

Super Detailed Guide: Using NinjaScript & Gemini AI

Prepared by EgoNoBueno

Super Detailed Guide: Using NinjaScript & Gemini AI	1
1. Introduction: Making NinjaScript Better with a Super Helper AI, Gemini	2
2. Understanding NinjaScript: Foundations for AI-Assisted Development	6
2.1. Purpose and Core Capabilities of NinjaScript	6
2.2. NinjaScript and C#: Leveraging a Powerful Language	7
2.3. Key Syntactical Elements and Data Types	8
2.4. The NinjaScript Development Environment: NinjaTrader Editor vs. External IDEs (e.g., VS Code)	13
2.5. Interfacing with Data Providers in NinjaTrader	17
3. NinjaScript Architecture and Code Organization Best Practices	18
3.1. The Event-Driven Model: OnBarUpdate() and Other Key Event Handlers	18
3.2. State Management in NinjaScript: OnStateChange() and its States	20
3.3. Architectural Schematic for a NinjaScript Project (A Picture in Your Mind)	29
3.4. Best Practices for Readability, Maintainability, and Performance	33
3.5. Error Handling and Debugging Fundamentals in NinjaScript	38
4. Gemini AI: Capabilities for C# and NinjaScript Enhancement	42
4.1. Overview of Gemini Code Assist and its Relevance to C#	43
4.2. Code Generation: From Snippets to Functions	44
4.3. Code Analysis and Understanding Existing NinjaScript	45
4.4. Debugging Assistance with Gemini	45
4.5. Code Explanation and Documentation Generation	46
4.6. Local Codebase Awareness and Customization (A More Advanced Feature)	47
5. Mastering Prompt Engineering for NinjaScript with Gemini	51
5.1. General LLM Prompting Best Practices (Good Ways to Talk to Any Smart AI)	52
5.2. Crafting Effective Prompts for NinjaScript Code Generation	55
5.3. Prompting Strategies for Code Explanation, Debugging, and Optimization in NinjaScript	57
5.4. Engineered AI Prompt Template for Initiating a New NinjaScript Project	59
NinjaScript Project Initiation Prompt for Gemini	59
1. Core Directive (Your Main Goal for Gemini):	59
2. NinjaScript Type:	60
3. Project Objective (What is this script supposed to do?):	60
4. Key Input Parameters (Settings the user can change in NinjaTrader):	60
5. Core Logic Requirements (Especially for OnBarUpdate or other key event handlers):	60
6. Practical Integration: Gemini in the NinjaScript Development Workflow (How to Actually Use Gemini When You Code)	61

6.1. Setting up Gemini Code Assist in Your Preferred IDE (e.g., VS Code)	62
6.2. Workflow Example: Developing a New Indicator with Gemini's Help (Step-by-Step)	64
6.3. Workflow Example: Understanding and Modifying Existing NinjaScript Strategies	70
6.4. Programmatic Access via Gemini SDK (This is for Super Advanced Users!)	72
7. Illustrative Examples and Use Cases (Seeing Gemini in Action)	73
7.1. Example 1: Generating a Moving Average Crossover Strategy Stub (The Basic Framework)	74
7.2. Example 2: Explaining a Complex Custom Indicator's OnBarUpdate() Logic	86
7.3. Example 3: Debugging a Common NullReferenceException in NinjaScript with Gemini	92
7.4. Example 4: Optimizing a Loop or Calculation within a NinjaScript (Making it Faster)	100
8. Addressing Potential Challenges and Limitations (Things to Watch Out For)	115
8.1. Understanding AI Hallucinations and Ensuring Code Accuracy (AI Making Stuff Up!)	116
8.2. Handling NinjaScript-Specific Nuances and Proprietary API Aspects (NinjaScript's Special Rules)	118
8.3. Dependency on Training Data and Staying Updated (AI Doesn't Know Tomorrow's News)	119
8.4. Rate Limits and API Usage Considerations (If You're Using the API Directly)	120
8.5. The Importance of Human Oversight and Testing (YOU Are Still the Boss!)	121
9. Conclusion: Advancing Your NinjaScript Development with AI (Wrapping It All Up)	129
Works Cited	133

1. Introduction: Making NinjaScript Better with a Super Helper AI, Gemini

Imagine the world of buying and selling things like stocks (parts of companies) or futures (promises to buy something later). This world has changed a LOT because of something called **algorithmic trading**. "Algorithmic" just means using a set of rules, and in this case, it means using smart computer programs to make trading decisions and actions really fast and super accurately.

A very popular computer program that people use for this is called **NinjaTrader**. Lots of regular folks who trade from home, and also big

professional trading companies, use NinjaTrader. Why? Because it has great tools for:

- Making charts that show how prices are moving.
- Analyzing (studying) all that price information.
- Letting computers do the trading automatically.

Now, to really unlock all the cool things NinjaTrader can do and make it work exactly how *you* want, there's a special coding language called **NinjaScript**. Think of NinjaScript as the secret recipe book for NinjaTrader. It's based on a common and powerful computer language called C# (you say it "C sharp").

With NinjaScript, people who know how to code (developers) and traders themselves can build their own:

- **Custom indicators:** These are special tools that can draw lines on charts, show signals, or calculate unique numbers to help understand what the market might do next.
- **Sophisticated automated trading strategies:** These are like robot traders! You write down a very detailed plan (the strategy) in NinjaScript, and the computer will follow that plan to buy and sell automatically.
- **Bespoke add-on windows:** These are extra, custom-made screens or panels that can be added to the NinjaTrader platform to show special information or provide new buttons and controls.

Basically, NinjaScript lets you change and add to NinjaTrader so it fits your own unique trading ideas and methods perfectly. ¹

Why would someone want to build their own custom stuff? Here are the main good reasons (the "value proposition"):

- **Enhanced speed:** Computer programs can react to changes in

the market much, much faster than a human can.

- **Precision in executing complex logic:** If your trading idea has a lot of complicated "if this, then that" rules, a computer can follow them perfectly every time.
- **Ability to rigorously backtest strategies:** "Backtesting" means trying out your trading idea on old market data from the past to see how it would have performed. This helps you see if your idea is any good before you risk real money.
- **Mitigation of emotional biases:** Humans can get scared, greedy, or overly excited when trading, which can lead to bad decisions. Computers just follow the rules you give them, so they don't have these emotional problems.

At the same time all this trading stuff is happening, the way people write computer programs (software development) is also going through a huge change. This is because of amazing improvements in **Artificial Intelligence (AI)**, especially something called **Large Language Models (LLMs)**. You might have heard of AI that can write stories or answer questions. Well, some AI, like **Google's Gemini**, are really good at helping with computer code too! ³

Gemini Code Assist, for example, is a version of Gemini that helps programmers. It can do things like:

- Suggest how to finish a line of code as you're typing it.
- Write whole chunks of code for you if you just describe what you want.
- Look at the code you've written and explain what it does in plain English.
- Help find and fix mistakes (bugs) in your code. ⁵

So, we have this special trading language, NinjaScript, and this

super-smart AI code helper, Gemini. Putting them together could be really powerful! Using Gemini to help write NinjaScript could make building trading tools faster, make the code better, and maybe even help people come up with brand new trading ideas.

This guide is all about showing you, in a very detailed and practical way, how to use Gemini AI to help you when you're building things with NinjaScript. We'll cover:

- The basic things you need to know about NinjaScript.
- How NinjaScript programs are usually structured (their "architecture").
- What Gemini can do with C# (the language NinjaScript is based on).
- How to write good instructions (prompts) for Gemini so it gives you the best help.
- Real examples of how you can use Gemini when you're actually working on a NinjaScript project.
- Some problems you might run into and how to deal with them.

The main goal is to give you clear, step-by-step help so you can use this powerful AI to make awesome trading tools.

AI tools like Gemini can also make it easier for more people to get into developing complex trading algorithms. NinjaScript, because it uses C#, is already pretty understandable if you have some programming experience. But AI can be like a turbo-boost, helping you learn faster and build complicated trading ideas more quickly. This means if you have great trading ideas but maybe aren't a super expert coder, Gemini could help you turn those ideas into working NinjaScript. And if you *are* an expert coder, Gemini can help you try out new things faster or quickly learn parts of NinjaScript you haven't

used before. So, this guide is for all kinds of developers, and it will always remind you that it's super important to understand and double-check any code that an AI helps you write.

2. Understanding NinjaScript: Foundations for AI-Assisted Development

Before you can get an AI like Gemini to help you well, you really need to understand the basics of NinjaScript itself. If you know how NinjaScript works, you can tell the AI what to do much better, you can check if the AI's ideas are good, and you can easily put the code Gemini helps you write into your NinjaTrader program.

2.1. Purpose and Core Capabilities of NinjaScript

NinjaScript is basically a collection of special C# tools (like classes, methods, and properties – don't worry too much about those exact words yet) that NinjaTrader gives you. These tools let you build your own custom trading strategies, chart indicators, and extra windows that work right inside the NinjaTrader platform. ²

What's its main job? To let you add your own special features to NinjaTrader. Traders and developers use it to create their own unique ways of looking at the market and their own automatic trading systems.

What can you build with it?

- **Simple visual indicators:** For example, an indicator that draws a blue dot on your chart whenever a certain price pattern happens.
- **Complex automated strategies:** These can be full robot traders that watch the market, make decisions based on rules you've programmed, and then place buy or sell orders all by

themselves, without you having to click anything.

NinjaScript lets your custom code talk directly to NinjaTrader's live market data and its system for sending orders to the market. This makes it a really powerful way to create trading tools that are perfectly customized for you.

2.2. NinjaScript and C#: Leveraging a Powerful Language

A super important thing about NinjaScript is that it's built using the **C# programming language**. Specifically, it uses a version called C# 8, which works with something called the .NET 4.8 framework. ² (These are just version numbers, like for a game or an app).

This is a big plus because Gemini Code Assist is officially good at helping with C#! ⁵ So, Gemini can understand and help you write NinjaScript more directly and effectively.

Using C# gives NinjaScript a lot of advantages because C# is a well-known, powerful language that's used for many different things, not just trading. Some of these advantages are:

- **Object-Oriented:** This is a fancy way of saying that C# helps you organize your code into neat, reusable pieces (like building with LEGOs). This makes your code cleaner and easier to manage.
- **Full Access to .NET Framework Libraries:** The .NET Framework is like a giant toolbox full of ready-to-use code for all sorts of common tasks. NinjaScript can use anything in this toolbox, like tools for doing math, working with dates and times, organizing lists of things, and much more.

This is different from some other trading platforms that have their own unique, custom-built languages. Those languages might be

simpler for very basic trading tasks, but C# gives NinjaScript the power and flexibility to build much more complex and detailed trading logic.

2.3. Key Syntactical Elements and Data Types

NinjaScript code follows the grammar rules of C#. Here are some of the basics:

- **Statements:** A statement is a complete instruction that the computer can understand. In C#, statements almost always end with a **semicolon (;)**. Think of it like a period at the end of a sentence. ⁷
 - Example: `int myNumber = 10;`
- **Blocks of Code:** When you have several statements that belong together (like the steps in a recipe), you group them inside **curly braces ({ })**. This is called a code block. ⁷

- Example:

```
C#  
if (price > 100)  
{  
    Print("Price is high!");  
    // Maybe do something else here  
}
```

- **Comments:** These are notes you write in your code for humans to read. The computer ignores them. Comments are super important for explaining what your code does, making it easier for you (and others, including AI!) to understand later. ⁷
 - **Single-line comment:** Starts with `//`. Everything after `//` on that line is a comment.
 - Example: `// This is a note for me.`
 - **Multi-line comment:** Starts with `/*` and ends with `*/`.

Everything between these is a comment.

- Example:

```
C#  
/*  
This is a longer note  
that spans multiple lines.  
*/
```

- **Case Sensitivity:** C# cares about uppercase and lowercase letters. This means if you have a variable named myVariable, it's different from MyVariable or myvariable. You have to be careful with your spelling and capitalization! ⁷

Common Types of Information (Data Types) in NinjaScript:

When you want to store information in your code, you use something called a variable. You have to tell the computer what kind of information the variable will hold. Here are some common types you'll use a lot in NinjaScript ⁷:

- **string:** Used for storing text, like words or sentences.
 - Example: string message = "Buy signal detected";
- **double:** Used for numbers that can have decimal points. This is what you'll usually use for prices or the values your indicators calculate.
 - Example: double entryPrice = 150.25;
- **int:** Stands for "integer." Used for whole numbers (no decimals).
 - Example: int lookbackPeriod = 14; (like for a 14-day moving average)
- **bool:** Stands for "boolean." Used for values that can only be true or false. These are great for checking conditions.
 - Example: bool isMarketOpen = true;
- **Object types:** Sometimes, a variable will hold a more complex thing, like an entire indicator. In NinjaScript, you'll often have variables that are specific types of objects.
 - Example: SMA mySmaIndicator = SMA(Close, 14); (Here,

mySmaIndicator is a variable that holds a Simple Moving Average indicator).

Declaring Variables:

To create a variable, you first say its data type, then give it a unique name, and then you can (optionally) give it an initial value using the equals sign (=). ⁷

- Example: `int numberOfShares = 100;`

Operators (Math and Logic Symbols):

NinjaScript uses common symbols for doing math and making comparisons ⁷:

- **Arithmetic Operators:** + (add), - (subtract), * (multiply), / (divide).
- **System.Math:** For more advanced math functions (like square roots, powers, etc.), C# gives you access to a built-in toolkit called `System.Math`. For example, `Math.Abs()` gives you the absolute value of a number. ⁷

A Special NinjaScript Data Holder: `DataService` or `Series<T>`

This is a really important one for NinjaScript! A `DataService` (or more generally, `Series<double>` if it's holding numbers with decimals) is like a special list that's designed to hold information that changes over time, like prices or indicator values for each bar on your chart.

For example, `Close` in NinjaScript is a `Series<double>` that holds the closing price for every bar. When you build indicators, they usually work with these kinds of series.

Table 1: NinjaScript Core Syntax and Constructs (The Basic Building Blocks)

What It Is (Construct Type)	How It Looks (Syntax Example)	What It Does (Description)	Why It's Important for NinjaScript
Statement	<code>int x = 10;</code>	A complete instruction for the computer. It always ends with a semicolon	This is the basic unit of any logic you write.

		(;). ⁷	
Code Block	if (Close > Open) { /* your logic here */ }	Groups one or more statements together. It's marked by curly braces { }. ⁷	It helps define where variables can be used and groups code for things like if conditions or loops.
Single-line Comment	// This is a comment	The computer ignores this. It's a note for humans. ⁷	Makes your code easier to read and remember what it does.
Multi-line Comment	/* This is a multi-line comment */	Also ignored by the computer. Used for longer notes. ⁷	Good for detailed explanations or for temporarily "turning off" a chunk of code.
Case Sensitivity	myVar is different from MyVar	C# treats uppercase and lowercase letters as different in names. ⁷	You need to be very careful with capitalization to avoid mistakes.
Variable (type int)	int period = 14;	Stores whole numbers. ⁷	Often used for things like how many bars an indicator looks back (its

			period), counting loops, or numbering bars.
Variable (type double)	double priceLevel = 150.75;	Stores numbers with decimal points. ⁷	The main type for prices, indicator values, and any calculations that need to be exact.
Variable (type bool)	bool isTradeAllowed = true;	Stores either true or false. ⁷	Used for making decisions (conditional logic), setting flags (on/off switches), and turning features on or off.
Variable (type string)	string message = "Entry signal";	Stores text. ⁷	Used for things like naming your plots on a chart, messages for alerts, or writing notes to a log file.
Object Variable	SMA mySma = SMA(Close, 10);	Stores a more complex thing, like an instance of an indicator. ⁷	Super important for using built-in NinjaTrader indicators (like SMA, EMA, RSI)

			or custom ones you build.
Series<double>	Series<double> customValues = new Series<double>(this);	A special list for time-based data, usually prices or indicator values for each bar.	This is the main way you get historical bar data (like Close, High, Low, Open) and the output from your indicators.
Arithmetic Operator	double result = (High + Low) / 2;	Does math like +, -, *, /. ⁷ (Note: High means the high price of the current bar).	Essential for all calculations in your trading logic and indicator formulas.

This table is like a quick cheat sheet. It shows the basic pieces you need to understand and write NinjaScript code. When Gemini helps you generate C# code, knowing these pieces will help you ask for the right things and understand what the AI gives you.

2.4. The NinjaScript Development Environment: NinjaTrader Editor vs. External IDEs (e.g., VS Code)

When you want to write NinjaScript code, you have a couple of choices for where to do it.

1. NinjaTrader's Built-in Editor:

NinjaTrader comes with its own simple code editor. You can find it by going to "New" in the NinjaTrader Control Center (the main window) and choosing "NinjaScript Editor." ⁸ You can create new indicators or strategies here and do some basic editing.

However, many experienced coders find this built-in editor a bit basic. They say it has "limited functionality." ⁸ For example, it might not be great at helping you organize big chunks of code, its undo/redo might not be very powerful, and it might not be as good at matching up your opening and closing brackets or parentheses. ⁹

2. External IDEs (Super-Powered Code Editors):

For a much better and more powerful coding experience, most developers recommend using an external Integrated Development Environment (IDE). Think of an IDE as a special workshop just for coders, with lots of cool tools. Popular ones are Microsoft Visual Studio or Visual Studio Code (VS Code). ⁸ These IDEs give you "more control, better debugging options, and an improved coding environment." ⁸

- **How it usually works:** You edit your NinjaScript files (they end in .cs because they are C# files) directly in VS Code or Visual Studio. NinjaTrader can be set up to watch these files. When you save a change in your external IDE, NinjaTrader can automatically recompile (re-check and re-build) your script. This auto-compile feature works best if you also keep the NinjaScript Editor window for that same script open inside NinjaTrader. ⁹
- **VS Code is a great choice:** If you don't want to pay for the full Visual Studio, VS Code is a fantastic free option. It's very powerful, and you can add extensions (like extra tools) to make it even better for C# coding. ⁹

Why does this matter for using AI like Gemini?

Advanced AI helpers like Gemini Code Assist are usually designed to work as plugins or extensions inside popular IDEs like VS Code or the JetBrains IDEs (another family of code editors). ⁵ The basic NinjaScript editor inside NinjaTrader probably won't be able to use these AI extensions directly.

So, if you want to use all the cool features of Gemini that work right in your editor (like getting code suggestions as you type), using an external IDE like VS Code isn't just for convenience – it becomes pretty much necessary. The good news is that the common way of editing NinjaScript files in VS Code while NinjaTrader handles the compiling ⁹ is exactly what you need to make this AI-powered coding work!

