Mastering Real-Time AI Responses: A Complete Guide to Streaming Completions

Welcome to this comprehensive tutorial on streaming responses with the OpenRouter SDK for .NET! This guide explores Example02.StreamingCompletion, which demonstrates how to receive AI responses in real-time as they're generated, creating a more responsive and engaging user experience.

Table of Contents

Overview

This example demonstrates how to stream AI completions in real-time rather than waiting for the entire response to be generated. Instead of showing a loading spinner for several seconds and then displaying the complete answer, streaming lets you show text as it's being generated - just like ChatGPT's typing effect.

What you'll learn:

- How streaming differs from standard completions
- How to use async streams with `await foreach`
- How to process chunks of data as they arrive
- When streaming provides better user experience
- How to handle errors in streaming scenarios
- Real-time response rendering techniques

What is Streaming and Why Use It?

The Problem with Non-Streaming

In Example01, we saw this pattern:

```
var response = await client.Chat.CreateAsync(request);
Console.WriteLine(response.Choices[0].Message.Content);
```

Here's what happens behind the scenes:

1. Send request to API
2. Wait 2-10 seconds (or more for complex queries)
3. Receive complete response
4. Display everything at once

The user experience:

- User sees nothing while waiting
- No indication of progress
- Feels slow and unresponsive
- Can't start reading until everything is done

The Streaming Solution

With streaming:

```
await foreach (var chunk in client.CallModelStreamAsync(...))
{
    Console.Write(chunk);
}
```

What happens:

1. Send request to API
2. Start receiving text immediately (within 1 second)
3. Display each word/phrase as it arrives
4. User can start reading right away

The user experience:

- Immediate feedback
- Feels fast and responsive
- Can start reading/processing while AI is thinking
- Modern, ChatGPT-like experience

When to Use Streaming

Use streaming when:

- Building chat interfaces
- Long-form content generation
- User is watching the output in real-time
- Perceived performance matters
- You want to show progress

Don't use streaming when:

- Need the complete response for processing
- Building batch operations
- Response is very short
- No user is watching
- Simplicity is more important than UX

Project Structure

The Example02.StreamingCompletion project has the same structure as Example01:

```
Example02.StreamingCompletion/
├── Example02.StreamingCompletion.csproj  # Project configuration file
├── Program.cs                             # Main application code
└── BLOG.md                                # This guide
```

Dependencies:

- **OpenRouter.SDK**: Provides streaming capabilities
- **OpenRouter.Examples.EnvConfig**: Manages API keys and configuration

---

Prerequisites

Before starting, ensure you have:

1. **.NET 8.0 SDK** installed (Download here)
2. **OpenRouter API Key** (Get one here)
3. **Completed Example01** or understand basic OpenRouter SDK usage
4. Understanding of async/await patterns
5. **Optional**: Familiarity with IEnumerable and iteration patterns

New Concept in This Example:

- **Async Streams**: Understanding `IAsyncEnumerable<T>` and `await foreach`

Understanding the Project File

The project file for Example02 is identical to Example01:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="OpenRouter.SDK" Version="1.0.0" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference
Include="..\OpenRouter.Examples.EnvConfig\OpenRouter.Examples.EnvConfig.csproj"
/>
  </ItemGroup>

</Project>
```

Why It's the Same

The streaming functionality is part of the OpenRouter.SDK package - no additional dependencies needed. The SDK handles:

- HTTP/2 Server-Sent Events (SSE)
- Chunk parsing and deserialization
- Async stream implementation
- Connection management

This is one of the benefits of using a well-designed SDK - complex features like streaming work out of the box.

Code Walkthrough

Let's analyze every line of `Program.cs`:

The Complete Code

```
using OpenRouter.SDK;
using OpenRouter.Examples.EnvConfig;

Console.WriteLine("===========================================");
Console.WriteLine("Example 2: Streaming Completion");
Console.WriteLine("===========================================\n");

await Example02.RunAsync();
```

```
Console.WriteLine("\n===========================================");
Console.WriteLine("Example completed!");
Console.WriteLine("===========================================");

public static class Example02
{
    public static async Task RunAsync()
    {
        var apiKey = ExampleConfig.ApiKey;

        var client = new OpenRouterClient(apiKey);

