                if (order.Name == "StopLoss3") { stop3Active =
true; Print($"StopLoss3 active at {order.StopPrice}."); }
                        }
                        else if (orderState == OrderState.Cancelled)
                        {
                                Print($"Order {order.Name} was cancelled.");
                                if (order.Name == "StopLoss1" &&
stopLossOrder1 != null && order.OrderId == stopLossOrder1.OrderId)
{ stop1Active = false; stopLossOrder1 = null; }
                                if (order.Name == "StopLoss2" &&
stopLossOrder2 != null && order.OrderId == stopLossOrder2.OrderId)
{ stop2Active = false; stopLossOrder2 = null; }
                                if (order.Name == "StopLoss3" &&
stopLossOrder3 != null && order.OrderId == stopLossOrder3.OrderId)
{ stop3Active = false; stopLossOrder3 = null; }
                        }
                }

                private void ManageCascadingStopLoss()
                {
                        if (Position.MarketPosition == MarketPosition.Flat ||
entryPrice == 0)
                                return;

                        double currentPrice = GetCurrentBid(); // For long
positions, we watch the bid
                        if (Position.MarketPosition == MarketPosition.Short)
                                currentPrice = GetCurrentAsk(); // For short
positions, we watch the ask

                        // Define stop prices based on entry
                        double sl1Price = (Position.MarketPosition ==
MarketPosition.Long) ? entryPrice - StopLossLevel1Points * TickSize :
entryPrice + StopLossLevel1Points * TickSize;
                        double sl2Price = (Position.MarketPosition ==
MarketPosition.Long) ? entryPrice - StopLossLevel2Points * TickSize
```

```
: entryPrice + StopLossLevel2Points * TickSize;
                    double sl3Price = (Position.MarketPosition ==
MarketPosition.Long) ? entryPrice - StopLossLevel3Points * TickSize
: entryPrice + StopLossLevel3Points * TickSize;
                    
                    // --- Logic for LONG positions ---
                    if (Position.MarketPosition == MarketPosition.Long)
                    {
                        // If price drops below SL1 before SL1 is even active or
attempted, try to set SL2 directly
                        if (currentPrice < sl1Price && !stop1Active &&
!stop1Attempted && !stop2Active && !stop2Attempted)
                        {
                            Print($"Price ({currentPrice}) dropped below
SL1 level ({sl1Price}) before SL1 placement. Attempting SL2.");
                            AttemptPlaceStopLoss2();
                            return; // Exit this check cycle
                        }
                        // If price drops below SL2 before SL2 is active or
attempted (and SL1 might have failed or not been set)
                        else if (currentPrice < sl2Price && !stop2Active
&& !stop2Attempted && !stop3Active && !stop3Attempted)
                        {
                            Print($"Price ({currentPrice}) dropped below
SL2 level ({sl2Price}) before SL2 placement. Attempting SL3.");
                            CancelOrderIfExists(stopLossOrder1); //
Cancel SL1 if it was somehow submitted
                            AttemptPlaceStopLoss3();
                            return;
                        }
```

```
                // If SL1 is active and price drops below it (this means
the order should have filled)
                // This block is more for handling if the order didn't fill
and price kept going.
                // However, OnExecutionUpdate should ideally handle
the fill.
                // This intra-bar check can act as a fallback or
aggressive adjustment.
                if (stop1Active && stopLossOrder1 != null &&
currentPrice < stopLossOrder1.StopPrice)
                {
                    Print($"Price ({currentPrice}) dropped below
active SL1 ({stopLossOrder1.StopPrice}). Cancelling SL1 and attempting SL2.");
                    CancelOrderIfExists(stopLossOrder1); //
Cancel it
                    stop1Active = false;
                    stopLossOrder1 = null;
                    AttemptPlaceStopLoss2(); // Immediately try
to place SL2
                    return;
                }
                // If SL2 is active and price drops below it
                if (stop2Active && stopLossOrder2 != null &&
currentPrice < stopLossOrder2.StopPrice)
                {
                    Print($"Price ({currentPrice}) dropped below
active SL2 ({stopLossOrder2.StopPrice}). Cancelling SL2 and attempting SL3.");
                    CancelOrderIfExists(stopLossOrder2);
                    stop2Active = false;
                    stopLossOrder2 = null;
                    AttemptPlaceStopLoss3();
```