Table 2: Comparing Places to Write NinjaScript Code

Feature	NinjaTrader Editor	Visual Studio Code (VS Code) with C# Extension	Visual Studio (Full Version)
Code Suggestions (Completion)	Basic	Very Smart (called IntelliSense)	Very Smart (IntelliSense)
Finding Mistakes (Debugging)	Limited (mostly using Print() statements, some basic NT8 tools) ¹⁰	Advanced (can attach a .NET debugger, set breakpoints to pause code, watch variable values)	Full-featured .NET Debugger
Moving Around in Code	Basic	Good (Go to Definition, Find All References, etc.)	Lots of powerful navigation tools
AI Helper	Not really	Excellent (e.g.,	Excellent (e.g.,

Integration	supported	Gemini Code Assist Extension) ⁵	GitHub Copilot, Gemini Code Assist)
Reusable Code Pieces (Snippets)	Limited or you do it manually	Lots, and you can make your own	Lots, and you can make your own
Tracking Code Changes (Version Control)	You have to do it manually with other tools	Built-in support for Git (a popular tool) and other extensions	Built-in Git support, Team Explorer
Smart Code Rewriting (Refactoring)	Very little	Good (with C# extensions)	Excellent
Changing How It Looks (UI Customization)	Fixed, you can't change it	You can change almost anything	You can change almost anything
Cost	Included with NinjaTrader	Free	Free (Community version) to Paid (Pro/Enterprise versions)
Learning Advanced Features	Not many to learn	Takes some time to learn all it can do	Takes some time to learn all it can do
How Much Computer Power	Low	Medium	Medium to High

It Uses			
---------	--	--	--

This table shows that external IDEs, especially if you want to use AI help and modern coding tools, have a lot of advantages.

2.5. Interfacing with Data Providers in NinjaTrader

NinjaScript works with market data (like prices and volume) that NinjaTrader gets from different **data providers**. NinjaTrader can connect to many different brokers (the companies that handle your trades) and special data feed services. ¹

A cool feature is "**Multi-Provider Mode**." This means NinjaTrader can be connected to more than one data source at the same time if you need it. ² For example, you might get your stock price data from one company and your currency (forex) data from another.

How do you set up these connections?

You usually do this in NinjaTrader's "Connections" menu. Different data providers might have slightly different setup steps. For example, if you're connecting to a data provider called Rithmic (maybe through a service like Topstep), you often need to have Rithmic's own program, RTrader Pro, installed and running with a successful login before NinjaTrader can connect to it. ¹²

Once your data feeds are set up and working in NinjaTrader, your NinjaScript code automatically uses this data for its calculations, for deciding when to trade (if it's a strategy), and for sending orders. ² When you see things like Input (the main price data for an indicator) or the built-in Close, High, Low, Open, Volume, and Time series in your script, these are all filled with data from the connections you've configured.

What about AI and data feeds?

AI tools like Gemini can be awesome for helping you write the NinjaScript logic that uses this market data. But, it's super important to remember that the AI can't control the data feed itself.

Things like:

- Making sure the data feed is set up correctly in NinjaTrader.
- The quality and accuracy of the data coming from the provider.
- Whether your internet connection to the data provider is stable.
- Any special rules for your data provider (like needing R|Trader Pro to be running for Rithmic ¹²).

...are all things that *you*, the developer or user, have to manage. Gemini can help you write a brilliant trading algorithm, but if the data going into that algorithm is bad or missing, the algorithm won't work well. Gemini can't fix a broken data connection or tell you if you've picked the wrong stock symbol in your NinjaTrader setup.

So, making sure your data connections are solid is a really basic and important part of NinjaScript development, whether you're using AI to help write the script's logic or not.

3. NinjaScript Architecture and Code Organization Best Practices

To build NinjaScript programs that work well, are easy to understand, and don't break easily (we call this "robust and maintainable"), you need to understand how they're structured. This includes how NinjaTrader tells your script when to do things (its "event-driven architecture"), especially how it manages different "states" or stages of your script's life. You also need to follow good habits for organizing your code, making it run fast, and handling errors.

These good habits are not just for you as a human coder. They also help you tell an AI like Gemini what to do so it can create good, efficient code for you.

3.1. The Event-Driven Model: OnBarUpdate() and Other Key Event Handlers

NinjaScript works using an **event-driven model**. This means that specific parts of your script (called methods or functions) are

automatically run by the NinjaTrader platform when certain "events" happen in the market or in the platform itself. ¹ Think of it like a doorbell: your script is waiting, and when an event (the doorbell ringing) happens, a specific part of your script (you opening the door) runs. This is how indicators calculate their values and how strategies decide when to trade.

The Most Important Event Handler: OnBarUpdate()

For both indicators (tools that draw on charts) and strategies (automatic traders), the OnBarUpdate() method is where most of the action happens. This is the heart of your script. ¹

NinjaTrader usually runs the code inside OnBarUpdate():

- **When a price bar closes:** If you have a 5-minute chart, OnBarUpdate() will run every 5 minutes when the current bar finishes.
- **Optionally, on every incoming tick:** You can also set it up (using a property called Calculate) so that OnBarUpdate() runs every single time a new piece of price information (a "tick") comes in. This is much more frequent!

Inside OnBarUpdate(), your code will typically:

- Look at price data (like Close, High, Low, Open).
- Calculate indicator values.
- Check if your trading conditions are met.
- If it's a strategy, tell NinjaTrader to place buy or sell orders.

Other Important Event Handlers:

Besides OnBarUpdate(), NinjaScript has several other event handlers that give you more detailed control and let you react to different things happening in the platform ¹⁰:

- **OnMarketData(MarketDataEventArgs e):** This runs when real-time market data updates happen, like the last traded price changing, or the bid/ask prices changing. It's useful if you need to react to things even faster than OnBarUpdate() (e.g., at the

tick level). ¹⁰

- **OnMarketDepth(MarketDepthEventArgs e):** This runs when there are changes in the market depth (also called Level II data, which shows orders waiting to be filled at different price levels).
¹⁰
- **OnOrderUpdate(Order order, double limitPrice, double stopPrice, int quantity, int filled, double averageFillPrice, OrderState orderState, DateTime time, ErrorCode error, string nativeError):** This is super important for strategies. It runs whenever there's an update to an order that your script has sent (like if the order is accepted by the market, filled, or cancelled). It helps your strategy keep track of its orders. ¹
- **OnPositionUpdate(Position position, double averagePrice, int quantity, MarketPosition marketPosition, DateTime time):** Also crucial for strategies. This runs when your trading position changes (like when you enter a trade, exit a trade, or change the number of shares/contracts you're holding). It helps your strategy know what it currently owns or owes in the market.
¹
- **OnExecution(ExecutionEventArgs e):** This runs when one of your orders actually gets filled (an "execution"). It gives you details like the price you got and how many shares/contracts were filled.
- **OnRender(ChartControl chartControl, ChartScale chartScale):** This is mainly for indicators. It handles any custom drawing or pictures you want to put on the chart. It runs very often, whenever the chart needs to be redrawn. ¹⁰

Knowing which event handler to use for what job is a basic part of designing good NinjaScript.

3.2. State Management in NinjaScript: OnStateChange() and its States

This is probably the **most important part of how NinjaScript is structured**: its **state management system**. This is mainly controlled by a method called **OnStateChange()**.¹¹

NinjaTrader calls this OnStateChange() method whenever the "state" (like the current stage or condition) of your NinjaScript indicator or strategy changes. This method is THE place to:

- Set things up when your script first starts (initialization).
- Get any resources ready that your script needs (resource allocation).
- And, when your script is done, clean up those resources (teardown).¹¹

Using OnStateChange() and its different states correctly is super, super important for making sure your script works, doesn't crash (like with "null reference exceptions" – trying to use something that doesn't exist yet), and runs efficiently.

The OnStateChange() method gives your NinjaScript a kind of lifecycle, with different stages. If you try to set up resources or look at certain kinds of data *before* the right stage, your script can act weirdly or give you errors. This careful state management is a must-do.

An AI like Gemini, if you don't tell it specifically about this lifecycle, could easily write code that tries to do things at the wrong time. For example, a general AI wouldn't automatically know that you can only safely look at market data like Input or Instrument details *after* a state called State.DataLoaded has been reached, and not in an earlier state like State.SetDefaults.¹¹

So, when you ask Gemini to help you write NinjaScript code, you

often need to tell it about this state model. For example, you might say: "Gemini, create the OnStateChange method for my NinjaScript indicator. Inside the State.Configure part, I need to create a new empty list of numbers called myPriceLevels. Inside the State.DataLoaded part, I need to set up an SMA indicator using a setting called Period that the user can change."

Here are the key states that OnStateChange() deals with ¹¹:

- **State.SetDefaults:**

- **What it's for:** This is the very first state that gets called when your script starts. Its main job is to set the **default values** for any settings (public properties) that you want the user to be able to change from the NinjaTrader settings window (like the period for an indicator, colors, or on/off switches for features). ¹¹ It's also where you set up how your script should behave by default, using NinjaScript properties like Calculate (e.g., Calculate.OnBarClose to run on bar close, or Calculate.OnEachTick to run on every tick), IsOverlay (should the indicator draw on top of the price chart or in a separate panel below?), DisplayInDataBox (should its values show up in the data box on the chart?), etc. If your indicator draws lines or plots on the chart, you should also define their default look (like color, style, name) here using methods like AddPlot(), AddLine(), or AddHorizontalLine(). ¹¹
- **What you do here:** Give default values to your public settings. Set up your Plot and Line objects. Set script-level properties like Name (the script's name), Description, and Calculate.
- **Important things to remember:**
 - Keep this state "lean" – don't put a lot of complicated stuff here. Avoid doing heavy calculations or trying to use

other objects (especially things related to market data or other indicators) in `State.SetDefaults`.¹¹ Save that kind of logic for later states like `State.Configure` or `State.DataLoaded`.

- The public settings you define here are saved by `NinjaTrader` when someone saves their workspace or a chart template.¹¹
- If you have a public setting that shouldn't be saved (maybe it holds something that can't be easily saved and will be recreated each time, like a custom `Brush` for drawing), you should mark it with something called the `[XmlIgnore]` attribute.¹¹
- Similarly, if you have a public setting that other scripts might need to access, but you don't want it to show up in the user's settings window, you can mark it with ````.¹¹

- **State.Configure:**

- **What it's for:** This state runs after `State.SetDefaults` and after any settings the user changed have been applied. This is the right place to set up variables that your script uses (class-level variables) that aren't just fixed numbers (constants). It's also for getting ready any resources that *don't* need historical market data to be loaded yet.¹¹
- A very important job for this state is adding any **extra data series** your script might need using `AddDataSeries()`. For example, if your indicator needs to look at prices for a different stock or a different time frame (like a daily chart when you're on a 5-minute chart), you add that here.¹¹
- **What you do here:** Create new lists (e.g., `myList = new List<int>();`), reset variables that aren't constants, and add any extra `DataSeries`.

- **Important things to remember:**
 - You *must* make all your AddDataSeries() calls in State.Configure. Trying to do it in later states won't work and can cause errors. ¹¹
 - If you're building a strategy that will be tested many times with different settings in NinjaTrader's Strategy Analyzer, and you set a special property IsInstantiatedOnEachOptimizationIteration to false (meaning the same copy of your script is reused for many tests), then it's super important to reset any class-level variables back to their starting values here in State.Configure. This makes sure each test run starts fresh. ¹¹
- **State.DataLoaded:**
 - **What it's for:** This state means that all the historical price data for your main chart and any extra data series you added has been loaded by NinjaTrader and is ready to be used. This is the **earliest** state where it's safe to:
 - Access things related to the market data of your instrument (like Instrument.FullName to get the full name of what you're trading, BarsPeriod.BarsPeriodType to see the chart timeframe, or the Input, Close, High, Low, Open, Volume data series themselves).
 - Set up (initialize) indicators or other objects that need this market data to work (like an SMA or RSI). ¹¹
 - **What you do here:** Create instances of your indicators (e.g., mySMA = SMA(Close, Period);). Get information about the Instrument, BarsPeriod (chart timeframe), TradingHours, and use the historical price and volume data.
 - **Important things to remember:** If you try to use market

data or set up data-dependent indicators *before* `State.DataLoaded` (like in `SetDefaults` or `Configure`), you'll probably get errors (like a `NullReferenceException` because the thing you're trying to use doesn't exist yet, or your indicators might get incomplete data). ¹¹

- **State.Historical:**

- **What it's for:** This state starts when your script begins to process historical data (meaning, `OnBarUpdate()` starts getting called for all the old bars on your chart). This is the right state if your script needs to interact with things on the user interface (UI), like the `ChartControl` (the whole chart window), the `ChartPanel` (the area where prices are drawn), or `ChartBars` (the actual bars). You might do this if you want to read chart settings or add your own custom buttons or displays to the chart. ¹¹
- **What you do here:** Read chart properties (like `ChartControl.Properties.ChartBackground` to get the chart's background color). Add custom controls (like buttons or text boxes) to the chart panel.
- **Important things to remember:** UI-related things are not guaranteed to be ready or fully set up until this state. ¹¹

- **State.Realtime:**

- **What it's for:** This state happens when your script finishes processing all the historical data and switches over to processing live, real-time market data as it comes in.
- For strategies, there's a very important thing to do here: if you submitted an order while your strategy was running on historical data (backtesting), the `Order` object you got back was a "backtest" order. If that order is still active when you switch to real-time trading, the actual *live* order object in

NinjaTrader will be a different one. You need to update any Order objects you've stored in your script to point to the new live order. You do this using a special function:

GetRealtimeOrder(yourOldHistoricalOrderObject).¹¹

- **What you do here:** For strategies, update your stored Order objects from their historical versions to their real-time versions.
- **Important things to remember:** If you forget to update these order references, your strategy might not be able to manage its live orders correctly (like trying to cancel or change an order that it no longer has the right reference to).

¹¹

- **State.Terminated:**

- **What it's for:** This is the very last state. It's called when your NinjaScript is being removed from the chart or system, or when NinjaTrader itself is closing down. This is the place where you should clean up and release any custom, "unmanaged" resources that your script might have created and that NinjaTrader doesn't automatically clean up for you. Examples could be if you opened a file to write to, started a custom timer, or used some special memory that NinjaTrader doesn't know about.¹¹
- **What you do here:** Call the Dispose() method on any objects you created that have one (these are usually objects that implement something called IDisposable). Release any external resources.
- **Important things to remember:**
 - Make sure you only try to clean up (dispose of) resources if they were actually successfully created in the first place, to avoid errors.¹¹
 - NinjaTrader will automatically take care of cleaning up its

own objects (like the built-in indicators you used). You only need to worry about custom things *you* created that need special cleanup.

Table 3: NinjaScript OnStateChange() Lifecycle (The Life Stages of Your Script)

State Stage	What It's For	Key Things To Do / What's Allowed	Common Mistakes to Avoid
State.SetDefaults	First setup of user settings, script behaviors, and how plots/lines look by default. ¹¹	Set Name, Description, Calculate, IsOverlay. Define Input settings with default values. Use AddPlot(), AddLine(). ¹¹	Doing complex math. Trying to use market data (Input, Close, Instrument). Trying to set up indicators here. ¹¹
State.Configure	Set up resources that <i>don't</i> need market data. Add any extra data series. ¹¹	Use AddDataSeries(). Create lists (e.g., List<T>). Reset script variables if reusing the script for many strategy tests. ¹¹	Forgetting to add all your DataSeries here. Trying to use market data. Trying to set up data-dependent indicators. ¹¹
State.DataLoaded	Now you can use market data!	Set up indicators (e.g.,	Trying to do these things in

	Set up indicators and other things that need this data. ¹¹	SMA(Close, Period)). Access Instrument details, BarsPeriod, TradingHours, Input, Close, High, etc. ¹¹	SetDefaults or Configure. Thinking data is ready before this state. ¹¹
State.Historical	Interact with chart screen elements. Add your own custom buttons/displays to the chart. ¹¹	Access ChartControl, ChartPanel. Add custom screen elements (WPF controls). Read chart settings. ¹¹	Trying to use screen elements before this state. Doing really heavy calculations here that might freeze the screen (if you need to update the screen from other code parts, use something called Dispatcher).
State.Realtime	Switch from old data to live data. For strategies, update your order references. ¹¹	For strategies: entryOrder = GetRealtimeOrder(entryOrder); ¹¹	Forgetting to update your stored Order objects, which means you can't control your live orders properly. ¹¹

State.Terminated	Last step: Clean up any custom "unmanaged" resources you created. ¹¹	Call Dispose() on things that need it (like a StreamWriter if you were writing to a file, or a custom Timer). Set big objects to null to help free up memory. ¹¹	Not cleaning up resources, which can cause memory leaks (your program uses more and more memory over time). Trying to clean up things that weren't even created. ¹¹
------------------	---	---	--

This whole lifecycle is super important. Understanding it well is key not just for writing code yourself, but also for telling AI tools like Gemini what to do. The AI needs to know to put the code it generates into the correct state block for it to work right and not cause errors.

3.3. Architectural Schematic for a NinjaScript Project (A Picture in Your Mind)

To help you understand how a NinjaScript program is put together and how it talks to the main NinjaTrader platform, it's helpful to have a kind of mental map or diagram. We can't draw a picture here, but we can describe what would be in it. This mental map helps you see where different pieces of your code should go and how to ask an AI assistant like Gemini for help more clearly.

For example, if you know that a new setting for the user (an input parameter) belongs in a special section called Properties, and that its default value should be set in State.SetDefaults, you can give Gemini a very precise instruction: "Gemini, please add a C# property (a setting) called LookbackPeriod. It should hold a whole number (int). Make sure users can see it in the settings window with the

name 'Lookback Period' and put it in a group called 'Parameters'. Then, in the State.SetDefaults part of the OnStateChange method, set its starting value to 20."

Here are the main parts of this mental map:

1. **NinjaTrader Platform Core (The Main Engine):**
 - **Data Manager:** This part gets all the live market data from your connected data providers and builds the price bars for your charts.
 - **Order Execution Engine:** This part handles sending your buy/sell orders to your broker, changing them, or canceling them.
 - **Charting Engine:** This part draws the charts, the lines from your indicators, and any other drawing objects.
 - **NinjaScript Host Environment:** This part is responsible for loading your NinjaScript files, checking them for errors (compiling), and then running them.
2. **Your NinjaScript Class (Your Actual Code File, e.g., public class MyIndicator : Indicator or public class MyStrategy : Strategy):**
 - **#region Properties... #endregion (Your Settings Section):**
 - This is where you define any settings that the user can change from the NinjaTrader UI (like public int Period { get; set; }).
 - You also define your plots here (the lines your indicator will draw, e.g., AddPlot(Brushes.Blue, "MyPlot");).
 - And any fixed lines (e.g., AddHorizontalLine(0, Brushes.Gray, "ZeroLine");).
 - **OnStateChange() Method (The Lifecycle Manager):**
 - This contains all the if blocks for the different states:

State.SetDefaults, State.Configure, State.DataLoaded, State.Historical, State.Realtime, State.Terminated. This method controls how your script starts up, runs, and shuts down.

- **OnBarUpdate() Method (The Main Action Taker)** (or other main data event handlers like OnMarketData()):
 - This is where you put the main calculation logic for your indicators.
 - It's where you put the main decision-making logic for your strategies.
 - It uses price data (like Close, Open, or Input[barsAgo] to get data from past bars).
 - It can also use other indicators (like SMA(Close, Period)).
- **Order Event Handlers (For Strategies Only):**
 - OnOrderUpdate(): Keeps track of what's happening with your orders.
 - OnExecution(): Handles when your orders get filled.
 - OnPositionUpdate(): Keeps track of what trades you're currently in.
- **OnRender() Method (Mainly for Indicators that do custom drawing):**
 - This is where you put code if you want to draw things on the chart in a very custom way, using NinjaTrader's Draw tools or a more advanced library called SharpDX.
- **Helper Methods/Functions (Your Own Mini-Tools):**
 - These are extra, private methods you create inside your script to keep your code organized. You might put reusable pieces of logic or complicated calculations in them to make your main methods easier to read.
- **Class-Level Variables (Your Script's Memory):**

- These are private variables that your script uses to remember things between different events or bar updates, like storing references to indicators it's using, lists of data, etc.