        Console.WriteLine("=== Example 2: Streaming Completion ===");
        try
        {
            Console.Write("Streaming response: ");
            await foreach (var chunk in client.CallModelStreamAsync(
                model: "openai/gpt-3.5-turbo",
                userMessage: "Count from 1 to 10 slowly",
                maxTokens: 100
            ))
            {
                Console.Write(chunk);
            }
            Console.WriteLine();
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error streaming: {ex.Message}");
        }
    }
}
```

Step-by-Step Breakdown

Step 1: Using Directives

```
using OpenRouter.SDK;
using OpenRouter.Examples.EnvConfig;
```

Same as Example01 - we need the SDK and configuration utilities.

Step 2: Welcome Banner

```
Console.WriteLine("===========================================");
Console.WriteLine("Example 2: Streaming Completion");
Console.WriteLine("===========================================\n");
```

Identifies which example is running.

Step 3: Initialize the Client

```
var apiKey = ExampleConfig.ApiKey;
var client = new OpenRouterClient(apiKey);
```

Exactly the same as Example01:

1. Get API key from configuration
2. Create client instance

The same client supports both streaming and non-streaming operations.

Step 4: The Streaming Loop (THE KEY PART)

```
Console.Write("Streaming response: ");
await foreach (var chunk in client.CallModelStreamAsync(
    model: "openai/gpt-3.5-turbo",
    userMessage: "Count from 1 to 10 slowly",
    maxTokens: 100
))
{
    Console.Write(chunk);
}
Console.WriteLine();
```

This is where the magic happens. Let's break it down further:

The Setup:

```
Console.Write("Streaming response: ");
```

Prints the label without a newline, so the streaming text appears on the same line.

The Async Stream:

```
await foreach (var chunk in client.CallModelStreamAsync(...))
```

This line does A LOT. Let's understand each part:

- `await foreach`: New C# 8.0 syntax for iterating over async streams
- `var chunk`: Each piece of text as it arrives from the API
- `client.CallModelStreamAsync(...)`: Returns `IAsyncEnumerable<string>`

How it works:

1. `CallModelStreamAsync` starts the HTTP request
2. As soon as the API sends the first chunk, the loop begins
3. Each iteration processes one chunk
4. The loop automatically `await`s for the next chunk
5. When the API finishes, the loop ends

The Method Parameters:

```
model: "openai/gpt-3.5-turbo",
```

Which AI model to use. GPT-3.5-turbo is fast and good for streaming.

```
userMessage: "Count from 1 to 10 slowly",
```

The prompt. A simple counting task perfect for seeing streaming in action.

```
maxTokens: 100
```

Maximum length of response. Limits cost and response size.

Processing Each Chunk:

```
{
    Console.Write(chunk);
}
```

For each chunk received:

- Write it immediately to console
- No buffering, no waiting
- User sees it right away

The Final Newline:

```
Console.WriteLine();
```

After all chunks are received, move to the next line.

Step 5: Error Handling

```
catch (Exception ex)
{
    Console.WriteLine($"Error streaming: {ex.Message}");
}
```

Catches any errors during streaming:

- Network interruptions
- API errors
- Authentication failures
- Timeout issues

Understanding Async Streams

Async streams are a powerful C# feature that Example02 relies on. Let's understand them deeply.

What is IAsyncEnumerable?

Think of it as a combination of:

- `IEnumerable<T>`: A collection you can iterate through
- `Task<T>`: An async operation

```
// Regular enumeration (synchronous)
IEnumerable<string> items = GetItems();
foreach (var item in items)
{
    Console.WriteLine(item);
}

// Async enumeration (asynchronous)
IAsyncEnumerable<string> items = GetItemsAsync();
await foreach (var item in items)
{
    Console.WriteLine(item);
}
```

Why Async Streams Are Perfect for Streaming

Traditional async (Example01):

```
Task<Response> response = client.GetResponseAsync();
// Wait for ENTIRE response
var result = await response;
// Process result
```

With async streams (Example02):