```
                                        return;
                                }

                                // If no stop is active yet, try to place the first one (this
is usually done on entry fill)
                                // This is a fallback if OnExecutionUpdate didn't trigger
it or if we want to be more aggressive.
                                if(!stop1Active && !stop1Attempted &&
!stop2Active && !stop2Attempted && !stop3Active &&
!stop3Attempted)
                                {
                                        AttemptPlaceStopLoss1();
                                }
                        }
                        // --- Logic for SHORT positions ---
                        else if (Position.MarketPosition ==
MarketPosition.Short)
                        {
                                // If price rises above SL1 before SL1 is even active, try
to set SL2 directly
                                if (currentPrice > sl1Price && !stop1Active &&
!stop1Attempted && !stop2Active && !stop2Attempted)
                                {
                                        Print($"Price ({currentPrice}) rose above SL1
level ({sl1Price}) before SL1 placement. Attempting SL2.");
                                        AttemptPlaceStopLoss2();
                                        return;
                                }
                                else if (currentPrice > sl2Price && !stop2Active
&& !stop2Attempted && !stop3Active && !stop3Attempted)
                                {
```

```
                    Print($"Price ({currentPrice}) rose above SL2
level ({sl2Price}) before SL2 placement. Attempting SL3.");
                    CancelOrderIfExists(stopLossOrder1);
                    AttemptPlaceStopLoss3();
                    return;
                }

                if (stop1Active && stopLossOrder1 != null &&
currentPrice > stopLossOrder1.StopPrice)
                {
                    Print($"Price ({currentPrice}) rose above active
SL1 ({stopLossOrder1.StopPrice}). Cancelling SL1 and attempting SL2.");
                    CancelOrderIfExists(stopLossOrder1);
                    stop1Active = false;
                    stopLossOrder1 = null;
                    AttemptPlaceStopLoss2();
                    return;
                }
                if (stop2Active && stopLossOrder2 != null &&
currentPrice > stopLossOrder2.StopPrice)
                {
                    Print($"Price ({currentPrice}) rose above active
SL2 ({stopLossOrder2.StopPrice}). Cancelling SL2 and attempting SL3.");
                    CancelOrderIfExists(stopLossOrder2);
                    stop2Active = false;
                    stopLossOrder2 = null;
                    AttemptPlaceStopLoss3();
                    return;
                }
                if(!stop1Active && !stop1Attempted &&
!stop2Active && !stop2Attempted && !stop3Active &&
!stop3Attempted)
                                {
                                        AttemptPlaceStopLoss1();
                                }
                        }
                }

                private void AttemptPlaceStopLoss1()
                {
                        if (Position.MarketPosition == MarketPosition.Flat ||
entryPrice == 0 || stop1Attempted || stop1Active)
                                return;

                        stop1Attempted = true; // Mark as attempted
                        double stopPrice = (Position.MarketPosition ==
MarketPosition.Long)
                                                                ?
Instrument.MasterInstrument.RoundToTickSize(entryPrice -
StopLossLevel1Points * TickSize)
                                                                :
Instrument.MasterInstrument.RoundToTickSize(entryPrice +
StopLossLevel1Points * TickSize);
                        
                        // Check if stop price is valid (not through current market
price already)
                        if (Position.MarketPosition == MarketPosition.Long
&& stopPrice >= GetCurrentBid()) {
                                Print($"SL1 for LONG at {stopPrice} is at or above
current Bid {GetCurrentBid()}. Cannot place. Will try SL2.");
                                // This triggers the cascade in OnOrderUpdate if
```