3. **How Things Interact (The Flows):**

- **Data Flow:** The Data Manager in NinjaTrader sends bar data (Close, High, etc.) or tick-by-tick data to your script's `OnBarUpdate()` or `OnMarketData()` method.
- **Event Triggering:** The NinjaTrader Platform Core "triggers" (calls or runs) the different event handler methods in your NinjaScript when market events happen or when the script's state changes.
- **Indicator Interaction:** Your NinjaScript can create and use other indicators that are built into NinjaTrader or other custom ones (e.g., `mySMA = SMA(14);`).
- **Order Flow (For Strategies):** The logic in your `OnBarUpdate()` method might decide to place an order using commands like `EnterLong()`, `ExitShort()`, or `SubmitOrderUnmanaged()`. These requests go to NinjaTrader's Order Execution Engine. Information about what happens to those orders comes back to your script through `OnOrderUpdate()` and `OnExecution()`.
- **Drawing Flow (For Indicators):** When you use `AddPlot` in `State.SetDefaults`, you're defining lines that will be drawn automatically by NinjaTrader. If you use `Draw` methods in `OnBarUpdate` or do custom drawing in `OnRender`, your script is telling the Charting Engine to create visual elements on the chart.

This mental map helps you see that NinjaScript is very organized and event-driven. Understanding this structure is key to writing good

code and to telling AI tools how to generate code that fits correctly into this system.

3.4. Best Practices for Readability, Maintainability, and Performance

Following good coding habits is super important, especially when you're writing trading scripts, because mistakes or slow code can actually cost you money. These habits make sure your scripts are:

- **Readable:** Easy for you and others to understand.
- **Maintainable:** Easy to change or fix later.
- **Performant:** Run quickly and efficiently.

Making Your Code Easy to Read & Change (Readability & Maintainability):

- **Use Good Names:** Give your variables, methods (functions), and settings (properties) names that clearly describe what they are or do. For example, `fastMovingAveragePeriod` is much better than just `p1`.
- **Write Comments:** Add notes to your code, especially for parts that are complicated, calculations that aren't obvious, or to explain what a particular setting is for. ⁷ Code with good comments is also easier for AI tools to understand and help you with.
- **Organize Your Code:** Group related pieces of code together. Use **helper methods** (small, private functions inside your script) to package up bits of logic that you use more than once, or to break down big, complex tasks into smaller, easier-to-manage chunks. This makes your code easier to read, test, and fix.
- **Avoid "Magic Numbers":** Instead of using plain numbers directly in your code (like `if (value > 2.5)`), give them names. If it's a number that never changes, make it a "named constant" (like `private const double SignificanceThreshold = 2.5;`). If it's

something the user should be able to change, make it an input property (a setting). ¹¹ This makes your code clearer and easier to change later if needed.

Making Your Code Run Fast (Performance Considerations) 11:

How fast your NinjaScript runs is really, really important, especially the code inside `OnBarUpdate()`. This method can be called very often (on every tick or every time a bar closes). Slow code here can make NinjaTrader lag, make your trading signals late, cause you to miss trades, or even make the whole platform unstable.

- **Keep `OnBarUpdate()` Speedy:** Be very careful about how much work you do inside `OnBarUpdate()` and other methods that run frequently, like `OnMarketData()`. ¹¹
- **Store Indicator References (Cache Them):** If you use the value of an indicator (like an SMA) many times in `OnBarUpdate()` or in different parts of your script, it's faster to get a reference to that indicator *once* (usually in `State.DataLoaded` or `State.Historical`, for example: `mySma = SMA(Period);`) and then just use that stored reference (e.g., `mySma`) to get its value. Calling the indicator function itself (e.g., `SMA(Period)`) over and over again has a small extra cost each time. ¹¹
- **Don't Do the Same Calculation Twice (Avoid Redundant Calculations):** Make sure you only do calculations when you absolutely need to. If a value only needs to be figured out once per bar, don't recalculate it on every single tick if your Calculate setting is `OnEachTick`. You can use flags (true/false variables) or check a special property called `IsFirstTickOfBar` ¹³ to control how often parts of your `OnBarUpdate()` code run. ¹¹
- **Prepare Values for `OnRender()`:** The `OnRender()` method (for custom drawing) is called very, very often when the chart needs to be redrawn. Don't do complicated calculations directly inside `OnRender()`. Instead, figure out the values you need in `OnBarUpdate()` (which runs when new bar/tick data comes in)

and store them in class-level variables. Then, `OnRender()` can just use these already-calculated values for drawing. ¹¹

- **Limit What `OnRender()` Does:** When you're doing custom drawing in `OnRender()`, try to limit your calculations and drawing loops to only the bars that are currently visible on the chart. You can use `ChartBars.FromIndex` and `ChartBars.ToIndex` to find out which bars are visible. This stops your script from doing unnecessary work for data that's off-screen. ¹¹
- **DrawObjects vs. Custom Rendering:** NinjaTrader has easy-to-use Draw methods (like `Draw.Dot()`, `Draw.Text()`) for adding simple shapes and text to your chart. However, every time you call one of these, NinjaTrader creates a new drawing object. If your script needs to draw a LOT of objects very often (like a dot on every single bar for hundreds of bars), this can slow things down. For these situations, doing your own custom drawing using a library called SharpDX inside the `OnRender()` method can be much faster, but it's also more complicated to code. ¹¹
- **Never Use `Thread.Sleep()`:** Do NOT, under any circumstances, call `Thread.Sleep()` inside your NinjaScript event handlers, especially not in `OnBarUpdate()` or `OnMarketData()`. This will completely freeze the part of NinjaTrader that's running your script for that instrument. Charts might freeze, data might stop updating, and orders might not get processed. If you need to wait for a certain amount of time before doing something (like wait 5 seconds after a signal before placing a trade), use a `System.Windows.Forms.Timer` object instead. ¹¹
- **Be Careful with `barsAgo` Indexing:** When you get historical data using an index like `Close[barsAgo]` (e.g., `Close[1]` for the previous bar's close), this way of counting back from the current

bar is only guaranteed to be perfectly in sync during market data events like `OnBarUpdate()` and `OnMarketData()`. If you're in other event handlers (like `OnRender()`) or when a user clicks a button) that aren't directly triggered by new price updates, these `barsAgo` pointers might not point to what you expect based on the `CurrentBar`. In those other situations, it's safer to get historical data using methods like `indicator.GetValueAt(barIndex)` or `Close.GetValueAt(barIndex)`, where you give an exact bar number.¹¹

- **Use `IsFirstTickOfBar`:** If your script is set to `Calculate = Calculate.OnEachTick` or `Calculate.OnPriceChange` (meaning `OnBarUpdate()` runs very often), you can use an `if (IsFirstTickOfBar)` check. This ensures that certain parts of your logic only run once when a new bar actually forms, instead of on every single tiny price change within that bar.¹³ This can save a lot of computer power.

Managing Resources (Like Memory) 11:

Good resource management stops your script from using up too much computer memory (causing "memory leaks") and helps keep NinjaTrader stable.

- **Clean Up `IDisposable` Objects:** If your script creates objects that need special cleanup (these objects usually implement something called `IDisposable` – examples could be if you're writing to files with `StreamWriter`, using custom timers, or some advanced graphics objects that NinjaTrader doesn't manage for you), it's your script's job to make sure their `Dispose()` method is called when they're no longer needed. You usually do this in the `State.Terminated` block of `OnStateChange()`. Another way in C# is to use a `using` statement, which automatically calls `Dispose()` for you when the object is no longer in use.¹¹
- **Help the Garbage Collector:** For very large objects or big lists of things that your script doesn't need anymore, setting their

variable references to null can help the .NET garbage collector (the part of the system that cleans up unused memory) reclaim that memory sooner. This is especially good for scripts that run for a long time or that create and destroy a lot of big things. ¹¹

- **Freeze Custom WPF Brushes:** If you create your own custom WPF Brush objects (like SolidColorBrush for drawing with specific colors) for your plots or UI elements, you *must* "freeze" them by calling their Freeze() method before you use them. This is especially important if they might be used by different parts of the code running at the same time (different threads). The built-in brushes (like Brushes.Blue) are already frozen and safe to use. ¹¹

The connection between how fast your code runs (performance) and whether it has errors (error handling) is even more important in trading scripts. Slow code doesn't just make things annoying; it can actually make you lose money or miss good trades because your signals are too slow or the platform becomes unstable. ¹³ Similarly, an error that isn't handled properly (like a NullReferenceException when your script is trying to place an important order) can stop a trade from happening or mess up how your strategy manages its current trades. ¹⁰

NinjaTrader uses multiple "threads" (like different workers doing tasks at the same time) ¹¹, so you also have to be careful about potential problems if different parts of your code try to change the same thing at the exact same moment (these are called "race conditions"). You also need to be careful if you try to update things on the screen (UI) from code that isn't running on the main UI thread; for that, you might need to use something like Dispatcher.InvokeAsync. ¹¹

While Gemini can help you write code, it's still up to *you*, the developer, to make sure that code follows all these strict rules for performance and error handling that are needed for financial programs. You can ask Gemini to think about these things, for example: "Gemini, can you make this C# loop run faster inside a NinjaScript OnBarUpdate method?" or "Add some good checks to make sure the entryOrder object isn't null before I try to use its properties in this NinjaScript strategy."

3.5. Error Handling and Debugging Fundamentals in NinjaScript

Knowing how to handle errors well and how to find and fix bugs (debugging) are skills every NinjaScript developer absolutely needs.

Good Habits for Handling Errors ¹¹:

- **Check for null (Emptiness):** Always check if an object is null (meaning it hasn't been set up or doesn't exist) *before* you try to use any of its parts (members). This is especially true for objects that NinjaTrader gives you (like Instrument, ChartControl, Account) or objects that your own script is supposed to create (like instances of indicators). These things might not be ready or available at all times during your script's life, or in all situations. ¹¹

C#

```
// Example: Safely using ChartControl
```

```
if (ChartControl!= null && ChartPanel!= null)
```

```
{
```

```
    // Now it's safe to use ChartControl or ChartPanel things
```

```
}
```

- **Use try-catch Blocks Wisely:** Use try-catch blocks to handle errors that you think *might* happen in specific, small parts of your code (like if you're reading from a file, or doing math that

could try to divide by zero). Don't just wrap your entire OnBarUpdate() method (or other big methods) in one giant try-catch block. Catching errors too broadly can hide bugs, make it harder to find out what's really going wrong, and can even slow down your code a tiny bit. ¹¹

- **Cast Types Safely:** Sometimes you need to change an object from one type to another (this is called "casting"). For example, if you get a list of drawing objects from the chart, you might need to cast one to a HorizontalLine to use its specific properties. When you do this, it's safer to use the as keyword and then check if the result is null, instead of doing a direct cast. This stops your script from crashing with an InvalidCastException if the object isn't actually the type you thought it was. ¹¹

```
C#  
// Example: Safely casting a drawing tool  
foreach (NinjaTrader.Gui.Chart.IDrawingTool tool in DrawObjects)  
{  
    NinjaTrader.Gui.NinjaScript.HorizontalLine hLine = tool as  
    NinjaTrader.Gui.NinjaScript.HorizontalLine;  
    if (hLine != null) // Check if the cast worked  
    {  
        Print("Found HorizontalLine at price: " +  
        hLine.StartAnchor.Price);  
    }  
}
```

- **Comparing double Numbers (Floating-Point):** Be careful when you compare numbers with decimal points (double type) using the == (equals) operator. Because of how computers store these numbers, 100.00001 might not be exactly equal to

100.00000 in the computer's memory. When you're comparing prices or indicator values, it's better to check if they are "close enough" to each other by using a very small difference (like TickSize for prices, which is the smallest amount a price can change). NinjaTrader also has a built-in helper method called ApproxCompare() for this. ¹¹

C#

```
// Example: Comparing double values with a small tolerance
```

```
double price1 = 100.00001;
```

```
double price2 = 100.00000;
```

```
// if (price1 == price2) // This might be false!
```

```
// Check if the difference is less than the TickSize
```

```
if (Math.Abs(price1 - price2) <
```

```
Instrument.MasterInstrument.TickSize)
```

```
{
```

```
    // Consider them equal enough for trading purposes
```

```
}
```

```
// Or use NinjaTrader's built-in helper:
```

```
if (price1.ApproxCompare(price2) == 0) // ApproxCompare returns 0  
if they are "close enough"
```

```
{
```

```
    // Consider them equal
```

```
}
```

Ways to Find and Fix Bugs (Debugging Techniques) ²:

- **Print() Statements:** This is often the simplest and best way to debug in NinjaScript. The Print() method lets you send messages, variable values, or markers to the "NinjaScript Output" window in NinjaTrader (you can open this from the

NinjaTrader Control Center: New > NinjaScript Output). This helps you see what your code is doing and what your variables look like at different points in your script. ²

```
C#  
// Example: Using Print() for debugging  
Print("OnBarUpdate just started. CurrentBar is: " + CurrentBar + ", and  
Close price is: " + Close);  
if (myCondition == true)  
{  
    Print("My condition was true! The value of myVar is: " + myVar);  
}
```

- **NinjaTrader Strategy Analyzer:** If you're building a strategy, the Strategy Analyzer is a fantastic tool. It's not just for seeing how much money your strategy might have made in the past (backtesting), but also for understanding *why* it did what it did. It gives you detailed reports on all the trades. By adding Print() statements to your strategy, you can watch its decision-making process as it runs on historical data. ²
- **NinjaScript Editor Tools:** The editor built into NinjaTrader has some basic debugging tools, but they are not as powerful as what you get in a full IDE. ¹⁰
- **Visual Studio Debugging:** If you're using Visual Studio (or VS Code with the right.NET debugging tools) to write your NinjaScript, you can use the full power of the Visual Studio debugger. This is like having a microscope for your code! You can:
 - Set **breakpoints** (tell your code to pause at a certain line).
 - **Step through** your code line by line to see exactly what it's doing.
 - **Inspect variables** in real-time to see their values.

- Look at the **call stack** (which functions called which other functions to get to where you are).
- And much more! To debug a NinjaScript that's running inside NinjaTrader using Visual Studio, you usually "attach" the Visual Studio debugger to the NinjaTrader.exe program that's running.
- **Common Errors to Look Out For:**
 - **NullReferenceException:** This happens when you try to use an object that is null (it hasn't been created or set up yet). This is often because you tried to initialize something at the wrong time (like using an indicator in `State.SetDefaults` before the data is loaded) or forgot to initialize it at all. ¹⁰
 - **IndexOutOfRangeException:** This happens if you try to get an item from a list or array using an index number that's too big or too small (e.g., trying to get `Close` if there are only 100 bars on the chart, or `Close[-1]`). ¹⁰
 - **Logic errors:** These are tricky! Your code runs without crashing, but it just doesn't do what you want it to do. It might give the wrong results or make the wrong trading decisions. These kinds of errors need careful checking with `Print()` statements or a debugger to trace what your code is actually doing.

Understanding these common problems and how to find them is key to writing NinjaScript efficiently. AI tools can help you here by looking at error messages or suggesting why your code might be behaving strangely if you give the AI the problematic code and explain the situation.

4. Gemini AI: Capabilities for C# and NinjaScript Enhancement

Google's Gemini AI, especially through a service called **Gemini Code**

Assist, has a bunch of features that can really help people who are writing code in C# (which, remember, is the language NinjaScript is based on). Knowing what these features are is the first step to using AI to make your NinjaScript development better.

4.1. Overview of Gemini Code Assist and its Relevance to C#

Gemini Code Assist is like an AI-powered coding partner. It's designed to help programmers be more productive and write better quality code at all stages of building software. ³

The really good news for NinjaScript developers is that **C# is one of the programming languages that Google has specifically checked and confirmed that Gemini Code Assist works well with.** ⁵ This means the AI's abilities are well-tuned for the way C# code is written, its grammar, and the common ways people use it.

Here are some key features of Gemini Code Assist that are useful for C# and therefore for NinjaScript ³:

- **Real-time Code Suggestions and Completions:** As you type your C# code, Gemini can pop up suggestions (sometimes called "ghost text" because it appears lightly) to finish the line you're on or even a whole block of code.
- **Code Generation from Prompts:** You can tell Gemini, using normal English, to write entire pieces of C# code for you, like functions or even the basic structure of a class.
- **Error Detection and Correction:** Gemini can look at your C# code, spot potential mistakes, and suggest how to fix them.
- **Code Explanation:** It can take a piece of C# code (or even a whole file) that you give it and explain in plain English what that code does and how it works.
- **IDE Integration:** Gemini Code Assist is made to work right inside

popular code editors (IDEs) like Visual Studio Code and the JetBrains IDEs (such as IntelliJ or Rider). As we talked about, these are often better for NinjaScript development than the basic editor inside NinjaTrader.

4.2. Code Generation: From Snippets to Functions

One of the most helpful things Gemini can do is write C# code for you. This can be anything from finishing a half-typed line to creating entire methods (functions) or classes based on what you describe in your prompts. ⁵

When you're using Gemini in a supported IDE, you can usually get it to generate code in a few ways:

- **Inline Suggestions (Ghost Text):** As you're typing, Gemini will automatically suggest ways to complete your code. You can usually press the Tab key to accept the suggestion. ¹⁴
- **Comment-Based Prompts:** You can write a comment in your code describing what you want the next piece of code to do (e.g., `// Function to calculate the average of a list of numbers`). Then, you can trigger Gemini to write the actual C# function that does that. ⁶
- **Dedicated AI Chat/Prompt Interface:** Many IDEs that work with Gemini have a special chat window or a quick command menu (for example, in VS Code, you might type a command like `/generate`). You can type your request in plain English in this chat window to ask for code. ⁶

This can save you a lot of time, especially for:

- Writing "boilerplate" code (the basic, repetitive parts of a program, like setting up a new class or defining properties).
- Turning an algorithm (a set of steps to solve a problem) that

you've described in English into actual code.

- Creating small utility functions that are common in trading scripts (like a function to calculate how much you might win versus how much you might lose on a trade, or a function to make your output messages look nice).

4.3. Code Analysis and Understanding Existing NinjaScript

Often, when you're working on NinjaScript, you might be looking at code that already exists. Maybe it's an old script you wrote, an indicator you downloaded from someone else, or a complex strategy that another developer built. Gemini can be a really helpful tool for understanding these existing C# codebases. ³

You can usually select a chunk of code in your IDE, or give Gemini an entire file, and then ask it for an explanation (often through the AI chat window or by right-clicking on the code). For example, you could ask:

- "Explain this C# function and what its inputs (parameters) are."
- "Summarize what the code inside this OnBarUpdate method is trying to do."

Gemini can then give you information about what the variables are for, how the code flows from one step to the next, and what the overall goal of that code is. This is super useful if you're trying to figure out complicated calculations or parts of the NinjaScript system that you're not familiar with.