```
IAsyncEnumerable<string> chunks = client.GetStreamAsync();
// Process each chunk AS IT ARRIVES
await foreach (var chunk in chunks)
{
    ProcessChunk(chunk);
}
```

The Flow of Execution

Here's what happens when you run the streaming code:

```
Time 0ms:  await foreach starts
Time 1ms:  HTTP request sent to OpenRouter
Time 200ms: First chunk arrives: "1"
           → Loop iteration 1: Print "1"
Time 400ms: Second chunk: ", 2"
           → Loop iteration 2: Print ", 2"
Time 600ms: Third chunk: ", 3"
           → Loop iteration 3: Print ", 3"
...
Time 2000ms: Last chunk: ", 10"
            → Loop iteration 10: Print ", 10"
Time 2001ms: Stream ends, loop exits
```

Key insight: The loop body executes immediately when each chunk arrives, not all at once at the end.

Running the Example

Using Visual Studio

1. Open `OpenRouter.SDK.sln`
2. Right-click `Example02.StreamingCompletion` → Set as Startup Project
3. Press **F5**

Using .NET CLI

```
cd Examples/Example02.StreamingCompletion
dotnet run
```

Using PowerShell

```
cd
"c:\Users\subhr\source\repos\OpenRouter.SDK\OpenRouter.SDK\Examples\Example02.StreamingCompletion"
dotnet run
```

What You'll See

```
===========================================
Example 2: Streaming Completion
===========================================

=== Example 2: Streaming Completion ===
Streaming response: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

===========================================
Example completed!
===========================================
```

Notice: The numbers appear one by one in real-time, not all at once!

Streaming vs. Non-Streaming

Let's compare both approaches side by side:

Non-Streaming (Example01 Style)

```
var request = new ChatCompletionRequest
{
    Model = "openai/gpt-3.5-turbo",
    Messages = new List<Message>
    {
        new UserMessage
        {
            Role = "user",
            Content = "Count from 1 to 10 slowly"
        }
    }
};

var response = await client.Chat.CreateAsync(request);
Console.WriteLine(response.Choices[0].Message.Content);
```

Characteristics:

- Simple code
- Get full structured response
- Easier error handling
- Wait for complete response
- No progress indication
- Higher perceived latency

Streaming (Example02 Style)

```
await foreach (var chunk in client.CallModelStreamAsync(
    model: "openai/gpt-3.5-turbo",
    userMessage: "Count from 1 to 10 slowly",
    maxTokens: 100
))
{
    Console.Write(chunk);
}
```

Characteristics:

- Lower perceived latency
- Modern UX
- Can process chunks as they arrive
- Slightly more complex
- Need to buffer if you want full text

- Can't easily cancel mid-stream

Performance Comparison

Time to First Token:

- Non-streaming: 2-5 seconds
- Streaming: 0.5-1 second

Time to Complete Response:

- Both: Same (depends on response length and model)

Perceived Speed:

- Non-streaming: Feels slow
- Streaming: Feels fast

Key Takeaway: Streaming doesn't make the AI faster, but makes your app *feel* faster.

Error Handling in Streams

Streaming introduces unique error scenarios:

Common Streaming Errors

1. Mid-Stream Network Failure

```
try
{
    await foreach (var chunk in client.CallModelStreamAsync(...))
    {
        Console.Write(chunk);
        // Network drops here - what happens?
    }
}
catch (HttpRequestException ex)
{
    Console.WriteLine($"\nConnection lost: {ex.Message}");
    // Response is incomplete
}
```

Challenge: You've already shown partial response to user.

Solutions:

- Show "(connection lost)" message
- Implement retry with continuation
- Save partial response and ask to retry

2. API Errors During Streaming

```
try
{
    await foreach (var chunk in client.CallModelStreamAsync(...))
    {
        Console.Write(chunk);
    }
}
catch (OpenRouter.SDK.Exceptions.RateLimitException ex)
{
    Console.WriteLine($"\nRate limit exceeded: {ex.Message}");
}
```

3. Timeout

Long responses might timeout:

