Semantic Kernel + Hugging Face Inference: Step-by-Step with Code References

This document maps each step in your C# program to the exact code snippet used. It covers configuration, kernel setup, chat history, streaming responses, and history reduction. API keys are redacted.

# appsettings.json (snippet)

{  
 "HuggingFace": {  
 "modelid": "deepseek-ai/DeepSeek-R1",  
 "endpoint": "https://router.huggingface.co",  
 "apikey": "hf\_\*\*\*REDACTED\*\*\*"  
 }  
}

# 1) Configuration Setup

Loads configuration from appsettings.json, environment variables, and user secrets into IConfiguration.

IConfiguration config = new ConfigurationBuilder()  
 .SetBasePath(Directory.GetCurrentDirectory())  
 .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)  
 .AddEnvironmentVariables()  
 .AddUserSecrets<HuggingFace>  
 ()  
 .Build();

# 2) Extract Settings (optional)

Reads values. In your code these variables are not used later because you pull from the HuggingFace section directly.

string? modelId = config["modelid"];  
string? endPoint = config["endpoint"];  
string? apikey = config["apiKey"];

# 3) Create Kernel Builder

Initializes the Semantic Kernel builder to register AI services.

var builder = Kernel.CreateBuilder();

# 4) Register Hugging Face Chat Completion

Adds the Hugging Face chat completion service using model ID, endpoint, and API key from the HuggingFace section.

builder.AddHuggingFaceChatCompletion(  
 config["HuggingFace:modelid"]!,  
 new Uri(config["HuggingFace:endpoint"])!,  
 config["HuggingFace:apikey"]!  
);

# 5) Build the Kernel

Finalizes the kernel with all registered services.

Kernel kernel = builder.Build();

# 6) Create Chat History

Creates a conversation history. The system message defines assistant behavior (use a professional style in production).

var history = new ChatHistory(systemMessage: "Talk very very rudely");

# 7) Get Chat Service

Retrieves the chat completion service from the kernel via dependency injection.

var chatCompleationService = kernel.GetRequiredService<IChatCompletionService>();

# 8) Define Execution Settings

Controls generation behavior: Temperature for creativity and MaxTokens for output length.

HuggingFacePromptExecutionSettings settings = new()  
{  
 Temperature = 1f,  
 MaxTokens = 1500,  
};

# 9) History Management (Reducer)

Keeps chat history short to avoid token limits. You used a truncation reducer; a summarization reducer is also available.

var reducer = new ChatHistoryTruncationReducer(targetCount: 2);  
// Alternative:  
// var reducer = new ChatHistorySummarizationReducer(chatCompleationService, 2, 2);

# 10) Inspect Service Attributes

Prints metadata about the configured provider (may include model, provider, etc.).

foreach (var attr in chatCompleationService.Attributes)  
 Console.WriteLine($"{attr.Key} \t\t{attr.Value}");

# 11) Interactive Loop

Accepts user prompts until an empty line is entered.

while (true)  
{  
 Console.Write("Enter your prompt:");  
 var prompt = Console.ReadLine();  
 if (string.IsNullOrEmpty(prompt))  
 break;

# 12) Add User Message to History

Adds the user's prompt into the conversation context.

history.AddUserMessage(prompt);

# 13) Stream the AI Response

Sends the history + settings to the provider and streams partial tokens as they arrive. Chunks are printed and accumulated.

string fullMessage = "";  
await foreach (StreamingChatMessageContent responseChunk in   
 chatCompleationService.GetStreamingChatMessageContentsAsync(history, settings))  
{  
 Console.Write(responseChunk.Content);  
 fullMessage += responseChunk.Content;  
}

# 14) Save Assistant Reply

Stores the assistant's full response into the conversation memory so the next turn has context.

history.AddAssistantMessage(fullMessage);

# 15) Token Usage Note (HF)

The Hugging Face connector does not currently expose detailed token usage; print a notice instead.

Console.WriteLine("\n\t[Token usage reporting not available for Hugging Face connector]");

# 16) Apply History Reducer

Optionally trims the chat history to your target to keep context size manageable.

var reduceMessages = await reducer.ReduceAsync(history);  
if (reduceMessages != null)  
 history = new(reduceMessages);

# Appendix: Using the OpenAI-Compatible Router Endpoint

If you target Hugging Face's OpenAI-compatible Router (paths like /v1/chat/completions), register the OpenAI connector instead. Keep the same base URL and HF token.

builder.AddOpenAIChatCompletion(  
 modelId: "deepseek-ai/DeepSeek-R1",  
 apiKey: config["HuggingFace:apikey"]!,  
 endpoint: new Uri("https://router.huggingface.co")  
);