4.4. Debugging Assistance with Gemini

Finding and fixing bugs is a normal part of programming, and Gemini has features to help with this often tricky task. If you run into errors or your C# (and therefore NinjaScript) code isn't behaving the way

you expect, you can use Gemini like this ³:

- **Error Explanation:** If you get an error message from the compiler (when NinjaTrader tries to build your script) or a runtime error (an error that happens when your script is actually running), you can give that error message to Gemini. It might be able to explain what the error means and what usually causes it.
- **Bug Identification and Fix Suggestions:** You can give Gemini a piece of your C# code that's causing problems and describe what's going wrong (e.g., "This function is supposed to return true if the price is going up, but sometimes it returns false even when the price is clearly rising. Why?"). Gemini can look at the code and suggest where the bug might be or even offer a way to fix it. Some IDE integrations might even have "Quick Fix" options for errors that are highlighted, and these could be powered by Gemini to automatically apply a suggested change. ⁶ You might also be able to use a command like /fix (e.g., /fix potential NullPointerExceptions in this method). ⁶
- **Logical Troubleshooting:** Even if you don't have a specific error message, if your script just isn't doing what you want, you can try describing to Gemini what you *want* it to do and what it's *actually* doing. Sometimes, this can lead Gemini to suggest where your logic might be flawed. ¹⁵

4.5. Code Explanation and Documentation Generation

Besides just understanding existing code so you can change it or fix it, Gemini can also help you make your C# code easier for others (and your future self!) to understand by adding explanations and documentation. ⁶

- **Generating Comments:** You can select a C# function or a complicated piece of code and ask Gemini to write comments

that explain what it does (e.g., using a command like `/doc` this function).

- **Simplifying Code:** If you have a piece of C# code that's really messy or hard to read, you can ask Gemini to try and simplify it (e.g., `/simplify` this complicated if-else structure) while still making sure it does the same thing. This can make your code easier to read and maintain.

4.6. Local Codebase Awareness and Customization (A More Advanced Feature)

One thing that makes Gemini Code Assist really effective, especially for special frameworks like NinjaScript, is its ability to use information from your own computer and your current project.

- **Local Codebase Awareness:** Standard versions of Gemini Code Assist can often look at the files you currently have open in your IDE and the general structure of your project.⁵ This means if you're working on a NinjaScript indicator and you have other related .cs files open (maybe a base class you wrote, or other utility scripts), Gemini's suggestions can become much more relevant to *your* specific project. It can learn to use your project's own special ways of doing things and its unique parts.
- **Code Customization (For Big Teams/Companies - Enterprise Feature):** For large organizations that have a lot of their own private code, Gemini Code Assist Enterprise has a feature called "code customization." This lets the organization tell Gemini to look at and learn from their private code libraries (like a whole collection of their own secret NinjaScripts). Then, the AI will give code suggestions that are much more in line with that organization's specific coding style, their internal tools, and their own special ways of using NinjaScript.⁵

This "local codebase awareness" is really important for NinjaScript. A

general C# AI model, even if it's good at C#, might not know the deep, specific details of NinjaTrader's own special classes, methods, and settings (like `NinjaTrader.NinjaScript.Indicators.SMA`, `NinjaTrader.Core.Globals.Now`, `NinjaTrader.NinjaScript.OrderAction`, or the exact meaning of the different State values in `OnStateChange`). Without this specific knowledge, a general AI might "hallucinate" (make up) NinjaScript parts that don't exist or use them incorrectly. ¹⁷

However, when Gemini can "see" other NinjaScript files in your current project – maybe other indicators or strategies you've made, or custom base classes you use – its suggestions get much better and more accurate for NinjaScript. It can learn to use NinjaTrader-specific things correctly and understand any custom helper functions you've defined in your project.

For developers or teams that have a lot of existing NinjaScripts, the enterprise-level "code customization" feature could be amazing. By letting Gemini learn from a whole library of these scripts, it could create a version of Gemini that's much, much better at understanding the tiny details and special ways of doing things in the NinjaScript world. ¹⁶

What this means for you is that to get the most out of Gemini, you should try to keep your project well-organized. And if you have relevant reference scripts or even documentation (if it's in a format Gemini can understand within your IDE), having them as part of your project environment can really help Gemini give you better assistance.

Table 4: Gemini AI Features for NinjaScript Developers (What Gemini Can Do For You)

Gemini Feature	What It Does	How It Helps with NinjaScript	Example of How You Might Ask (Conceptual)
Code Completion	Suggests code (lightly visible "ghost text") as you type. ⁵	Auto-completes NinjaScript/C# words, variable names, function calls.	If you type Close. it might suggest ApproxCompare ().
Code Generation	Creates blocks of code or whole functions based on your plain English requests. ⁵	Helps you create OnBarUpdate logic, helper functions, settings for State.SetDefaults, rules for entering orders.	/generate a NinjaScript function to calculate Average True Range over 14 bars.
Code Explanation	Explains selected code in plain English. ³	Helps you understand complicated indicator math, strategy entry/exit rules, or the logic in OnStateChange in existing scripts.	Select a block of code, then ask: Explain this NinjaScript strategy's profit target logic.
Debugging Assistance	Helps find errors in your code and	Helps you figure out errors like	My NinjaScript is giving an error

	suggests ways to fix them. ³	NullReferenceException, IndexOutOfRangeException, mistakes in your OnBarUpdate logic, or problems with sending orders.	here [paste your code and the error message]. What's wrong? or /fix this error in my NinjaScript.
Documentation Generation	Adds comments or documentation to your code. ⁶	Can generate XML documentation comments (special comments for C#) for your public settings or summaries for complex functions.	Select a function, then ask: /doc this NinjaScript function.
Code Simplification / Refactoring	Suggests ways to make your code easier to read or run faster. ⁶	Can help rewrite deeply nested if statements in OnBarUpdate, or simplify complicated math.	Simplify this complex conditional logic in my NinjaScript indicator.
Local Codebase Awareness	Uses information from your current project/open files to give	Gives suggestions that fit with your own custom helper functions, base	(This happens automatically) Gemini uses your project information

	more relevant suggestions. ⁵	classes, or other NinjaScripts in the same project.	when it gives suggestions.
Code Customization (Enterprise)	Makes suggestions based on a big company's private collection of code. ⁵	Generates NinjaScript code that matches a team's established ways of doing things, their internal libraries, or their own special additions to the framework.	(This happens automatically after setup) Suggestions will reflect the private NinjaScript code it has learned from.

This table connects the general things Gemini can do with how they can be practically used when you're developing NinjaScript. It gives you a clear idea of how this AI helper can fit into your coding tasks.

5. Mastering Prompt Engineering for NinjaScript with Gemini

How well any Large Language Model (LLM) like Gemini works depends a LOT on how good your instructions (prompts) are.

"Prompt engineering" is like the art and science of writing these instructions in a way that gets the AI to give you the most accurate, helpful, and useful answers.

For NinjaScript development, this means you need to:

1. Know the general best ways to talk to LLMs.
2. Also think specifically about the C# language and the NinjaTrader system.

5.1. General LLM Prompting Best Practices (Good Ways to Talk to Any Smart AI)

There are some basic rules that help you write good prompts for LLMs like Gemini, and these apply to many tasks, including asking for help with code ¹⁵:

- **Be Clear and Specific:** Your requests should be direct, easy to understand, and not confusing. If you ask a vague question, you'll get a vague or unhelpful answer. Clearly tell the AI what you want it to do, what kind of result you expect, and any rules or limits it needs to follow. ¹⁸
 - Instead of: "Write some NinjaScript."
 - Try: "Write a NinjaScript indicator that draws a 10-period Simple Moving Average on the chart."
- **Give Context (Background Info):** Include important background information. For NinjaScript, this means:
 - Tell it you're using C#.
 - Mention you're working in the NinjaTrader 8 environment.
 - Use key NinjaScript words or class names (like Indicator, Strategy, OnBarUpdate, State.DataLoaded). ¹⁵ The more background the AI has, the better it can make its answer fit your needs.
- **Put Instructions First:** It's often best to put your main instruction or task at the very beginning of your prompt. Some people like to use special markers like ### or "" to clearly separate the instructions from other information or examples you might give, but this can depend on the specific AI tool you're using. ¹⁸
- **Say What Kind of Answer You Want (Output Format):** If you need the answer in a specific format (like a complete C# class, just a single function, a list of ideas, or code that includes special XML comments), tell the AI this directly in your prompt.

Giving an example of the format you want (this is called "few-shot prompting") can be very helpful. ¹⁸

- **Zero-Shot vs. Few-Shot Prompting:**
 - **Zero-Shot:** You ask a question or give an instruction directly without giving any examples first.
 - Example: "Explain what State.DataLoaded is used for in NinjaScript."
 - **Few-Shot:** You give one or more examples of the task and the kind of answer you want *before* you ask the AI to do a similar task.
 - Example: "Here's an example of an input property in NinjaScript: public int Length { get; set; }. Now, create an input property for a Brush Color and call it SignalColor." Few-shot prompts are often much better for teaching the AI about your preferred style, how you want things structured, and how to use specific NinjaScript parts correctly. ¹⁸
- **Try, Try Again (Iterative Refinement):** It's pretty rare to get the perfect answer from an AI on your very first try, especially if you're asking for something complicated.
 - Start with a basic prompt.
 - Look at what Gemini gives you.
 - If it's not quite right, change your prompt to be more specific, add more background info, or give clearer examples.
 - If your request is really big and complex, break it down into smaller, easier-to-manage sub-questions. ¹⁵
- **Tell the AI to Be Someone (Persona/Role Setting):** You can tell the AI to act like it's a certain kind of expert. This can change the style and content of its answer.

- Example: "You are an expert C# developer who specializes in NinjaTrader 8 NinjaScript. Please generate..." ²⁰
- **Ask for What You *Do* Want (Positive Framing):** Usually, it works better to tell the AI what you want it *to do*, rather than what you *don't* want it to do. ¹⁸
 - Better: "Generate code that uses Calculate.OnBarClose."
 - Less good: "Don't generate code that uses Calculate.OnEachTick."
- **Give It a Starting Hint for Code (Leading Words):** When you want Gemini to generate code, giving it the very beginning of a code structure or some "leading words" can help push it in the right direction. ¹⁸
 - Example: If you start your prompt with protected override void OnBarUpdate(), it signals to Gemini that the code it generates should fit inside this specific NinjaScript method.

A really important thing for getting good NinjaScript code from Gemini is **context priming**. "Priming" means getting the AI ready with the right background. While Gemini is good at C#, NinjaScript is a very *specialized* system built on top of C#. ¹ If you don't give it specific clues, Gemini might just write generic C# code that doesn't use the special NinjaScript tools or follow its specific ways of doing things. For example, it might try to create its own Bar object instead of using NinjaTrader's built-in data lists like Close or High.

People who have used other LLMs (like Claude) have found that giving the AI NinjaTrader 8 documentation and sample strategies helps it give much better results because it "primes" the AI with the right context. ¹⁷

So, to get good NinjaScript from Gemini, your prompts need to

"surround" the AI with the NinjaScript world. You do this by:

- Explicitly using NinjaScript words and terms.
- Mentioning NinjaScript class names (like Indicator, Strategy).
- Referring to its lifecycle methods (like OnStateChange, OnBarUpdate).
- And its special data structures (like Series<double>, or functions like SMA()).

5.2. Crafting Effective Prompts for NinjaScript Code Generation

When you're asking Gemini to write NinjaScript code for you, think about these specific things to include in your prompt:

- **What Type of Script?** Clearly say if you're building an Indicator, a Strategy, or an AddOn.
- **Which Key Methods?** Tell it which of the main NinjaScript methods you need help with (e.g., "Generate the OnStateChange method with logic for State.SetDefaults and State.DataLoaded," or "Provide the OnBarUpdate logic for...").
- **Refer to Data Series Correctly:** Use the exact names NinjaTrader uses for its price data: Close, High, Low, Open, Volume, Time. If it's an indicator, Input usually refers to the main data series (like the closing prices of the chart it's on). If you're using extra data series (like a different stock or timeframe), you'll refer to them using BarsArray[x] (where x is a number).
- **Name Your Indicators:** When you want to use indicators that are built into NinjaTrader or custom ones you've made, use their correct names and any settings they need (e.g., SMA(Close, 14), RSI(14, 3)).
- **Define Your Input Parameters (User Settings):**
 - List all the settings you want the user to be able to change.
 - Say what C# data type each setting should be (e.g., int

Period, double Offset, Brush UpTrendColor - Brush is for colors).

- Explain what each setting is for.
- Ask Gemini to make sure these settings show up in the NinjaTrader user interface (UI).
- **Describe the Logic Clearly:** Explain the conditions for any actions in detail.
 - Example: "If the Close price crosses above the SMA(Close, 20) AND the Volume is greater than the SMA(Volume, 20), then..."
- **For Strategies, Be Specific About Orders:**
 - Define the conditions for entering a trade and what kind of order to use (e.g., EnterLong("MyEntryName"), EnterShortLimit(DefaultQuantity, Close - 2 * TickSize, "MyShortEntryName")).
 - Define how to exit trades (e.g., using profit targets with SetProfitTarget(), stop losses with SetStopLoss(), or custom exit rules like ExitLong()).
- **Plotting and Drawing:**
 - Tell Gemini what you want to plot on the chart using AddPlot() in State.SetDefaults (e.g., what style the line should be, like PlotStyle.Line, and what color).
 - Or, if you need to draw things dynamically (that change on every bar), tell it about using Draw methods in OnBarUpdate or OnRender.

Example of a Good Generation Prompt:

"Please generate a complete NinjaScript indicator for NinjaTrader 8. Call it 'PriceChannel'. It needs two input parameters that the user can change:

1. ChannelPeriod (this should be an int, default value 20).
2. OffsetMultiplier (this should be a double, default value 2.0).

In the `State.SetDefaults` part of `OnStateChange`:

- Set the default for `ChannelPeriod` to 20.
- Set the default for `OffsetMultiplier` to 2.0.
- Set `IsOverlay` to true (so it draws on the main price chart).
- Add three plots:
 - 'UpperBand' (color Green)
 - 'MiddleBand' (color Orange)
 - 'LowerBand' (color Red) All of these should use `PlotStyle.Line`.

In the `State.DataLoaded` part of `OnStateChange`:

- Get an SMA (Simple Moving Average) indicator ready. It should use the Close price and the `ChannelPeriod` setting.
- Get an ATR (Average True Range) indicator ready. It should use the `ChannelPeriod` setting.

In the `OnBarUpdate` method:

- Calculate the MiddleBand: This should be the value of the `SMA(Close, ChannelPeriod)`.
- Calculate the UpperBand: This should be `MiddleBand + (ATR(ChannelPeriod) * OffsetMultiplier)`.
- Calculate the LowerBand: This should be `MiddleBand - (ATR(ChannelPeriod) * OffsetMultiplier)`.
- Make sure these calculated values are shown on their respective plots."

5.3. Prompting Strategies for Code Explanation, Debugging, and Optimization in NinjaScript

Gemini isn't just for writing new code. It can also help you understand, fix, and improve NinjaScript code that you already have.

- **Getting Code Explained:**

- You can paste the NinjaScript C# code snippet you're curious about, or if the code is in a file you have open in your IDE, you can just refer to a function or method in that file.
- **Prompt examples:**
 - "Explain this NinjaScript C# code. I especially want to understand the logic inside the OnOrderUpdate method and how it's trying to handle orders that only get partially filled."
 - "What is the purpose of the State.Configure block in this NinjaScript indicator code I'm showing you?" ⁶

- **Getting Help with Debugging (Fixing Bugs):**

- Give Gemini:
 1. The NinjaScript code that's causing problems.
 2. Any error messages you're getting from NinjaTrader (these could be compiler errors when you try to build the script, or runtime errors that show up in the NinjaScript Output window when the script is running).
 3. A clear description of what the script is doing wrong or what you expected it to do.
- **Prompt examples:**
 - "My NinjaScript strategy is supposed to place a stop loss order 30 ticks below the price where I entered the trade, but the SetStopLoss() command seems to be ignored. Here is my OnBarUpdate and OnExecution code: [Paste your code here]. The NinjaScript Output window shows this message: [Paste any relevant log message or error here]. What could be the problem?" ¹⁵
 - "I'm getting a System.ArgumentOutOfRangeException in my script called MyIndicator.cs on line 75. This happens

during OnBarUpdate. Here's the code around that line:
[Paste the code snippet here]. Can you help me figure out what's wrong?"

- **Getting Help with Optimization (Making Code Faster or Better):**
 - Show Gemini the part of your NinjaScript code that you think might be slow or inefficient.
 - **Prompt examples:**
 - "Please review this OnBarUpdate method from my NinjaScript indicator. It seems to be looping through historical data many times. Can you suggest ways to make it run faster, specifically for NinjaScript? Maybe by storing some results temporarily (caching) or by using NinjaScript's built-in tools more efficiently?" ¹⁵
 - "How can I rewrite this nested loop structure (a loop inside another loop) in my NinjaScript strategy so it doesn't have to do so much work on every bar update?"

5.4. Engineered AI Prompt Template for Initiating a New NinjaScript Project

To make it easier and faster to start a new NinjaScript project with Gemini, and to make sure you (and Gemini!) think about all the important NinjaScript structure from the beginning, you can use a **prompt template**. A template is like a fill-in-the-blanks form. It acts as a checklist for you and gives Gemini a really complete set of instructions. This makes it less likely that the AI will forget some of the special details about NinjaScript.

Here's a template you can adapt:

NinjaScript Project Initiation Prompt for Gemini

1. Core Directive (Your Main Goal for Gemini):

You are an expert C# and NinjaTrader NinjaScript developer. Your job is to generate the starting C# code for a new NinjaScript file.

2. NinjaScript Type:

- Type:
- Target NinjaTrader Version: NinjaTrader 8

3. Project Objective (What is this script supposed to do?):

- Brief Description: [e.g., "A momentum oscillator that measures how fast the price is changing."]
- Main Goal:

4. Key Input Parameters (Settings the user can change in NinjaTrader):

(List each parameter like this example)

- Example:
 - Name in code: RocPeriod (Data type: int): This is the number of bars to look back for the Rate of Change calculation. Default value: 14. Show in UI Group: "Parameters". Order in UI: 1.
 - Name in code: SmaPeriod (Data type: int): This is the period for smoothing the RoC value with an SMA. Default value: 5. Show in UI Group: "Parameters". Order in UI: 2.
 - Name in code: OscillatorColor (Data type: Brush): This is the color for the main oscillator line. Default value: Brushes.DodgerBlue. Show in UI Group: "Plots". Order in UI: 3.

5. Core Logic Requirements (Especially for OnBarUpdate or other key event handlers):

(Describe the main calculations or decisions)

- Example for an Indicator:
 - "In OnBarUpdate:
 1. Calculate Rate of Change (RoC): $(\text{Current Close} - \text{Close X bars ago}) / (\text{Close X bars ago}) * 100$. (Make sure to handle the case where Close X bars ago might be zero to avoid a division by zero error). Use RocPeriod for X.
 2. This RoC value needs to be stored in a temporary Series<double> (a list of numbers over time).
 3. Calculate an SMA of this RoC series, using SmaPeriod.
 4. The final smoothed SMA value should be assigned to the main plot of the indicator (usually called Value)."
- Example for a Strategy:
 - "In OnBarUpdate:
 1. If the MACD(12, 26, 9).Default line crosses above the MACD(12, 26, 9).Avg line, AND if the strategy is currently flat (not in any trade: Position.MarketPosition == MarketPosition.Flat), then enter a long trade: EnterLong(DefaultQuantity, "MACD_Cross_LE");.
 2. If the MACD(12, 26, 9).Default line crosses below the MACD(12, 26, 9).Avg line, AND if the strategy is currently in a long trade (Position.MarketPosition == MarketPosition.Long), then exit the long trade: ExitLong("MACD_Cross_LX");."