```
using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(30));

try
{
    await foreach (var chunk in client.CallModelStreamAsync(
        model: "openai/gpt-3.5-turbo",
        userMessage: "Write a long essay",
        maxTokens: 1000
    ).WithCancellation(cts.Token))
    {
        Console.Write(chunk);
    }
}
catch (OperationCanceledException)
{
    Console.WriteLine("\nResponse timed out");
}
```

Best Practices for Stream Error Handling

1. **Always wrap in try-catch**
2. **Indicate partial responses to users**
3. **Implement graceful degradation**
4. **Log errors for debugging**
5. **Consider retry strategies**

Practical Use Cases

Use Case 1: Chat Application

```
public async Task StreamChatResponse(string userMessage)
{
    Console.Write("AI: ");

    await foreach (var chunk in client.CallModelStreamAsync(
        model: "openai/gpt-4",
        userMessage: userMessage,
        maxTokens: 500
    ))
    {
        Console.Write(chunk);
        await Task.Delay(10); // Simulate typing animation
    }

    Console.WriteLine("\n");
}
```

Use Case 2: Building Complete Response

Sometimes you need both streaming UX and complete text:

```
var completeResponse = new StringBuilder();

await foreach (var chunk in client.CallModelStreamAsync(
    model: "openai/gpt-3.5-turbo",
    userMessage: "Explain quantum computing",
    maxTokens: 500
))
{
    Console.Write(chunk);      // Show to user
    completeResponse.Append(chunk);  // Save for processing
}

// Now you have both!
string fullText = completeResponse.ToString();
await SaveToDatabase(fullText);
```

Use Case 3: Progressive Web Response

For web applications:

```
public async IAsyncEnumerable<string> StreamToWeb(string prompt)
{
    await foreach (var chunk in client.CallModelStreamAsync(
        model: "openai/gpt-3.5-turbo",
        userMessage: prompt,
        maxTokens: 300
    ))
    {
        yield return chunk; // Forward to web client
    }
}
```

Then in your web framework (e.g., ASP.NET Core):

```
[HttpGet("stream")]
public async IAsyncEnumerable<string> StreamResponse(string prompt)
{
    await foreach (var chunk in aiService.StreamToWeb(prompt))
    {
        yield return chunk;
    }
}
```

Use Case 4: Real-Time Translation

```
Console.WriteLine("Translating to Spanish...\n");

await foreach (var chunk in client.CallModelStreamAsync(
    model: "openai/gpt-3.5-turbo",
    userMessage: "Translate to Spanish: The quick brown fox jumps over the lazy
dog",
    maxTokens: 100
))
{
    Console.Write(chunk);
}
```

Best Practices

1. Choose Appropriate Models

Best for streaming:

- `openai/gpt-3.5-turbo` - Fast responses
- `openai/gpt-4-turbo` - Balanced
- `anthropic/claude-instant` - Quick streaming

Avoid for streaming:

- Very slow models (defeats the purpose)
- Models with high first-token latency

2. Set Reasonable Token Limits

```
// Good - specific limit
maxTokens: 500

// Bad - unlimited (user waits forever)
maxTokens: null
```

3. Handle Partial Responses

Always consider: "What if the stream ends unexpectedly?"

```
var response = new StringBuilder();
bool completed = false;

try
{
    await foreach (var chunk in stream)
    {
        response.Append(chunk);
    }
    completed = true;
}
finally
{
    if (!completed)
    {
        LogPartialResponse(response.ToString());
    }
}
```

4. Provide Visual Feedback

In console apps:

```
Console.Write("AI: ");
```

In GUIs:

- Show typing indicator
- Animate text appearance
- Add cursor blink

5. Buffer for UI Performance

Don't update UI for every tiny chunk:

```
var buffer = new StringBuilder();
var lastUpdate = DateTime.Now;

await foreach (var chunk in stream)
{
    buffer.Append(chunk);

    // Update UI every 50ms, not every chunk
    if ((DateTime.Now - lastUpdate).TotalMilliseconds > 50)
    {
        UpdateUI(buffer.ToString());
        lastUpdate = DateTime.Now;
    }
}
```

6. Monitor and Log

```
var startTime = DateTime.Now;
var chunkCount = 0;

await foreach (var chunk in stream)
{
    chunkCount++;
    Console.Write(chunk);
}

var duration = DateTime.Now - startTime;
Console.WriteLine($"\n[Streamed {chunkCount} chunks in 
{duration.TotalSeconds}s]");
```

7. Graceful Cancellation