```
rejected or ManageCascadingStopLoss on next tick
                                        // For more immediate action, you could call
AttemptPlaceStopLoss2 directly here.
                                        // However, let's rely on the rejection or next tick to
keep it cleaner.
                                        // AttemptPlaceStopLoss2(); // Potentially too
aggressive here, let rejection handle or next OnBarUpdate cycle
                                        return; // Do not place
                              }
                              if (Position.MarketPosition == MarketPosition.Short
&& stopPrice <= GetCurrentAsk()) {
                                        Print($"SL1 for SHORT at {stopPrice} is at or below
current Ask {GetCurrentAsk()}. Cannot place. Will try SL2.");
                                        // AttemptPlaceStopLoss2();
                                        return; // Do not place
                              }

                              Print($"Attempting to place StopLoss1 at {stopPrice}.
Current Bid: {GetCurrentBid()}, Current Ask: {GetCurrentAsk()}");
                              if (Position.MarketPosition == MarketPosition.Long)
                                        ExitLongStopMarket(DefaultQuantity,
stopPrice, "StopLoss1", "MyEntry");
                              else if (Position.MarketPosition ==
MarketPosition.Short)
                                        ExitShortStopMarket(DefaultQuantity,
stopPrice, "StopLoss1", "MyEntry");
                    }

                    private void AttemptPlaceStopLoss2()
                    {
                              if (Position.MarketPosition == MarketPosition.Flat ||
entryPrice == 0 || stop2Attempted || stop2Active)
                    return;

                CancelOrderIfExists(stopLossOrder1); // Ensure
previous level is cancelled
                stop1Active = false; // Mark previous as inactive
                stop1Attempted = true; // Mark previous as attempted,
even if now moving to SL2

                stop2Attempted = true;
                double stopPrice = (Position.MarketPosition ==
MarketPosition.Long)
                                                ?
Instrument.MasterInstrument.RoundToTickSize(entryPrice -
StopLossLevel2Points * TickSize)
                                                :
Instrument.MasterInstrument.RoundToTickSize(entryPrice +
StopLossLevel2Points * TickSize);

                if (Position.MarketPosition == MarketPosition.Long
&& stopPrice >= GetCurrentBid()) {
                    Print($"SL2 for LONG at {stopPrice} is at or above
current Bid {GetCurrentBid()}. Cannot place. Will try SL3.");
                    // AttemptPlaceStopLoss3();
                    return;
                }
                if (Position.MarketPosition == MarketPosition.Short
&& stopPrice <= GetCurrentAsk()) {
                    Print($"SL2 for SHORT at {stopPrice} is at or below
current Ask {GetCurrentAsk()}. Cannot place. Will try SL3.");
                    // AttemptPlaceStopLoss3();
                    return;
                }

                Print($"Attempting to place StopLoss2 at {stopPrice}.
Current Bid: {GetCurrentBid()}, Current Ask: {GetCurrentAsk()}");
                if (Position.MarketPosition == MarketPosition.Long)
                    ExitLongStopMarket(DefaultQuantity,
stopPrice, "StopLoss2", "MyEntry");
                else if (Position.MarketPosition ==
MarketPosition.Short)
                    ExitShortStopMarket(DefaultQuantity,
stopPrice, "StopLoss2", "MyEntry");
            }

            private void AttemptPlaceStopLoss3()
            {
                if (Position.MarketPosition == MarketPosition.Flat ||
entryPrice == 0 || stop3Attempted || stop3Active)
                    return;

                CancelOrderIfExists(stopLossOrder1);
                CancelOrderIfExists(stopLossOrder2);
                stop1Active = false;
                stop2Active = false;
                stop1Attempted = true;
                stop2Attempted = true;

                stop3Attempted = true;
                double stopPrice = (Position.MarketPosition ==
MarketPosition.Long)
```

```
                                              ?
Instrument.MasterInstrument.RoundToTickSize(entryPrice -
StopLossLevel3Points * TickSize)
                                              :
Instrument.MasterInstrument.RoundToTickSize(entryPrice +
StopLossLevel3Points * TickSize);

                if (Position.MarketPosition == MarketPosition.Long
&& stopPrice >= GetCurrentBid()) {
                    Print($"SL3 for LONG at {stopPrice} is at or above
current Bid {GetCurrentBid()}. Cannot place. Critical - consider market exit.");
                    // ExitLong(); // Example: Bail out if even the widest
stop is invalid
                    return;
                }
                if (Position.MarketPosition == MarketPosition.Short
&& stopPrice <= GetCurrentAsk()) {
                    Print($"SL3 for SHORT at {stopPrice} is at or below
current Ask {GetCurrentAsk()}. Cannot place. Critical - consider market exit.");
                    // ExitShort();
                    return;
                }