6. Practical Integration: Gemini in the NinjaScript Development Workflow (How to Actually Use Gemini When You Code)

Getting Gemini to work smoothly when you're building NinjaScript

involves a few steps:

1. Setting up the AI helper (Gemini Code Assist) in a good code editor (IDE).
2. Then, using its features smartly as you go through the different stages of creating and fixing your scripts.

This section will walk you through these steps and show you some examples of how it all works together.

6.1. Setting up Gemini Code Assist in Your Preferred IDE (e.g., VS Code)

We've already talked about how the code editor built into NinjaTrader is a bit basic and doesn't directly support AI helpers like Gemini Code Assist. So, the first thing you need to do is use a more powerful external code editor. **Visual Studio Code (VS Code)** is a very popular one that's free and works great with Gemini Code Assist.

Here's generally how you set it up:

1. **Install VS Code:** If you don't have it, go to the official VS Code website and download and install it.
2. **Install C# Support:** VS Code needs some extra tools to understand C# well (remember, NinjaScript files are C# files, ending in `.cs`). You'll need to install the official C# extension from Microsoft. It's often called something like "C# Dev Kit." This gives you cool features like smart code suggestions (IntelliSense) and tools for debugging.
- 3.
4. **Install the Gemini Code Assist Extension:**
 - Open VS Code.

- Look for the Extensions icon (it usually looks like some squares fitting together on the side panel, or you can press **Ctrl+Shift+X**).
- In the search bar, type "Gemini Code Assist" (or whatever the official name Google gives its VS Code extension is).
- Find it and click "Install."
-

5. **Configure Gemini Code Assist (Get it Ready):**

- After you install it, the Gemini extension might ask you to log in with a Google account that has permission to use Gemini services. Just follow the instructions on the screen.
- Check if there are any special settings for the extension. You might be able to turn on/off things like inline suggestions (where it suggests code as you type) or other options.

6. **Open Your NinjaScript Project/Files:**

- Your NinjaScript files are usually stored in a folder like **Documents\NinjaTrader 8\bin\Custom\NinjaScript** on your computer. Inside that, there are subfolders like **Indicators** and **Strategies**.
- In VS Code, you can open the specific **.cs** file you want to work on, or you can open the whole **Custom** folder as a "workspace."
- Make sure VS Code knows these are C# files and that the Gemini Code Assist features (like a chat panel or code suggestions) start working.

Once this is all set up, Gemini Code Assist will be ready to help you edit your NinjaScript **.cs** files right inside VS Code. **Important Tip:**

Remember to keep NinjaTrader running. It can also be helpful to have the built-in NinjaScript Editor open for the same script you're editing in VS Code. This helps NinjaTrader automatically recompile (re-check and rebuild) your script whenever you save changes in VS Code.

6.2. Workflow Example: Developing a New Indicator with Gemini's Help (Step-by-Step)

This example shows how you (the developer) and Gemini can work together like a team. You guide Gemini, and it helps you out.

- **Phase 1: Thinking and Asking (Conceptualization & Prompting):**
 - **Your Job (Developer):** First, you need to decide what your indicator should do. For example, let's say you want to create a custom "oscillator" (an indicator that usually moves up and down between certain levels). This oscillator will measure how far the current closing price is from a moving average, and then adjust that distance based on the Average True Range (ATR, which is a measure of how much the price has been moving around lately).
 - **Your Job (Developer):** Now, use the "Engineered AI Prompt Template" (we talked about this in Chapter 5) to write a very detailed first instruction for Gemini.
 - Tell it what settings the user should be able to change (inputs), like the period for the Moving Average, the period for the ATR, and maybe even what *type* of Moving Average to use.
 - Tell it what lines it should draw on the chart (plots), like the main oscillator line and maybe a zero line.

- Clearly explain the math for the main calculation that needs to happen in `OnBarUpdate()`.
- Crucially, tell it what should happen in the different `OnStateChange` states (for example, tell it to get the Moving Average and ATR indicators ready in `State.DataLoaded`).
-
- **Phase 2: Getting the Basic Structure (Initial Scaffolding and Boilerplate Generation):**
 - **Your Job (Developer):** Give your detailed prompt to Gemini Code Assist (you can usually do this through its chat window in VS Code).
 - **Gemini's Job:** Gemini will try to generate the basic starting C# code for your NinjaScript indicator. This should include:
 - The main class structure (e.g., `public class MyCustomOscillator : Indicator { ... }`).
 - The `Properties` section with all the user settings you asked for, correctly set up so they appear in the NinjaTrader UI.
 - The `OnStateChange` method with the basic `if` blocks for `State.SetDefaults`, `State.Configure`, and `State.DataLoaded`, and maybe some initial setup code in them.
 -
 - A basic outline for the `OnBarUpdate` method.
 - **Your Job (Developer):** Carefully look at the code Gemini gave you. Does the basic structure look right? Are the settings defined correctly? Is the `OnStateChange` setup

logical?

- **Phase 3: Writing the Main Logic with AI Help (Implementing Core Logic):**

- **Your Job (Developer):** Now, focus on the `OnBarUpdate` method. This is where the main calculation will happen. If the calculation is complicated, break it down into smaller steps.
- **Your Job (Developer):** You can use comments in your code to ask Gemini for help, or select a section and use the chat window. For example, inside `OnBarUpdate`, you could type a comment like this:
C#
`// Gemini, please use the 'MovingAverage' indicator and the 'AverageTrueRange' indicator`
 - `// (which we should have set up in State.DataLoaded).`
 - `// Now, calculate the oscillator value like this:`
 - `// oscillatorValue = (CurrentClosePrice - MovingAverageValue) / AverageTrueRangeValue;`
 - `// Then, assign this oscillatorValue to the main plot (Value).`
 - `// Also, please make sure to handle the case where AverageTrueRangeValue might be zero,`
 - `// so we don't get a "division by zero" error.`
 - Then, you'd trigger Gemini to generate the C# code for that comment.
 -
 - **Gemini's Job:** Gemini will try to write the C# code for the calculation.
 - **Your Job (Developer):** Look at the code Gemini generated.

- Is it correct?
- Does it do exactly what you asked?
- This is where you might go back and forth with Gemini. If it's not quite right, you can ask it to try again with a slightly different instruction, ask for other ways to do it, ask for clarification, or tell it to fix something. (e.g., "How would I use a different MA type here based on the `MAType` input I defined earlier?"). This "try, review, ask again" loop is really important.

■

- **Phase 4: Adding Plots, UI Things, and Alerts:**

- **Your Job (Developer):** Ask Gemini to make sure the `AddPlot()` calls in `State.SetDefaults` are correct for drawing your oscillator line and any other lines (like a zero line).
 - Example prompt: "Ensure `State.SetDefaults` includes `AddPlot(new Stroke(Brushes.Cyan, 2), PlotStyle.Line, "MyOscillator");` and also `AddHorizontalLine(0, Brushes.Gray, "ZeroLine");`."
- **Your Job (Developer):** If you need to do any really custom drawing in `OnRender()` (though for simple indicators, you often try to avoid this because `OnRender` runs so often and can slow things down if you do too much work there), you can ask Gemini for help with that. Just remember to keep performance in mind.
-
- **Your Job (Developer):** If you want alerts (like a pop-up

message or a sound): "Add an `Alert()` call in `OnBarUpdate` if the oscillator crosses above the 1.0 level. The alert message should say 'Oscillator High'."

- **Phase 5: Making the Code Better (Refining and Optimizing):**

- **Your Job (Developer):** If there are parts of the code Gemini wrote that you don't understand, ask Gemini to explain them.
- **Your Job (Developer):** Ask Gemini if it can make the code better or faster: "Review the `OnBarUpdate` method of this NinjaScript indicator. Are there any obvious parts that might be slow, or ways to write it more efficiently following NinjaScript best practices?" or you could use a command like `/simplify this section of code to make it easier to read`.
-
- **Gemini's Job:** Gemini will give you suggestions for how to improve the code.

- **Phase 6: Finding and Fixing Mistakes with Gemini's Help (Debugging):**

- **Your Job (Developer):** Try to compile (build) your script in NinjaTrader. If you get compiler errors (NinjaTrader tells you there's a mistake before it can even run the script), copy the error message and the part of the code that's causing it into Gemini's chat.
- **Your Job (Developer):** If your indicator loads on a chart but then causes a runtime error (an error that happens while it's running, often shown in the NinjaScript Output

window, like a `NullReferenceException`), give that error message, any other details (like a "stack trace" which shows where the error happened), and the relevant code to Gemini.

- Example prompt: "My NinjaScript indicator is crashing with a `NullReferenceException` in `OnBarUpdate` at line X. Here's the code around that line: [paste your code snippet here]. What could be causing this, and how can I fix it in NinjaScript?"

- **Gemini's Job:** Gemini will try to figure out what's wrong based on the error and your code. It might suggest things like "you forgot to set up this variable" or "you're trying to use data before it's ready in `State.DataLoaded`," and offer ways to fix it.

- **Phase 7: Testing and Making Sure It Works (Validation):**

- **Your Job (Developer):** This is a super important step that *only you* can do properly. You need to test your indicator thoroughly in NinjaTrader.
 - Try it on different stocks or futures.
 - Try it on different timeframes (like 1-minute charts, 1-hour charts, daily charts).
 - Watch it in different market conditions (when the market is calm, when it's crazy).
 - If you can, use the Strategy Analyzer (you might need to turn your indicator into a very simple strategy just for testing) or just watch it very carefully on your charts.
- **Super Crucial Point:** AI-generated code is a *helper*. It's not guaranteed to be perfect. **You** are the one who is

ultimately responsible for making sure the code is correct, safe, and does what you want.

This back-and-forth teamwork, where you tell the AI what to do and check its work, and the AI helps you write and understand code, is the best way to use Gemini. It's not like you just tell it once and it's done; it's an ongoing conversation.

6.3. Workflow Example: Understanding and Modifying Existing NinjaScript Strategies

Gemini can also be a huge help when you're working with NinjaScript strategies that someone else wrote, or that you downloaded, or that are just old and you don't quite remember how they work.

1. Load and Understand the Old Code:

- Open the `.cs` file of the existing strategy in VS Code (with Gemini Code Assist active).
- Find parts of the code that look complicated or that you don't understand. This could be:
 - Fancy logic for managing orders in `OnOrderUpdate()` or `OnPositionUpdate()`.
 - A complex condition for entering a trade in `OnBarUpdate()`.
- Select that code and ask Gemini: "Explain what this selected C# code block is doing. It's part of a NinjaScript strategy. What conditions is it checking before it tries to submit an order?" Gemini will try to explain the existing logic to you.
-

2. Decide What to Change (Identify Modification Areas):

- Once you understand the code better, you can decide what you want to change. For example, you might want to:
 - Change the rules for when to enter a trade.
 - Add a new way to set a stop-loss that changes with the market.
 - Add another indicator as a filter (e.g., only take trades if this other indicator agrees).

3. Ask Gemini to Help with the Changes (Prompt for Modifications):

- Give Gemini the part of the code you want to change and a very clear instruction about what the change should be.
- **Example prompt:** "In the following `OnBarUpdate` method, I want to add another condition before entering a long trade. Right now, it enters if `SMA(Close, 10)` is greater than `SMA(Close, 30)`. I want to change it so that it only enters if BOTH that condition is true AND the `ADX(14)` indicator is greater than 25. Here is the current method:."
- Gemini will try to generate the modified code for you.

4. Review, Refine, and Put It Together (Refactor and Integrate):

- Carefully look at the changes Gemini suggested.
- You can also ask Gemini to help you clean up parts of the older code to make it easier to read or to follow current NinjaScript best practices. For example: "Can you rewrite this part of the strategy to use a separate helper function for checking all the entry conditions?"

- Very carefully put the new or changed code into the strategy, making sure it doesn't mess up other parts of the strategy.

5. Test, Test, Test! (Debug and Test):

- Try to compile the changed strategy in NinjaTrader. If you get any new errors, use Gemini to help you figure them out, just like in the previous example.
- This is super important for strategies:
 - **Backtest it thoroughly** in NinjaTrader's Strategy Analyzer. This means running it on old market data to see how it would have performed.
 - **Forward-test it (paper trading):** Run the modified strategy on a demo (fake money) account with live market data for a while to see how it behaves in real current market conditions.

6.4. Programmatic Access via Gemini SDK (This is for Super Advanced Users!)

For really advanced stuff, like if you wanted to build your own custom software tools that talk to Gemini directly (outside of your code editor), Google provides something called an **SDK (Software Development Kit)** for its AI services, including Gemini. There's a C# SDK. This lets a C# program you write make calls directly to the Gemini API (Application Programming Interface – how programs talk to each other).

- **What could you do with this?** Maybe you could build a custom tool where you type a high-level description of a trading strategy in plain English, and your tool uses the Gemini API (via the SDK) to try and turn that description into

NinjaScript C# code, and then automatically save it as a `.cs` file in the right NinjaTrader folder.

- **Things to think about:** This is much more complicated. You'd need to know a lot more about programming, how to keep your API keys (like passwords for the API) safe, how to handle the data that the API sends back and forth, and how to deal with limits on how often you can call the API (rate limits). This is usually for developers who are creating *tools to help with* NinjaScript development, rather than for just writing the scripts themselves.
-

The main idea for using Gemini well is this "human-AI team" approach. The AI can write code quickly or give you analysis, but *your* knowledge of trading, your understanding of how NinjaScript really works, and your good judgment are absolutely essential for:

- Telling the AI what to do.
- Checking if what the AI gives you is correct.
- Making sure the final script is solid, works right, and actually helps you with your trading goals.

7. Illustrative Examples and Use Cases (Seeing Gemini in Action)

This section will give you some real, concrete examples of how you can use Gemini Code Assist for common tasks when you're developing NinjaScript. For each example, we'll include:

- **Objective:** What's the goal of this task?
- **Engineered Prompt:** A well-written instruction for Gemini,

using the ideas from Chapter 5.

- **Anticipated Gemini-Generated C# NinjaScript Output:** What a good AI, given good instructions, *might* produce. (This is just an example, the actual output can vary).
- **Explanation and Analysis:** We'll talk about the code Gemini might generate, whether it follows NinjaScript best practices, and how it fits with the architectural ideas we discussed earlier.

These examples are meant to show you how Gemini can be used in practice to help write, explain, fix, and improve NinjaScript code, and we'll point out how they relate to the NinjaScript structure concepts.

7.1. Example 1: Generating a Moving Average Crossover Strategy Stub (The Basic Framework)

- **Objective:** Create the basic C# starting point (a "stub" or framework) for a NinjaScript strategy. This strategy should:
 - Enter a long trade (buy) when a fast-moving average (MA) crosses above a slow-moving average.
 - Enter a short trade (sell) when the fast MA crosses below the slow MA. This first version will just focus on these entry rules and drawing the MAs on the chart.
- **Engineered Prompt:** (We'll use the template from Section 5.4)

NinjaScript Project Initiation Prompt for Gemini

1. Core Directive:

You are an expert C# and NinjaTrader NinjaScript developer. Your task is to generate the initial C# code for a new NinjaScript file.

2. NinjaScript Type:

- Type: Strategy
- Target NinjaTrader Version: NinjaTrader 8

● 3. Project Objective:

- Brief Description: "A simple moving average crossover strategy."
- Main Goal: "Generate long entry signals when a fast MA crosses above a slow MA, and short entry signals when a fast MA crosses below a slow MA."

● 4. Key Input Parameters (Settings the user can change in NinjaTrader UI):

- **FastMAPeriod** (data type: **int**): Period for the fast moving average. Default: 10. UI Group: "Parameters". UI Order: 1.
- **SlowMAPeriod** (data type: **int**): Period for the slow moving average. Default: 20. UI Group: "Parameters". UI Order: 2.
- **FastMAColor** (data type: **Brush**): Color for the fast MA plot. Default: **Brushes.Green**. UI Group: "Plots". UI Order: 3.
- **SlowMAColor** (data type: **Brush**): Color for the slow MA plot. Default: **Brushes.Red**. UI Group: "Plots". UI Order: 4.

● 5. Core Logic Requirements (Especially for

OnBarUpdate):

- "In `OnBarUpdate`:
 - If the fast MA crosses above the slow MA, AND the current market position is flat (not in a trade), then `EnterLong(DefaultQuantity, "FastCrossAboveSlowLE");`.
 - If the fast MA crosses below the slow MA, AND the current market position is flat, then `EnterShort(DefaultQuantity, "FastCrossBelowSlowSE");`
- **6. Output & Visualization Requirements (Plots, Drawings, Alerts):**
 - "In `State.SetDefaults`, set `IsOverlay` to `true` (so it draws on the main price chart).
 - Add two plots: one for the fast MA using `FastMAColor`, and one for the slow MA using `SlowMAColor`. Both should use `PlotStyle.Line`."
- **7. NinjaScript Structure & State Management (`OnStateChange`):**
 - `State.SetDefaults`: Initialize all input parameters with their default values. Set script properties like `IsOverlay = true` and `Calculate = Calculate.OnBarClose`. Define the plots for FastMA and SlowMA.
 -
 - `State.Configure`: (No specific actions needed for this basic example).
 -

- `State.DataLoaded`: Initialize the SMA indicators for the FastMA and SlowMA. They should use their respective period inputs and the `Close` price series.
-
- `State.Terminated`: (No specific actions needed for this basic example).
-

● 8. Code Style & Comments:

- Generate well-formatted C# code.
- Include brief comments for the input parameters and the main logic in `OnBarUpdate`.
- Use standard `using` directives (these are like telling C# which toolboxes to use) for things like `NinjaTrader.NinjaScript`, `NinjaTrader.NinjaScript.Indicators`, `System.ComponentModel.DataAnnotations`, etc.