```
var cts = new CancellationTokenSource();

// User clicks "Stop" button
stopButton.Click += (s, e) => cts.Cancel();

try
{
    await foreach (var chunk in stream.WithCancellation(cts.Token))
    {
        Console.Write(chunk);
    }
}
catch (OperationCanceledException)
{
    Console.WriteLine("\n[Cancelled by user]");
}
```

Advanced Topics

Understanding CallModelStreamAsync

The helper method `CallModelStreamAsync` is a convenience wrapper. Under the hood, it does:

```
var request = new ChatCompletionRequest
{
    Model = model,
    Messages = new List<Message>
    {
        new UserMessage { Role = "user", Content = userMessage }
    },
    MaxTokens = maxTokens,
    Stream = true  // KEY: Enable streaming
};

return client.Chat.CreateStreamAsync(request);
```

Full Control Streaming

For advanced scenarios, use the full API:

```
var request = new ChatCompletionRequest
{
    Model = "openai/gpt-4",
    Messages = new List<Message>
    {
        new SystemMessage
        {
            Role = "system",
            Content = "You are a helpful assistant"
        },
        new UserMessage
        {
            Role = "user",
            Content = "Explain streaming"
        }
    },
    Temperature = 0.7,
    MaxTokens = 500,
    Stream = true
};

await foreach (var chunk in client.Chat.CreateStreamAsync(request))
{
    Console.Write(chunk.Choices[0].Delta.Content);
}
```

Common Issues and Solutions

Issue 1: No Output Appears

Symptom: Code runs but nothing prints

Cause: Using `Console.WriteLine` instead of `Console.Write`

Solution:

```
// Wrong - adds newline after each chunk
Console.WriteLine(chunk);

// Right - continuous output
Console.Write(chunk);
```

Issue 2: Output Appears All at Once

Symptom: Not actually streaming, just slower

Cause: Buffering somewhere in the stack

Solution: Ensure `Stream = true` in request

Issue 3: Chunks Arrive Slowly

Possible causes:

- Slow model chosen
- Network latency
- API under heavy load

Solutions:

- Switch to faster model
- Check internet connection
- Try again during off-peak hours

Issue 4: Stream Ends Abruptly

Cause: Token limit reached

Solution: Increase `maxTokens` or handle gracefully:

```
if (response.Choices[0].FinishReason == "length")
{
    Console.WriteLine("\n[Response truncated - increase maxTokens]");
}
```

Key Concepts Summary

Async Streams (IAsyncEnumerable)

- Asynchronous version of IEnumerable
- Produces values over time
- Perfect for real-time data

await foreach

- Special syntax for async iteration
- Automatically awaits each item
- Cleaner than manual async loops

Streaming Benefits

- Immediate user feedback
- Lower perceived latency
- Modern, responsive UX
- Can process data incrementally

Streaming Tradeoffs

- Slightly more complex code
- Need to handle partial responses
- Harder to retry/cancel
- Must consider network interruptions

Challenge Exercises

1. Modify the example to stream a longer response
2. Add a typing animation effect (delay between chunks)
3. Build a buffer that collects complete sentences before displaying
4. Implement a "Stop Generation" feature with cancellation
5. Add chunk counting and timing statistics

Additional Resources

- **OpenRouter Streaming Docs:** https://openrouter.ai/docs/streaming
- **C# Async Streams:** https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/generate-consume-asynchronous-stream
- **IAsyncEnumerable:** https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.iasyncenumerable-1
- **Server-Sent Events:** https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events

---

Conclusion

Congratulations! You've mastered streaming completions with OpenRouter SDK. You now understand:

✅ What streaming is and when to use it
✅ How async streams work with `await foreach`
✅ The difference between streaming and non-streaming
✅ How to handle errors in streaming scenarios

- ✅ Real-world use cases and best practices
- ✅ How to build responsive, modern AI applications

Streaming is a fundamental technique for building great AI user experiences. The patterns you learned here apply to chat applications, content generation, code assistants, and much more.

Keep experimenting and building!