                Print($"Attempting to place StopLoss3 at {stopPrice}.
Current Bid: {GetCurrentBid()}, Current Ask: {GetCurrentAsk()}");
                if (Position.MarketPosition == MarketPosition.Long)
                    ExitLongStopMarket(DefaultQuantity,
stopPrice, "StopLoss3", "MyEntry");
                else if (Position.MarketPosition ==
MarketPosition.Short)
                    ExitShortStopMarket(DefaultQuantity,
stopPrice, "StopLoss3", "MyEntry");
                }

                private void CancelOrderIfExists(IOrder order)
                {
                        if (order != null && order.OrderState !=
OrderState.Cancelled && order.OrderState != OrderState.Filled)
                        {
                                Print($"Cancelling order: {order.Name}");
                                CancelOrder(order);
                        }
                }

                private void ResetStopStates()
                {
                        Print("Resetting all stop states and orders.");
                        CancelOrderIfExists(stopLossOrder1);
                        CancelOrderIfExists(stopLossOrder2);
                        CancelOrderIfExists(stopLossOrder3);

                        stopLossOrder1 = null;
                        stopLossOrder2 = null;
                        stopLossOrder3 = null;

                        stop1Active = false;
                        stop2Active = false;
                        stop3Active = false;

                        stop1Attempted = false;
                        stop2Attempted = false;
                        stop3Attempted = false;
                }


                #region Properties
                [NinjaScriptProperty]
                [Range(0.1, double.MaxValue)]
                [Display(Name="StopLossLevel1 Points", Description="Initial stop
loss in points.", Order=1, GroupName="Parameters")]
                public double StopLossLevel1Points
                { get { return stopLossLevel1Points; }
                  set { stopLossLevel1Points = Math.Max(0.1, value); }
                }

                [NinjaScriptProperty]
                [Range(0.1, double.MaxValue)]
                [Display(Name="StopLossLevel2 Points", Description="Second
stop loss in points if first is skipped/fails.", Order=2,
GroupName="Parameters")]
                public double StopLossLevel2Points
                { get { return stopLossLevel2Points; }
                  set { stopLossLevel2Points = Math.Max(0.1, value); }
                }

                [NinjaScriptProperty]
                [Range(0.1, double.MaxValue)]
                [Display(Name="StopLossLevel3 Points", Description="Third stop
loss in points if second is skipped/fails.", Order=3, GroupName="Parameters")]
                public double StopLossLevel3Points
                { get { return stopLossLevel3Points; }
                  set { stopLossLevel3Points = Math.Max(0.1, value); }
```