● 9. Error Handling (Basic):

- In `OnBarUpdate`, include a check like `if (CurrentBar < SlowMAPeriod) return;` to make sure you don't try to do calculations before there's enough price data available for the slowest MA.
- Please generate the complete NinjaScript C# code file content based on these specifications.
- **Anticipated Gemini-Generated C# NinjaScript Output (This is just a part of what it might look like):**

C#

- `// #region Using declarations (These are like importing toolboxes)`
- `// using System;`
- `// using System.Collections.Generic;`
- `// using System.ComponentModel;`
- `// using System.ComponentModel.DataAnnotations; // For UI attributes`
- `// using System.Linq;`
- `// using System.Text;`
- `// using System.Threading.Tasks;`
- `// using System.Windows.Media; // For Brushes (colors)`
- `// using System.Xml.Serialization; // For saving/loading settings`
- `// using NinjaTrader.Core.Globals;`
- `// using NinjaTrader.Gui;`
- `// using NinjaTrader.Gui.Chart;`
- `// using NinjaTrader.Gui.NinjaScript;`
- `// using NinjaTrader.Data;`
- `// using NinjaTrader.NinjaScript;`
- `// using NinjaTrader.NinjaScript.DrawingTools;`
- `// using NinjaTrader.NinjaScript.Indicators; // Where SMA lives`
- `// #endregion`
-
- `// This namespace (like a folder for code) holds Strategies and is required.`
- `namespace NinjaTrader.NinjaScript.Strategies`
- `{`
- `public class MACrossoverStrategy : Strategy // Our strategy inherits from NinjaTrader's Strategy class`
- `{`
- `// Variables to hold our SMA indicators`

- private SMA FastMA;
- private SMA SlowMA;
-
- protected override void OnStateChange() // The lifecycle manager!
- {
- if (State == State.SetDefaults) // First stage: set defaults
- {
- Description = @"Simple Moving
- Average Crossover Strategy";
- Name = @"MACrossoverStrategy";
- Calculate = Calculate.OnBarClose; //
- Run on bar close
- EntriesPerDirection = 1; // Only one entry
- per direction at a time
- EntryHandling =
- EntryHandling.AllEntries;
- IsExitOnSessionCloseStrategy = true; // Exit at
- end of session
- ExitOnSessionCloseSeconds = 30;
- IsFillLimitOnTouch = false;
- MaximumBarsLookBack =
- MaximumBarsLookBack.TwoHundredFiftySix;
- OrderFillResolution =
- OrderFillResolution.Standard;
- Slippage = 0;
- StartBehavior =
- StartBehavior.WaitUntilFlat; // Wait until no trades are on
- TimeInForce = TimeInForce.Gtc; //
- Good 'til Canceled

- // Initialize Indicators (as requested)
- FastMA = SMA(FastMAPeriod); // Uses the
FastMAPeriod input
- SlowMA = SMA(SlowMAPeriod); // Uses the
SlowMAPeriod input
-
- // Make sure the plots use the colors (sometimes
needed explicitly)
- FastMA.Plots.Brush = FastMAColor; // Plots is the first
(and usually only) plot of an SMA
- SlowMA.Plots.Brush = SlowMAColor;
- // Add the indicators to the chart so they are drawn
- AddChartIndicator(FastMA);
- AddChartIndicator(SlowMA);
- }
- }
-
- protected override void OnBarUpdate() // Runs on every bar
(or tick)
- {
- if (BarsInProgress!= 0) // For
multi-series/multi-instrument, only process primary
- return;
-
- // Ensure enough bars have loaded for the slowest MA (as
requested)
- if (CurrentBar < Math.Max(FastMAPeriod, SlowMAPeriod))
- return;
-
- // Plot MAs (assign their current values to the plots we

added)

- // For strategies, you often assign to Values[plotIndex]
- // Values corresponds to the first AddPlot call ("FastMA")
- // Values[1] corresponds to the second AddPlot call ("SlowMA")
- // However, since we used AddChartIndicator, they plot themselves.
- // If we wanted to plot their values on THIS script's plots, we'd do:
- // Values = FastMA;
- // Values[1] = SlowMA;
- // For simplicity, AddChartIndicator is fine here.
-
- // Entry Logic (as requested)
- if (Position.MarketPosition == MarketPosition.Flat) // Only if we are not in a trade
- {
- // Check for fast MA crossing above slow MA
- if (CrossAbove(FastMA, SlowMA, 1)) // The '1' means check the most recent crossover
- {
- EnterLong(DefaultQuantity, "FastCrossAboveSlowLE"); // LE for Long Entry
- }
- // Check for fast MA crossing below slow MA
- else if (CrossBelow(FastMA, SlowMA, 1))
- {
- EnterShort(DefaultQuantity, "FastCrossBelowSlowSE"); // SE for Short Entry
- }

- }
- }
-
- #region Properties (User-changeable settings)
- // These attributes tell NinjaTrader how to show them in the UI
-
-
-
- public int FastMAPeriod { get; set; } // Property for Fast MA Period
-
-
-
-
- public int SlowMAPeriod { get; set; } // Property for Slow MA Period
-
- // For Brush properties, we need to handle saving/loading them specially
- [XmlIgnore] // Don't save the Brush object directly
-
- public Brush FastMAColor { get; set; }
-
- // This string version IS saved, and then converted back to a Brush
-
- public string FastMAColorSerializable
- {
- get { return Serialize.BrushToString(FastMAColor); }

- set { FastMAColor = Serialize.StringToBrush(value); }
- }
-
- [XmlIgnore] // Same for the slow MA color
-
- public Brush SlowMAColor { get; set; }
-
-
- public string SlowMAColorSerializable
- {
- get { return Serialize.BrushToString(SlowMAColor); }
- set { SlowMAColor = Serialize.StringToBrush(value); }
- }
- #endregion
- }
- }

-
- **Explanation and Analysis (What did Gemini do well?):**

- The AI (hopefully!) created a C# class called **MACrossoverStrategy** that "inherits" from NinjaTrader's base **Strategy** class. This is the correct starting point.
- It included the input parameters (**FastMAPeriod**, **SlowMAPeriod**, and the colors) in a special section called **#region Properties**. It also (hopefully) added special "attributes" (like **,, , [XmlIgnore]**) to these properties. These attributes tell NinjaTrader how to display them in the settings window and how to save/load them. This is good NinjaScript practice.

-
- **State Management is Key:** The `OnStateChange` method is structured correctly.
 - In `State.SetDefaults`, it set the default values for your inputs, set `IsOverlay` to true (so it draws on the main price chart), and set `Calculate` (e.g., to `Calculate.OnBarClose` as requested). It also added the plots for the MAs using `AddPlot()`. The way it handles the `Brush` (color) properties with an `XmlIgnore` attribute and a separate "Serializable" string version is the correct way to make sure colors can be saved and loaded properly by NinjaTrader.
 -
 - In `State.DataLoaded`, it correctly initialized the `SMA` indicators (one for `FastMA` and one for `SlowMA`) using the period settings you provided. It also assigned them to be drawn on the chart using `AddChartIndicator()`. This is the right place to set up indicators that need price data.
 -
- **Core Logic in `OnBarUpdate`:**
 - It included a "guard clause" like `if (CurrentBar < Math.Max(FastMAPeriod, SlowMAPeriod)) return;`. This is important to stop the code from trying to calculate the MAs before enough price bars have loaded on the chart, which would cause errors.
 - The main entry logic uses NinjaTrader's built-in `CrossAbove()` and `CrossBelow()` functions to detect when the MAs cross. This is a good, efficient way to do it.

- It checks `if (Position.MarketPosition == MarketPosition.Flat)` before trying to enter a trade, which means it will only try to enter if it's not already in a trade (as requested by the `EntriesPerDirection = 1` and the prompt).
- It uses `EnterLong()` and `EnterShort()` to place the trades.
- **Architectural Alignment (Does it fit the NinjaScript way?):** Yes, this code generally follows the typical structure for a NinjaScript strategy: settings in Properties, setup driven by states in `OnStateChange`, and the main trading logic in `OnBarUpdate`.
- **What You (the Developer) Would Do Next:** This is just a starting point! A real strategy needs more. You'd need to add:
 - **Exit logic:** How and when do you get out of a trade? (e.g., using `ExitLong()`, `ExitShort()`, or setting profit targets and stop losses with `SetProfitTarget()` and `SetStopLoss()`).
 - **Position sizing:** How many shares or contracts should it trade? (Right now, it just uses `DefaultQuantity`).
 - **More risk management:** Maybe other rules to control risk.
 - **Lots and lots of testing** in the Strategy Analyzer!

7.2. Example 2: Explaining a Complex Custom Indicator's `OnBarUpdate()` Logic

- **Objective:** You've found an old indicator, or someone gave you one, and the `OnBarUpdate()` method looks really complicated

and has no comments. You want Gemini to help you understand what it's doing.

- **Code Snippet (A made-up example of a complex, uncommented `OnBarUpdate`):**

C#

// Let's imagine this is a complicated OnBarUpdate from an old indicator.

- // 'Input' is the main price data for this indicator (like Close price).
- // 'Typical' is usually (High + Low + Close) / 3.
- // Let's pretend Period1 = 10 and Period2 = 5 are input settings.
-
- protected override void OnBarUpdate()
- {
- if (CurrentBar < Period1) return; // Period1 is used for val1
-
- double val1 = 0;
- for (int i = 0; i < Period1; i++) // Loop 'Period1' times
- {
- val1 += (High[i] - Low[i]); // Add up the range (High - Low) of the last 'Period1' bars
- }
- val1 = val1 / Period1; // Get the average range over Period1 bars
-
- double val2 = SMA(Typical, Period2); // Get the 5-period SMA of the Typical price
-
- // Now the tricky part:

- if (Input > (val2 + (val1 * 0.5))) // If current Input is above (SMA + half the average range)
- {
- Value = 1; // Set the main plot value to 1
- PlotBrushes = Brushes.Green; // Make the plot green
- }
- else if (Input < (val2 - (val1 * 0.5))) // If current Input is below (SMA - half the average range)
- {
- Value = -1; // Set the main plot value to -1
- PlotBrushes = Brushes.Red; // Make the plot red
- }
- else // Otherwise (if it's in between)
- {
- Value = 0; // Set the main plot value to 0
- PlotBrushes = Brushes.Gray; // Make the plot gray
- }
- }

- **Engineered Prompt:**

"Please explain the following NinjaScript C# code from an `OnBarUpdate()` method of an indicator. Specifically, I want to know:

- What does the variable `val1` represent after it's calculated?
- What does the variable `val2` represent?
- Under what conditions is the main plot of the indicator (`Value`) set to 1, -1, or 0?

- How does the color of the plot (**PlotBrushes**) change based on these conditions?
- Assume **Period1** and **Period2** are integer input parameters for this indicator. **Input** refers to the primary data series (like the closing price). Here is the code:"

C#

// [Paste the code snippet from above here]

- protected override void OnBarUpdate()
- {
- if (CurrentBar < Period1) return;
-
- double val1 = 0;
- for (int i = 0; i < Period1; i++)
- {
- val1 += (High[i] - Low[i]);
- }
- val1 = val1 / Period1;
-
- double val2 = SMA(Typical, Period2);
-
- if (Input > (val2 + (val1 * 0.5)))
- {
- Value = 1;
- PlotBrushes = Brushes.Green;
- }
- else if (Input < (val2 - (val1 * 0.5)))
- {
- Value = -1;
- PlotBrushes = Brushes.Red;

- }
- else
- {
- Value = 0;
- PlotBrushes = Brushes.Gray;
- }
- }
-

- **Anticipated Gemini-Generated Explanation (What Gemini might say):**

"This NinjaScript `OnBarUpdate()` code seems to be calculating a custom oscillator or a state indicator that changes based on price action relative to a dynamic channel. Here's a breakdown:

- **val1**: This variable calculates the **average range** (the difference between the High and Low price) of the price bars over the last `Period1` number of bars. It's essentially a measure of recent price volatility or how much the price has been swinging up and down.
- **val2**: This variable calculates a **Simple Moving Average (SMA)** of the 'Typical Price' (which is usually $(\text{High} + \text{Low} + \text{Close}) / 3$) over the last `Period2` number of bars. This `val2` acts as a kind of short-term central baseline or average price.
- **Conditions for Value (the main plot value)**: The indicator then sets its main output value (`Value`) based

on where the current **Input** price (e.g., the current bar's closing price) is in relation to a channel created around **val2**. This channel's width is determined by **val1**.

- **Value is set to 1 (and PlotBrushes becomes Green):** This happens if the current **Input** is greater than **val2** (the SMA of Typical Price) PLUS half of **val1** (half of the average range). This likely signals a strong bullish (upward) condition or breakout, where the price has moved significantly above its recent baseline, and the breakout is considered strong relative to recent volatility.
- **Value is set to -1 (and PlotBrushes becomes Red):** This happens if the current **Input** is less than **val2** MINUS half of **val1**. This likely signals a strong bearish (downward) condition or breakdown.
- **Value is set to 0 (and PlotBrushes becomes Gray):** This happens if the current **Input** is within the channel (i.e., not above $\text{val2} + 0.5 * \text{val1}$ and not below $\text{val2} - 0.5 * \text{val1}$). This indicates a neutral or sideways market condition according to this indicator's logic.
- **PlotBrushes (Plot Color):** As explained above, the color of the first plot of this indicator changes to Green for a bullish signal (Value=1), Red for a bearish signal (Value=-1), and Gray for a neutral signal (Value=0)."

- **Explanation and Analysis (How did Gemini do?):**

- Gemini's explanation (hopefully!) successfully broke down

the complex code. It should have correctly identified what `val1` (average range/volatility) and `val2` (SMA of Typical Price) are calculating.

- It should have accurately figured out the conditions for the different states (1, -1, 0) by seeing that the code is creating a sort of dynamic channel around the SMA (`val2`) using the average range (`val1`) as a measure for the channel's width.
- This shows that Gemini can be really useful for understanding C# code, math operations, and even common trading ideas (like volatility bands or price relative to a moving average). This makes it a great tool if you need to figure out what an unfamiliar NinjaScript is doing.
-

7.3. Example 3: Debugging a Common `NullReferenceException` in NinjaScript with Gemini

- **Objective:** You're getting a `NullReferenceException` error, which is very common in programming. It means you're trying to use something (an object) that hasn't been properly created or set up yet (it's "null" or empty). This often happens in NinjaScript if you try to use an indicator before it's ready. We want Gemini to help us find why.
- **Problematic Code Snippet (A made-up example that would cause this error):**

```
C#  
// Inside a NinjaScript Indicator class
```

- public class MyNullRefIndicator : Indicator
- {
- private SMA mySma; // We declare a variable to hold an SMA, but where do we create it?
- // Let's assume 'Period' is an input property, e.g., public int Period { get; set; } = 14;
-
- protected override void OnStateChange()
- {
- if (State == State.SetDefaults)
- {
- Description = @"This indicator will probably cause a NullReferenceException.";
- Name = @"MyNullRefIndicator";
- Calculate = Calculate.OnBarClose;
- IsOverlay = true;
- Period = 14; // Setting default for our input property
-
- // COMMON MISTAKE: Trying to initialize the SMA here!
- // mySma = SMA(Period); // If we did this here, it's WRONG!
- // Why? Because in State.SetDefaults:
- // 1. 'Period' might not have its final user-set value yet.
- // 2. More importantly, the 'Input' price data series (like Close)
- // that the SMA needs is NOT YET AVAILABLE.
- }
- else if (State == State.Configure)

- {
- // Still too early to initialize 'mySma' if it needs price data.
- }
- else if (State == State.DataLoaded)
- {
- // THIS IS THE CORRECT PLACE TO INITIALIZE 'mySma'
- // mySma = SMA(Period); // If we put it here, it would be
- okay.
- }
- }
-
- protected override void OnBarUpdate()
- {
- // Let's assume 'Period' is defined and has a value (e.g., 14)
- if (CurrentBar < Period) // A common guard clause
- return;
-
- // THE ERROR WILL HAPPEN HERE if 'mySma' was never
- initialized
- // or was initialized incorrectly (so it's still null).
- // If mySma is null, trying to get a value from it (mySma) will
- crash.
- Values = mySma; // Values is the first plot of our indicator.
-
- // A safer way would be:
- // if (mySma != null)
- // {
- // Values = mySma;
- // }
- // else

- `// {`
- `// Print("Error: mySma is null! Check initialization in`
`OnStateChange -> State.DataLoaded.");`
- `// }`
- `}`
-
- `// To make 'Period' an input, it should be a property like this:`
- `#region Properties`
-
-
-
- `public int Period { get; set; }`
- `#endregion`
- `}`
-

- **Engineered Prompt:**

"I am getting a `System.NullReferenceException: Object reference not set to an instance of an object.` error in the `OnBarUpdate()` method of my NinjaScript indicator. This error happens on the line where I try to use `mySma`. The `mySma` variable is declared as `private SMA mySma;` I think the problem might be with how or when `mySma` is being initialized (created). Here are the important parts of my `OnStateChange()` and `OnBarUpdate()` methods:"

C#

`// private SMA mySma;`

- `// public int Period { get; set; } // This is an input property`
-

- `// protected override void OnStateChange()`
- `// {`
- `// if (State == State.SetDefaults)`
- `// {`
- `// Period = 14; // Default for the input`
- `// // I might have tried to do: mySma = SMA(Period); here,`
`but I'm not sure if that's right.`
- `// }`
- `// else if (State == State.DataLoaded)`
- `// {`
- `// // Or maybe I should do: mySma = SMA(Period); here?`
- `// }`
- `// }`
-
- `// protected override void OnBarUpdate()`
- `// {`
- `// if (CurrentBar < Period) return;`
- `// Values = mySma; // The error happens on this line`
- `// }`
- "What is the most likely reason for this `NullReferenceException` in NinjaScript, and what is the correct way to fix it?"

- **Anticipated Gemini Suggestion (What a helpful AI might say):**

"The `NullReferenceException` you're seeing when you try to access `mySma` in `OnBarUpdate` is almost certainly because the `mySma` object (your SMA indicator) has not been properly created and assigned to the `mySma` variable before

`OnBarUpdate` tries to use it. In other words, `mySma` is `null` (empty) at that point.

In NinjaScript, it's very important where you initialize (create) indicators that depend on market data (like an SMA, which needs price series like `Close`) or that use input parameters (like your `Period` setting).

The most likely cause and the solution are:

1. Incorrect Initialization Location:

- If you tried to initialize `mySma` in `State.SetDefaults` (e.g., `mySma = SMA(Period);`), this is **incorrect** and is the most common reason for this error.
- **Why it's wrong:** In `State.SetDefaults`, the historical price data (like the `Input` series that the SMA needs by default) is **not yet available**. Also, if `Period` is a user-changeable input, it might not have its final value yet in `State.SetDefaults`. Trying to create the SMA here would likely result in `mySma` either being `null` or being set up incorrectly.

■

2. Missing Initialization:

- It's also possible that `mySma` was declared (`private SMA mySma;`) but you never actually assigned an SMA object to it anywhere (i.e., the line `mySma = SMA(Period);` is missing or commented out). If so,

it will definitely be `null`.

- **The Correct Solution:** You **must** initialize indicators like `SMA` that depend on market data or user inputs in the `State.DataLoaded` block of your `OnStateChange()` method. By this state, NinjaTrader guarantees that all necessary historical data is loaded and your input parameters have their final values.
- Here's how your code should look (assuming `Period` is a public input property):

```
C#
public class MyFixedIndicator : Indicator
{
    • private SMA mySma; // Declare it here
    •
    • // Make Period an input property
    • #region Properties
    •
    •
    •
    • public int Period { get; set; }
    • #endregion
    •
    • protected override void OnStateChange()
    • {
    •     if (State == State.SetDefaults)
    •     {
    •         Description = @"Correctly initialized SMA indicator.";
    •         Name = @"MyFixedIndicator";
    •         Calculate = Calculate.OnBarClose;
```

- IsOverlay = true;
-
- // Set the default for the Period input property
- Period = 14;
-
- // Add your plot(s) here
- AddPlot(Brushes.CornflowerBlue, "MySMAOutput");
- }
- else if (State == State.DataLoaded) // <<<< CORRECT PLACE!
- {
- // Now it's safe to initialize mySma.
- // It will use the 'Period' value (which might have been changed by the user)
- // and by default, it will use the primary Input series (e.g., Close prices)
- // of the chart the indicator is on.
- mySma = SMA(Period);
- }
- }
-
- protected override void OnBarUpdate()
- {
- // It's good practice to check if CurrentBar is high enough for your Period
- // AND also check if mySma is not null, just in case.
- if (CurrentBar < Period |
-

| mySma == null) return;

- `// Now it should be safe to access mySma`
- `Values = mySma; // Assign the SMA's current value to our indicator's first plot`
- `}`
- `}`
- `...`
- By moving the initialization of ``mySma`` to ``State.DataLoaded``, you ensure that everything it needs (data and parameters) is ready, which should prevent the ``NullReferenceException``." [14, 7]

- **Explanation and Analysis (How did Gemini do?):**

- Gemini (hopefully!) correctly diagnosed the problem as being related to when and where the **SMA** indicator was initialized, which is a very common mistake in NinjaScript.
- It should have accurately pointed to **State.DataLoaded** as the correct lifecycle state for initializing data-dependent objects like **SMA**.
-
- The suggested fix should show the correct placement of the `mySma = SMA(Period);` line and also remind you that **Period** should be a proper input property.
- This example shows that Gemini can be helpful in figuring out errors that are very specific to how NinjaScript works, as long as you give it enough context and describe the error well.

7.4. Example 4: Optimizing a Loop or Calculation within a NinjaScript (Making it Faster)

- **Objective:** You have a piece of code in `OnBarUpdate` that does a calculation, but it does it in a way that's not very efficient. It might be re-calculating something over many old bars every single time a new bar comes in. You want Gemini to help you make it faster.
- **Inefficient Code Snippet (A slow way to manually calculate an SMA):**

```

C#
// This code tries to calculate a simple moving average manually,
• // but it does it inefficiently.
• // Assume 'Period' is an input property, e.g., public int Period {
  get; set; } = 20;
•
• protected override void OnBarUpdate()
• {
•   if (CurrentBar < Period) // If not enough bars yet for the full
    Period
•   {
•     Value = 0; // Or some other default value for the plot
•     return;
•   }
•
•   double sum = 0;
•   // THIS IS INEFFICIENT:
•   // This loop runs on EVERY OnBarUpdate call.
•   // It re-adds 'Period' number of Close prices every single time.
•   // If Period is 20, it does 20 additions. If Period is 200, it does
    200 additions!

```

- for (int i = 0; i < Period; i++) // Loop from current bar (i=0) back to 'Period-1' bars ago
- {
- sum = sum + Close[i]; // Close[i] gets the close price 'i' bars ago
- }
- double average = sum / Period;
- Value = average; // Assign the calculated average to our indicator's plot
- }
-
- **Engineered Prompt:**

"This NinjaScript C# code in my `OnBarUpdate()` method calculates a simple moving average over a `Period` number of bars. I think it's inefficient because the `for` loop recalculates the sum of all `Period` closes on every single bar update. How can I optimize this code to make it perform better in NinjaScript? Please suggest an alternative way to do this that avoids this repetitive summation. Maybe there's a way to update the sum more smartly (incrementally), or perhaps there's a built-in NinjaScript feature I should be using instead. Here's the code:"

- ```
C#
// [Paste the inefficient code snippet from above here]
• // Assume 'public int Period { get; set; }' is an input property.
• // protected override void OnBarUpdate()
• // {
• // if (CurrentBar < Period)
• // {
```

- `// Value = 0;`
- `// return;`
- `// }`
- `// double sum = 0;`
- `// for (int i = 0; i < Period; i++)`
- `// {`
- `// sum = sum + Close[i];`
- `// }`
- `// double average = sum / Period;`
- `// Value = average;`
- `// }`

- **Anticipated Gemini Suggestion (What a helpful AI might say):**

"You're right, the way your code calculates the simple moving average is inefficient because it re-sums all the **Close** prices in the **for** loop every time **OnBarUpdate** is called. Here are two much better and more performant ways to do this in NinjaScript:

**1. Use the Built-in **SMA** Indicator (This is the BEST and Recommended Way for SMAs):** NinjaScript has a built-in **SMA** indicator that is already highly optimized for performance. You should almost always use this when you need a Simple Moving Average.

C#

// In your indicator class, declare a variable to hold the SMA instance:

- `// private SMA mySmaInstance;`

- // And make sure you have your 'Period' input property:
- // public int Period { get; set; }
- 
- // Then, in OnStateChange:
- // protected override void OnStateChange()
- // {
- //   if (State == State.SetDefaults)
- //   {
- //     Period = 20; // Set your default period
- //     //... other default settings...
- //     AddPlot(Brushes.Blue, "OptimizedSMA\_Plot"); // Add a plot for it
- //   }
- //   else if (State == State.DataLoaded)
- //   {
- //     // Initialize your SMA instance HERE, using the Close price and your Period
- //     mySmaInstance = SMA(Close, Period);
- //   }
- // }
- 
- // And in OnBarUpdate:
- // protected override void OnBarUpdate()
- // {
- //   // Make sure enough bars have loaded and the SMA instance is ready
- //   if (CurrentBar < Period |
- 

| mySmaInstance == null) // return;



- `//` `//` Simply get the current value from your SMA instance!
- `//` `Value = mySmaInstance;` `//` Assign it to your indicator's plot
- `// }`
- `````
- This method is much better because it uses NinjaTrader's own optimized code for its indicators. [7]
- 
- **\*\*2. Incremental Summation** (If you were calculating a custom rolling sum/average where no built-in indicator exists):\*\*
- If you absolutely had to calculate a custom rolling sum or average for something that NinjaTrader doesn't have a built-in indicator for, you could make your loop more efficient by updating the sum incrementally (adding the newest value and subtracting the oldest value that's no longer in the window).
- 
- ````csharp`
- `//` In your indicator class:
- `//` `private double currentSum = 0;` `//` Variable to keep track of the sum
- `//` `public int Period { get; set; }` `//` Input property
- 
- `//` protected override void OnStateChange()
- `//` {
- `//` `if (State == State.SetDefaults)`
- `//` {
- `//` `Period = 20;`
- `//` `//... other defaults...`
- `//` `AddPlot(Brushes.Green, "CustomRollingAvg_Plot");`
- `//` }
- `//` `else if (State == State.Configure)` `//` Good place to reset sum

if script is reconfigured

- `// {`
- `//     currentSum = 0;`
- `// }`
- `// }`
- 
- `// protected override void OnBarUpdate()`
- `// {`
- `//     // Add the Close price of the newest bar (current bar, index`  
`0) to our sum`
- `//     currentSum = currentSum + Close;`
- 
- `//     if (CurrentBar >= Period) // Once we have at least 'Period'`  
`number of bars`
- `//     {`
- `//         // Subtract the Close price of the bar that is now falling`  
`out of our 'Period' window.`
- `//         // Close[Period] is the bar that is (Period + 1) bars ago`  
`from the very start,`
- `//         // or 'Period' bars ago from the *previous* bar's`  
`perspective.`
- `//         // This is the bar that's just leaving our sum.`
- `//         currentSum = currentSum - Close[Period];`
- `//         Value = currentSum / Period; // Calculate the average`
- `//     }`
- `//     else if (CurrentBar >= 0) // For the initial bars before we have`  
`a full 'Period'`
- `//     {`
- `//         // You could choose to calculate an expanding average`  
`here,`

- //        // or just wait until you have a full 'Period' of data.
- //        // This example calculates an average over the bars available so far.
- //        if (CurrentBar == 0)
- //        {
- //            Value = Close; // On the very first bar, sum is just Close
- //        }
- //        else
- //        {
- //            // This isn't a true 'Period' SMA yet, but an average of available bars
- //            // Value = currentSum / (CurrentBar + 1);
- //            // For a true SMA, you'd usually wait until CurrentBar >= Period - 1
- //            // and only assign Value in the block above.
- //            // For simplicity, let's just say we wait for full period:
- //            if (CurrentBar == Period - 1) // This is the first bar where we have a full 'Period'
- //            {
- //                Value = currentSum / Period;
- //            } else {
- //                // Not enough bars yet for a full period average, do nothing or set to a default
- //                Value = 0; // Or some other indicator of not-yet-ready
- //            }
- //        }
- //        }
- //        }
- // }
- ```

- **\*\*However, for a standard Simple Moving Average, the first method (using the built-in `SMA()` indicator) is definitely the one you should use.\*\* It's simpler, less error-prone, and already optimized by NinjaTrader." [7, 15]**
- **Explanation and Analysis (How did Gemini do?):**
  - Gemini (hopefully!) correctly spotted why the original loop was slow.
  - Most importantly, it should have suggested the **best NinjaScript practice**: use the built-in, optimized **SMA** indicator. This shows it understands more than just general C# optimization; it knows about specific tools within the NinjaScript framework.
  - 
  - It also (hopefully!) provided the "incremental sum" method as an alternative. This is a good general programming technique for optimizing rolling calculations and would be useful if you were trying to calculate something custom that NinjaTrader doesn't have a built-in indicator for.
  - The suggestions should also follow good NinjaScript structure, like reminding you to initialize indicators in **State.DataLoaded**.
  - This example shows that Gemini can not only help fix code but also guide you towards better and more efficient ways of using the NinjaScript framework itself.

In all these examples, notice how the prompts try to give Gemini a lot of information about the NinjaScript world (like mentioning **OnBarUpdate**, **State.DataLoaded**, the type of script like **Indicator** or **Strategy**, and specific NinjaScript functions like

`SMA()`). The answers Gemini might give show how, if you guide it well, it can create code that fits into NinjaScript's way of doing things, like managing states correctly for setup and putting the main logic in the right event handlers.

Remember, your job as the developer is still super important: you need to write these clear, detailed prompts and then very carefully check and test everything the AI gives you.

**Table 6: Sample Gemini Prompts for NinjaScript Tasks (Quick Ideas for Asking Gemini)**

| What You Want To Do (Task Category) | How You Might Ask Gemini (Sample Prompt)                                                                                                                                                         | Key NinjaScript Things to Mention                                                                                                                                           | What Kind of Answer to Expect                  |
|-------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------|
| Start a New Indicator (Scaffolding) | "Generate a NinjaScript Indicator for NT8. Call it 'MyCustomVol'. It needs one <code>int</code> input setting called <code>Length</code> (default value 20) and one plot called 'Volatility'. In | <code>Indicator</code> , <code>NT8</code> , names/types/defaults for inputs, plot names, <code>IsOverlay</code> , <code>Calculate</code> , <code>State.SetDefaults</code> . | The full C# class structure for the indicator. |

```

State.SetDefaults,
make sure
IsOverlay
= false and
Calculate
=
Calculate.
OnBarClose.
"

```

### Logic for Entering a Strategy Trade

"In the `OnBarUpdate` of a NinjaScript Strategy, if `RSI(14,3)` is less than 30 AND `Close` is greater than `EMA(Close, 200)`, AND if `Position.MarketPosition == MarketPosition.Flat` (not in a trade), then

Strategy, `OnBarUpdate`, specific indicators (RSI, EMA), `Position.MarketPosition, EnterLong()`.

A C# code block for the `OnBarUpdate` method that does this logic.

```
EnterLong(
DefaultQuantity,
"RSI_LE");
"
```

## Add a New User Setting (Input Parameter)

"Add a `bool` (true/false) input property named `EnableFeature` to my existing NinjaScript indicator. Set its default value to `true` in `State.SetDefaults`. For the UI, the display name should be 'Enable Feature', the group should be 'Settings', and it should be order

`NinjaScriptProperty`, `Display` attribute details (like `Name`, `GroupName`, `Order`), C# type (`bool`), default value, `State.SetDefaults`.

The C# property definition with all its attributes, and the update to `State.SetDefaults`.

number 5 in  
that group."

## Drawing Data on a Plot

"In `State.SetDefault` of my NinjaScript Indicator, add a `Plot` named 'SignalLine'. It should use `PlotStyle.Dash` and the color `Brushes.Magenta`. Then, in `OnBarUpdate`, I need to assign the value of my variable `myCalculatedValue` to this 'SignalLine' plot (use `Values[plot`