```
            }
            #endregion
        }
}
```

Explanation and How It Works:

1. **Calculate = Calculate.OnEachTick;**: This is set in OnStateChange under State.SetDefaults. It's crucial because you want to react to price changes *within* the current bar, not just at the close of the bar.
2. **Stop Loss Point Variables (stopLossLevel1Points, etc.)**: These store your desired stop distances in points. They are not active orders initially. TickSize is used to convert these points to actual price offsets.
3. **Order Objects (stopLossOrder1, etc.)**: These will hold references to your actual stop-loss orders once they are submitted. This allows you to cancel or check their status.
4. **State Flags (stop1Active, stop1Attempted, etc.)**:
   - stopXActive: Becomes true when the corresponding Exit...StopMarket() order has been accepted by the broker and is working.
   - stopXAttempted: Becomes true as soon as an attempt is made to place the stop. This helps prevent multiple attempts for the same level and aids in the cascading logic.
5. **OnEntryExecution() / OnOrderUpdate() for Entry:**
   - When your entry order is filled (OnExecutionUpdate), we record the entryPrice. This is the base for calculating stop-loss prices.
   - Immediately after a fill confirmation, AttemptPlaceStopLoss1() is called.

6. **AttemptPlaceStopLossX() Methods:**
   - These methods calculate the actual stop price based on entryPrice and the respective StopLossLevelXPoints.
   - **Crucial Check:** Before submitting the stop order, they check if the calculated stop price is *already through the current market price*.
     - For a long position, if stopPrice >= GetCurrentBid(), it means the market has already dropped to or below your intended stop level. Placing the order would likely result in an immediate fill or rejection.
     - For a short position, if stopPrice <= GetCurrentAsk(), similar logic applies.
   - If the stop price is invalid (too close or already passed), it *doesn't* place that stop. The logic in ManageCascadingStopLoss or OnOrderUpdate (for rejections) will then attempt the next wider stop.
   - They submit the ExitLongStopMarket() or ExitShortStopMarket() order.
7. **ManageCascadingStopLoss() (Called from OnBarUpdate):**
   - This is the core of the intra-bar monitoring.
   - It continuously checks the GetCurrentBid() (for longs) or GetCurrentAsk() (for shorts) against your defined stop levels *even if no stop order is currently active*.
   - **Scenario 1: Price blows through Level 1 before it's even placed:**
     - If currentPrice is already beyond where sl1Price *would be*, and stop1 hasn't been attempted/activated, and stop2 hasn't either, it skips trying to place stop1 and directly calls AttemptPlaceStopLoss2().
     - Similar logic applies if it blows through sl2Price before

        stop2 is attempted.
    - **Scenario 2: An active stop is breached intra-bar (but order hasn't filled yet):**
        - If stop1Active is true and currentPrice drops below stopLossOrder1.StopPrice, it means the market has hit your active stop. While the order *should* fill, this provides a more aggressive mechanism to CancelOrderIfExists(stopLossOrder1) and then immediately AttemptPlaceStopLoss2(). This is the part that addresses your "cancel the 3.5 point stop loss and move to the 5-point stop loss if the price drops sharply."

8. **OnOrderUpdate() for Stop Orders:**
    - This is vital for reacting to how the broker handles your stop orders.
    - **Rejection Handling:** If a stop order (e.g., "StopLoss1") is Rejected (often with ErrorCode.PriceTooCloseToMarket or similar), it means your initial attempt to place it failed because the price was too close or had already passed the stop level *at the moment the broker processed the order submission*.
        - In this case, the code will:
            - Print a message.
            - Cancel the (now useless) rejected order object reference.
            - Mark stop1Active = false;.
            - Immediately call AttemptPlaceStopLoss2() to try the next wider stop.
    - **Accepted/Working:** If a stop order is Accepted or Working, the corresponding stopXActive flag is set to true.
    - **Cancelled:** If an order is cancelled (either by your code or

manually), the flags and order objects are reset.

9. **CancelOrderIfExists(IOrder order):** A helper to safely cancel an order only if it exists and isn't already filled or cancelled.
10. **ResetStopStates():** Called on a new entry or when a position is closed to ensure a clean state for the next trade.

How to Use:

1. Copy this code into a new Strategy file in NinjaTrader's NinjaScript Editor.
2. Compile it.
3. Add it to a chart.
4. Adjust the StopLossLevelXPoints parameters in the strategy settings as needed.
5. **Thoroughly test in simulation (Sim101) and then with very small size in live trading.** Pay close attention to the "Output" window (for Print() messages) and the "Orders" tab in the Control Center to understand how the orders are being managed.

This is a complex piece of logic, and rigorous testing is key! Good luck!

Sources
1. https://github.com/WesleyMcCammon/NinjaTrader
2. https://ninjatrader.com/support/forum/forum/ninjatrader-8/strategy-development/101866-max-trades-per-day?p=801366
3. https://ninjatrader.com/support/helpGuides/nt8/onexecutionupdate.htm
4. https://ninjatrader.com/support/helpguides/nt8/ninjascript_best_practices.htm