```
State.SetDefault
faults,
AddPlot(),
PlotStyle,
Brush,
OnBarUpdate,
Values[plotIndex][
barsAgo].
```

C# code  
for the  
`AddPlot()` call  
and the  
line in  
`OnBarUpdate`  
that  
assigns  
the  
value.



tIndex])."

**Fixing a  
NullReferenceException**

"My NinjaScript indicator is crashing with a `NullReferenceException` when it tries to use `indicatorInstance` in `OnBarUpdate`. I declared `indicatorInstance` as `private MACD macd`;. Can you show me the correct way to initialize (create) this `macd` object in the `OnStateChange` method?"

The error type (`NullReferenceException`), the variable name (`indicatorInstance`), its type (`MACD`), the methods involved (`OnBarUpdate`, `OnStateChange`), and what you think might be wrong (initialization).

C# code showing how to correctly initialize the `MACD` in `State.DataLoaded`.

## Explaining a Piece of Code

"Can you explain this NinjaScript C# code snippet from an `OnExecutionUpdate()` method? It says: `if (execution.Order.Name == "MyEntry" && execution.Order.OrderState == OrderState.Filled) { /* some logic here */ }`"

The specific NinjaScript method (`OnExecutionUpdate`), the types of objects (`Execution, Order`), and the properties being checked (`Order.Name, Order.OrderState`).

A plain English explanation of what that code is checking for and what it's trying to do.

|                                          |                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                           |                                                                                                                                                            |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Making a Loop Faster (Optimizing)</b> | "This loop in my NinjaScript <code>OnBarUpdate</code> goes from bar <code>0</code> all the way to <code>CurrentBar</code> just to find the highest high price. How can I make this faster so it only looks back <code>LookbackPeriod</code> number of bars and does it efficiently?" | Where the code is ( <code>OnBarUpdate</code> ), what the current slow logic is, what you want to achieve (faster, look back specific period), and any relevant variables ( <code>LookbackPeriod</code> ). | An optimized C# code snippet. It might suggest using a built-in NinjaScript function like <code>MAX(High, LookbackPeriod)</code> or a more efficient loop. |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|

Export to Sheets

This table gives you some quick ideas for how to ask Gemini for help with different common tasks when you're working on NinjaScript.

## 8. Addressing Potential Challenges and Limitations (Things to Watch Out For)

While Gemini AI is a super powerful helper for NinjaScript

development, it's really important to know that it's not perfect and there can be some challenges or limits. Knowing these things will help you have realistic expectations and use strategies to make sure the trading scripts you build are strong and reliable.

### **8.1. Understanding AI Hallucinations and Ensuring Code Accuracy (AI Making Stuff Up!)**

Sometimes, Large Language Models like Gemini can create answers that sound really good and confident, but are actually wrong, don't make sense, or are not relevant to your question. This is sometimes called **"AI hallucination."**

When it comes to writing code, this could mean Gemini might:

- Write C# code that looks correct and compiles, but the logic inside it is flawed (it doesn't do what you actually wanted).
- Try to use parts of the NinjaScript system (API calls) that don't actually exist.
- Misunderstand what you asked for in a small but important way.

For something like NinjaScript, where mistakes in your code can actually lead to losing real money, the risk from these "hallucinations" is a big deal. Some users have reported that when LLMs try to work with very specific systems like NinjaScript (which isn't as widely known as general programming libraries), they might sometimes "hallucinate" functions or settings that aren't really part of NinjaScript. This is probably because NinjaScript is more specialized, so there's less information about it in the huge amounts of data the AI was trained on.

### **How to Deal With This (Mitigation Strategies):**

- **ALWAYS Check the Code Carefully (Critical Review):** *Never* just assume that code from Gemini is correct, even if it looks good at first glance. You need to read it, understand it, and think about whether it really does what you need.
- **TEST, TEST, TEST!** This is the most important thing. All NinjaScript code, especially any logic that places orders or does critical calculations, *must* be tested like crazy. This includes:
  - **Compilation:** Make sure the code builds without any errors in NinjaTrader.
  - **Strategy Analyzer (Backtesting):** If it's a strategy, test it on lots of historical data. Try it on different stocks/futures and in different market conditions (when the market is trending, when it's choppy, etc.).
  - **Demo Trading (Paper Trading):** Before you ever use it with real money, run your strategy or indicator on a demo account with live market data. This lets you see how it behaves in real-time without risking anything.
  - **Unit Testing (If Possible):** If you have really complex helper functions or algorithms inside your NinjaScript, it might be a good idea to write separate small tests (unit tests) just for those pieces, if you know how.
- **Check the Official Docs (Cross-Referencing):** If Gemini uses a NinjaScript API call, class name, or method signature that you're not sure about, look it up in the official NinjaTrader NinjaScript documentation to make sure it's real and used correctly.
- 
- **Write Super Clear Prompts:** The clearer and more specific your instructions are to Gemini (like we talked about in Chapter

5), the less chance there is for the AI to misunderstand you.

## 8.2. Handling NinjaScript-Specific Nuances and Proprietary API Aspects (NinjaScript's Special Rules)

Gemini was trained on a huge amount of general C# code and public information. While it's good with C#, it won't have a perfect, built-in understanding of *every single detail* of NinjaTrader's own special API (Application Programming Interface – how different parts of NinjaTrader talk to each other). It won't know exactly how every NinjaScript object behaves in every possible situation, or the exact order and meaning of all the event handler calls (like `OnBarUpdate`, `OnOrderUpdate`, etc.) that happen when NinjaTrader is running.

For example, the tiny differences between when `OnMarketData` runs versus when `OnBarUpdate` runs if you have `Calculate.OnEachTick` set, or how `BarsArray[x]` (for extra data series) works with different types of trading sessions – these are very specific to NinjaTrader.

### How to Deal With This:

- **Be Explicit in Your Prompts:** As we've said before, when you ask Gemini for help, you need to clearly mention NinjaScript-specific things:
  - Class names (like `Indicator`, `Strategy`, `SMA`, `OrderFlowCumulativeDelta`).
  - Method names (like `EnterLong()`, `AddPlot()`, `SetStopLoss()`).
  - Event handlers (like `OnStateChange`, `OnExecution()`).
  - States (like `State.DataLoaded`, `State.Configure`).
- **Use Few-Shot Prompting (Show Examples):** If you want

Gemini to use a specific NinjaScript pattern, especially for things like attributes (e.g., `or`) or complicated API interactions, give it an example in your prompt. This helps "show" Gemini what you want.

- 
- **Let Gemini See Your Other Code (Local Codebase Awareness):** If you're using Gemini Code Assist in an IDE like VS Code, make sure your project includes other relevant NinjaScript files. These could be:
  - Base classes you've written.
  - Other indicators or strategies you're working on.
  - Even well-commented examples from NinjaTrader itself. Gemini can often learn from these local files, which can make its suggestions much more accurate and relevant for your NinjaScript-specific code.
- 
- **Give it Docs/Samples (If the Tool Allows):** Some AI tools let you upload documents or give them a collection of sample code to use as a reference. While Gemini Code Assist in an IDE mainly uses your local files, the idea is the same: having good reference material helps guide the AI, even if it's indirect.
- 

### 8.3. Dependency on Training Data and Staying Updated (AI Doesn't Know Tomorrow's News)

Gemini's knowledge is based on the information it was trained on, and that training happened up to a certain date. It might not know about the very newest features, changes to the API, or bug fixes that were added to NinjaTrader or NinjaScript *after* its last training session.

## How to Deal With This:

- **Always Check the Official NinjaTrader Documentation:** The official NinjaTrader developer documentation (you can find it at <https://developer.ninjatrader.com/docs/desktop>) is THE place to go for the most current and accurate information about the API, how classes work, and best practices.
- 
- **Be Careful with Brand New Features:** If NinjaTrader just released a new feature or is testing something in beta, be extra cautious if you ask Gemini about it. Gemini is more likely to be accurate about parts of the API that have been around for a while and are well-documented.
- **Mention the Version (If Possible):** If it makes sense, you can try telling Gemini which version of NinjaTrader you're using (e.g., "NinjaTrader 8") in your prompts. However, Gemini's ability to understand very specific version differences might be limited.

### 8.4. Rate Limits and API Usage Considerations (If You're Using the API Directly)

This is mostly for advanced users. If you decide to talk to the Gemini API directly using code (like with the C# SDK we mentioned earlier ) instead of using an IDE extension, you need to be aware of **API rate limits**.

These limits control:

- How many requests you can send per minute (RPM) or per day (RPD).
- How many "tokens" (like small pieces of text) you can process



per minute (TPM) or per day (TPD).

- 

If you go over these limits, you'll get errors, and your program won't be able to talk to Gemini. Different Gemini models might have different limits, and how much you pay (your usage tier) can also affect these limits.

This is usually less of a worry if you're just using an integrated IDE extension like Gemini Code Assist, because your subscription to that service typically covers normal use.

### 8.5. The Importance of Human Oversight and Testing (YOU Are Still the Boss!)

This is the **most important thing to remember**. AI, as it is today, is a very powerful *assistant* and a great tool to help you be more productive. It is **NOT** a replacement for:

- Your own expertise as a developer.
- Your ability to think critically.
- Your ultimate responsibility for the code.
- 

You, the developer, are still the main architect (the designer), the quality controller (the one who makes sure it's good), and the final judge of whether the code is right and safe.

The idea of "trust, but verify" (meaning, you can trust it to help, but you **MUST** check its work) is even **MORE** important when you're developing financial trading scripts. A small, hidden mistake in the logic of a strategy that Gemini helped you write, or an unusual market situation or NinjaScript behavior that the AI didn't account

for, could lead to you losing a lot of money.

Sometimes, LLMs can give you code, and it might not be totally clear *why* the AI wrote it that way (it can be like a "black box"). This means you need to be even *more* careful when you review it. Unlike a small visual bug on a website, a mistake in an automated trading script can have immediate, real, and bad financial results.

### How to Deal With This (Mitigation Strategies):

- **You Are Accountable:** You must take full responsibility for all the code in your scripts, whether you wrote it all by hand or an AI helped you.
- **Have a Super Strong Testing Plan:**
  - **Code Reviews:** If you work in a team, have other developers look at the AI-assisted code. Even if you work alone, take the time to critically review the parts Gemini generated yourself.
  - **Backtesting:** Use NinjaTrader's Strategy Analyzer to test your strategies on many different sets of historical data. Check if the logic makes sense, look at the performance numbers, and see if it holds up in different kinds\_of markets (trending, sideways, volatile, calm).
  - **Forward Testing (Paper Trading):** Run your script on a demo account with live market data for a good amount of time. This lets you see how it behaves in real-time market conditions before you risk any real money.
  - **Think About Edge Cases:** Try to imagine and, if you can, test how your script would behave in unusual market situations (like a sudden "flash crash," super high volatility, or if there are gaps in the data).

- **Use Your Trading Knowledge:** You need a good understanding of trading principles, how markets work, and the specific thing you're trying to trade (like a particular stock or future) to be able to tell if the logic of a trading script is sound.

**Table 7: Common NinjaScript Issues & How AI Might Help You Debug Them**

| Common Problem in NinjaScript                                                      | Likely Cause (Based on NinjaScript Rules)                                                                       | How Gemini Can Help You Think About It                                                     | Example Gemini Prompt for Debugging                                                                                                                                                 |
|------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>NullReferenceException</b><br>(Trying to use something that's empty/not set up) | You're trying to use an indicator or object that wasn't initialized (created properly. Often happens if you try | Explain what the error means; suggest you check where and when you initialized the object; | "I'm getting a <b>NullReferenceException</b> on <b>myIndicator</b> in <b>OnBarUpdate</b> . Here's my <b>OnStateChange</b> code and where I declared <b>myIndicator</b> . What did I |

to initialize in the wrong `OnStateChange` state (like in `State.SetDefaults` instead of `State.DataLoaded`).

review your `OnStateChange` logic.

do wrong?"

### Doing Setup in the Wrong State

Example: Trying to use `AddDataSeries()` outside of `State.Configure`, or trying to initialize

Identify that you're doing something in the wrong state; suggest the correct state

"Is it correct to initialize `SMA(Close, Period)` in `State.SetDefaults` within NinjaScript? If not, where should I do it?"

an SMA  
in  
State.  
SetDef  
aults.

based  
on  
NinjaScr  
ipt's  
lifecycle  
.

### Script is Slow / Laggy in OnBarUpdat e

You have  
inefficie  
nt loops;  
you're  
doing  
the  
same  
calculati  
ons over  
and  
over;  
you're  
creating  
too  
many  
DrawOb  
jects;  
or you  
have  
very  
complex  
logic  
running

Look at  
your  
code for  
slow  
parts  
(bottlen  
ecks);  
suggest  
faster  
ways to  
do  
things  
or using  
built-in  
function  
s;  
recomm  
end  
storing  
(caching  
) results  
you use  
often.

"This  
OnBarUpdat  
e loop in my  
NinjaScript  
indicator is  
really slow.  
[Paste your  
code here].  
How can I  
make it run  
faster?"

on every  
single  
tick.

## Strategy Order Logic Isn't Working Right

Your  
conditio  
ns for  
`EnterLong/Short`  
might be  
wrong;  
you  
might be  
handling  
`OnOrderUpdate` or  
`OnExecution`  
incorrect  
ly; or you  
might  
have  
problem  
s  
keeping  
track of  
your  
current

Analyze  
your  
order  
entry/exi  
t logic;  
help you  
trace  
what  
happens  
to an  
order  
through  
its  
lifecycle  
; explain  
how  
`OrderState`  
changes  
.

"My NinjaScript  
strategy  
enters trades  
okay, but my  
`OnExecution`  
code isn't  
updating my  
profit target  
variable  
correctly.  
Here's the  
code. Why  
isn't it  
working?"

Position.

**IndexOutOfRangeException** (Trying to get data from a list using a bad index)

You're trying to get `Close[ barsAg o]` but `barsAg o` is too big (goes past the start of your data) or negative. Or, `Curren tBar` is less than the number of bars you need for your lookback period.

Explain what the error means; suggest you add checks for `Curren tBar` or make sure your `barsAg o` index is valid before using it.

"My NinjaScript indicator is crashing with an `IndexOutOfRangeException` when it tries to access `High[lookbackPeriod]`. My `lookbackPeriod` is 20. What check should I add in `OnBarUpdate` to prevent this?"

**Plot on Chart  
Isn't  
Showing Up  
or Updating**

Your `AddPlot` call might be wrong; you might be assigning values to the wrong `Values[plotIndex]`; there could be issues with your `Calculate` mode; or maybe `IsOverlay=false` when it should be true for a plot

Review how you set up your plot in `State.SetDefaults` and how you're assigning values to it in `OnBarUpdate`.

"My second plot (I called it 'Plot2') in my NinjaScript indicator isn't showing any data on the chart. I have `AddPlot(Brushes.Red, "Plot2");` in `State.SetDefaults`. In `OnBarUpdate`, I assign a value to `Values`. What could be missing or wrong?"



on the  
price  
panel.

By knowing about these possible problems and actively using ways to prevent or fix them, you can use the power of Gemini to make your NinjaScript development better, while still keeping the high standards of reliability and accuracy that are so important in the world of finance.

## 9. Conclusion: Advancing Your NinjaScript Development with AI (Wrapping It All Up)

Bringing smart AI tools like Google's Gemini into how you build NinjaScript is a big step forward for traders and developers who want to create their own custom analysis tools and automatic trading strategies. This guide has tried to give you a complete picture of how to understand and use this powerful combination.

**Let's Quickly Remember the Benefits:** As we've gone through the basics of NinjaScript, how it's structured, what Gemini can do with C#, and how to talk to Gemini effectively, we've seen several key advantages:

- **Faster Development:** Gemini can really speed up how quickly you can write NinjaScript code. It can help with the basic starting structure, write repetitive parts, and even help implement complicated logic. This means you can build things faster.
-

- **Better Code Understanding:** If you have existing NinjaScripts or ones from other people, Gemini can be a great help in quickly understanding what the C# code does, figuring out the logic, and seeing the purpose of different parts.
- 
- **Easier Debugging:** Gemini can help you figure out errors, explain what error messages mean, and suggest ways to fix common problems in NinjaScript. This can make the often frustrating job of debugging much smoother.
- 
- **Help with Optimization:** You can use Gemini to look at your code for parts that might be slow and suggest ways to make it run faster, following NinjaScript best practices. This leads to more efficient scripts.
- 
- **Helps You Learn and Innovate:** If you're new to C# or NinjaScript, Gemini can be like an interactive teacher. If you're an experienced developer, it can help you try out new parts of the NinjaScript system or build prototypes of new ideas more quickly. The idea that AI can make it easier for more people to get into building complex trading algorithms (while still needing very careful checking) is pretty exciting.
- 

However, you get these benefits best when you combine AI's abilities with:

- A strong understanding of the basics of NinjaScript.
- Knowing its event-driven structure (how it reacts to things happening).
- Careful management of its lifecycle states using

## OnStateChange.

- 
- Always following good development habits for performance and handling errors.

The role of the developer changes a bit. You're not just the only coder anymore. You become more like an architect and a team leader working *with* the AI. You tell the AI what you need, and then you carefully check and approve what it produces.

**What's Next for AI in Building Trading Algorithms?** The use of AI in financial technology and building trading algorithms is only going to grow. This guide focused on using AI as a *code assistant* for NinjaScript. But in the future, AI might do even more direct things, like:

- **Helping Come Up with Strategy Ideas:** AI models might be able to look at market data and suggest new ideas for trading strategies, or even create the starting logic for a strategy if you give it some high-level goals and risk limits.
- **Smarter Analysis of Backtesting Results:** AI could be used to do a more advanced study of your backtesting results, helping to find hidden patterns, warning you if your strategy might be "overfit" (meaning it only works on old data but not new data), or finding the best settings for your strategy.
- **Strategies That Change Themselves (Dynamic Adaptation):** In the future, we might see systems where AI parts can automatically adjust a strategy's settings or even its logic if the market conditions change. However, this would be very complex and would need extremely careful risk management.
-

Even with all these exciting future ideas, the most important things will still be:

- Testing everything very, very thoroughly.
- Having humans oversee what the AI is doing.
- Having a deep understanding of both the market and the tools you're using. The idea of "trust, but verify" will become even more critical as AI gets more involved.
- 

**Your Turn to Try! (Final Call to Action):** Using Gemini in your NinjaScript development is something you learn by doing. It's a process of trying things out and getting better over time.

We really encourage you to start experimenting with Gemini Code Assist in your favorite code editor (like VS Code). Try using the prompt engineering tips and thinking about the NinjaScript structure that we've talked about in this guide.

- Start with simple tasks:
  - Ask Gemini to generate small functions.
  - Ask it to explain a piece of code you already have.
  - Try getting it to create the basic starting point for a simple indicator.
- As you get more comfortable, you can try more complex things.
- **And always, always remember to carefully check, test, and double-test any code that AI helps you with.**

By using AI as a powerful helper, while still sticking to disciplined and careful development practices, the NinjaScript community can reach new levels of productivity, come up with new ideas, and build even

more sophisticated tools for effective algorithmic trading.

## Works Cited

1. Adaptrade Software. "NinjaScript for Automated Strategy Development." *Adaptrade Software Newsletter*, May 2014. (<http://www.adaptrade.com/Newsletter/NL-NinjaScript.htm>)
2. NinjaTrader Support. "Developer Guide: Getting Started with NinjaScript." *NinjaTrader Support*, Accessed May 28, 2025. (<https://support.ninjatrader.com/s/article/Developer-Guide-Getting-Started-with-NinjaScript>)
3. SonarSource. "What is Gemini Code Assist?" *SonarSource Learn*, Accessed May 28, 2025. <https://www.sonarsource.com/learn/gemini-code-assist/>
4. Ninja AI. "Ninja AI | All-In-One AI Assistant." *NinjaTech.ai*, Accessed May 28, 2025. <https://www.ninatech.ai/>
5. Google Cloud. "Gemini Code Assist." *Google Cloud*, Accessed May 28, 2025. <https://cloud.google.com/products/gemini/code-assist>
6. Google Cloud. "Write code with Gemini Code Assist." *Gemini for Google Cloud Docs*, Last updated May 28, 2025. <https://cloud.google.com/gemini/docs/codeassist/write-code-gemini>
7. NinjaTrader Support. "Basic Syntax." *NinjaTrader 8 Help Guide*, Accessed May 28, 2025. [https://ninjatrader.com/support/helpguides/nt8/basic\\_syntax.htm](https://ninjatrader.com/support/helpguides/nt8/basic_syntax.htm)
8. Toolify.ai. "Build Your Own NinjaTrader Indicator." *Toolify.ai AI News*, Accessed May 28, 2025. <https://www.toolify.ai/ai-news/build-your-own-ninjatrader-indicator>

[ator-352247](#)

9. NinjaTrader Forum. "NinjaScript programmer's IDE?" *NinjaTrader Support Forum*, December 25, 2023.  
<https://forum.ninjatrader.com/forum/ninjatrader-8/strategy-development/1283580-ninjascript-programmer-s-ide>
10. LuxAlgo. "NinjaScript Basics for Custom Indicators." *LuxAlgo Blog*, Accessed May 28, 2025.  
<https://www.luxalgo.com/blog/ninjascript-basics-for-custom-indicators/>
11. Topstep. "NinjaTrader 8 Connection Instructions." *Topstep Help Center*, Accessed May 28, 2025.  
<https://help.topstep.com/en/articles/8307768-ninjatrader-8-connection-instructions>
12. NinjaTrader Support. "NinjaScript Best Practices." *NinjaTrader 8 Help Guide*, Accessed May 28, 2025.  
[https://ninjatrader.com/support/helpguides/nt8/ninjascript\\_best\\_practices.htm](https://ninjatrader.com/support/helpguides/nt8/ninjascript_best_practices.htm)
13. NinjaTrader Forum. "Making NinjaScript indicator efficient." *NinjaTrader Support Forum*, September 3, 2024.  
<https://forum.ninjatrader.com/forum/ninjatrader-8/indicator-development/1316664-making-ninjascript-indicator-efficient>
14. Google Cloud. "Code with Gemini Code Assist." *Cloud Workstations Docs*, Accessed May 28, 2025.  
<https://cloud.google.com/workstations/docs/write-code-gemini>
15. JetBrains. "Write effective AI prompts." *JetBrains AI Guide*, Accessed May 28, 2025.  
<https://www.jetbrains.com/guide/ai/tips/ai-prompting/>
16. Google Cloud. "Code customization overview." *Gemini Code Assist Enterprise Docs*, Accessed May 28, 2025.  
<https://developers.google.com/gemini-code-assist/docs/code-c>

[ustomization-overview](#)

17. OpenAI. "Best practices for prompt engineering with the OpenAI API." *OpenAI Help Center*, Accessed May 28, 2025.  
<https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api>
18. VisibleThread. "Creating Effective Prompts: Best Practices, Prompt Engineering, and How to Get the Most Out of Your LLM." *VisibleThread Blog*, Accessed May 28, 2025.  
<https://www.visiblethread.com/blog/creating-effective-prompts-best-practices-prompt-engineering-and-how-to-get-the-most-out-of-your-llm/>
19. C# Corner. "Craft Most Effective AI Prompts To Get The Most Out Of ChatGPT." *C# Corner*, Accessed May 28, 2025.  
<https://www.c-sharpcorner.com/article/craft-most-effective-ai-prompts-to-get-the-most-out-of-chatgpt/>
20. Google AI for Developers. "Prompt design strategies." *Gemini API Docs*, Accessed May 28, 2025.  
<https://ai.google.dev/gemini-api/docs/prompting-strategies>
21. NinjaTrader Developer. "NinjaScript Documentation." *NinjaTrader Developer Docs*, Accessed May 28, 2025.  
<https://developer.ninjatrade.com/docs/desktop>
22. Reddit. "Best AI coding assistant for developing NinjaTrader 8 indicators? (Beginner coder)." *Reddit r/ninjatrade*, Accessed May 28, 2025.  
[https://www.reddit.com/r/ninjatrade/comments/1hozapv/best\\_ai\\_coding\\_assistant\\_for\\_developing/](https://www.reddit.com/r/ninjatrade/comments/1hozapv/best_ai_coding_assistant_for_developing/)
23. TrendSpider. "Use AI To Code Your Ideas Into Custom Indicators." *TrendSpider Blog*, Accessed May 28, 2025.  
<https://trendspider.com/blog/use-ai-to-code-your-ideas-into-custom-indicators/>

24. Microsoft Learn. "Quickstart - Create a .NET AI app using the AI app template." *Microsoft Learn*, Last updated March 7, 2025.  
<https://learn.microsoft.com/en-us/dotnet/ai/quickstarts/ai-templates>
25. GitHub - gunpal5. "Google\_GenerativeAI: Most complete C#.Net SDK for Google Generative AI and Vertex AI." *GitHub*, Last updated February 24, 2025.  
[https://github.com/gunpal5/Google\\_GenerativeAI](https://github.com/gunpal5/Google_GenerativeAI)
26. DEV Community - gunpal5. "Google Gemini's Multimodal Live API with C#.Net." *DEV Community*, Last updated March 22, 2025.  
<https://dev.to/gunpal5/google-geminis-multimodal-live-api-with-c-net-4df1>
27. Google AI for Developers. "Rate limits." *Gemini API Docs*, Last updated May 21, 2025.  
<https://ai.google.dev/gemini-api/docs/rate-limits>
28. NinjaTrader Developer. "NinjaScript Documentation." *NinjaTrader Developer Docs*, Accessed May 28, 2025.  
<https://developer.ninjatrade.com/docs/desktop> (Duplicate of 21, included for completeness of all references)
29. Arsturn. "Unraveling the Real-World Applications and Limitations of Gemini 2.5 Pro." *Arsturn Blog*, Last updated April 16, 2025.  
<https://www.arsturn.com/blog/exploring-gemini-2-5-pro-coding-applications-limitations>
30. arXiv. "TradingAgents: Multi-Agents LLM Financial Trading Framework." *arXiv*, January 10, 2025.  
<https://arxiv.org/html/2412.20138v3>
31. Phaedra Solutions. "Debug AI Code: 10 Tips to Solve Common Errors." *Phaedra Solutions Blog*, Accessed May 28,



2025. <https://www.phaedrasolutions.com/blog/debug-ai-code>
32. Visual Studio Code. "Build agents and prompts in AI Toolkit."  
*Visual Studio Code Docs*, Last updated March 7, 2025.  
<https://code.visualstudio.com/docs/intelligentapps/agentbuilder>