Building Conversational AI: A Complete Guide to Chat Service Direct Access

Welcome to this comprehensive tutorial on using the OpenRouter Chat Service directly! This guide explores Example03.ChatServiceDirect, which demonstrates how to build structured conversations with AI models using system messages, accessing detailed response metadata, and understanding the full chat completion API.

Table of Contents

Overview

This example demonstrates how to interact directly with the Chat service API, giving you full control over the conversation structure. Unlike the simplified helper methods in previous examples, this approach lets you:

- Define system-level instructions that guide the AI's behavior
- Access detailed metadata about the response (model used, token counts)
- Build more sophisticated conversation patterns
- Understand the underlying API structure

What you'll learn:

- How to use the Chat service API directly
- The role and power of system messages
- How to structure conversations with multiple message types
- How to access and interpret response metadata
- Token usage tracking for cost management

- Building the foundation for multi-turn conversations

Project Structure

The Example03.ChatServiceDirect project maintains the same structure as previous examples:

```
Example03.ChatServiceDirect/
├── Example03.ChatServiceDirect.csproj  # Project configuration file
├── Program.cs                          # Main application code
└── BLOG.md                             # This guide
```

Dependencies:

- **OpenRouter.SDK**: The main SDK package
- **OpenRouter.Examples.EnvConfig**: Shared configuration library

---

Prerequisites

Before starting, ensure you have:

1. **.NET 8.0 SDK** installed (Download here)
2. **OpenRouter API Key** (Get one here)
3. **Completed Example01 and Example02** or understand basic SDK usage
4. Understanding of async/await patterns
5. Basic knowledge of chat-based AI interactions

New Concepts in This Example:

- **System Messages**: Instructions that guide AI behavior
- **Message Types**: User, Assistant, and System messages
- **Response Metadata**: Usage statistics and model information
- **Structured Conversations**: Building proper chat contexts

---

Understanding the Project File

The project file for Example03 is identical to previous examples:

```xml
<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>
```

Key Elements

PropertyGroup Section

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

- **OutputType**: Console application executable
- **TargetFramework**: .NET 8.0 for modern C# features
- **ImplicitUsings**: Automatic namespace imports
- **Nullable**: Enhanced null-safety analysis

Package References

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

The OpenRouter.SDK package provides:

- `OpenRouterClient` for API communication
- `ChatCompletionRequest` and `ChatCompletionResponse` models
- Message type classes (UserMessage, SystemMessage, AssistantMessage)
- Exception handling for API errors
- Full strongly-typed API access

Project References

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

The shared configuration project handles:

- API key management from .env files

- Environment variable fallback
- Default model configuration
- Centralized settings across all examples

Code Walkthrough

Let's analyze every line of `Program.cs` and understand the complete implementation:

The Complete Code

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

Console.WriteLine("============================================");
Console.WriteLine("Example 3: Using Chat Service Directly");
Console.WriteLine("============================================\n");

await Example03.RunAsync();

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

public static class Example03
{
    public static async Task RunAsync()
    {
        var apiKey = ExampleConfig.ApiKey;

        var client = new OpenRouterClient(apiKey);

        Console.WriteLine("=== Example 3: Using Chat Service Directly ===");
        try
        {
            var chatRequest = new ChatCompletionRequest
            {
                Model = ExampleConfig.ModelName,
                Messages = new List<Message>
                {
                    new SystemMessage { Content = "You are a helpful assistant."
},
                    new UserMessage { Content = "What is 2+2?" }
                }
            };

            var chatResponse = await client.Chat.CreateAsync(chatRequest);

            Console.WriteLine($"Model: {chatResponse.Model}");
            if (chatResponse.Choices?.Count > 0)
            {
                var message = chatResponse.Choices[0].Message;
                Console.WriteLine($"Response: {message.Content}");
```

```
            }

            if (chatResponse.Usage != null)
            {
                Console.WriteLine($"Tokens used:
{chatResponse.Usage.TotalTokens}");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}
```

Step-by-Step Breakdown

Step 1: Using Directives

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

New in Example03:

```
using OpenRouter.SDK.Models;
```

This explicitly imports the Models namespace, giving us direct access to:

- `ChatCompletionRequest`: The request object
- `ChatCompletionResponse`: The response object
- `Message` and its derived types: `UserMessage`, `SystemMessage`, `AssistantMessage`
- `Usage`: Token usage statistics
- `Choice`: Response choices structure

Why import explicitly?

While `ImplicitUsings` covers common namespaces, explicitly importing `OpenRouter.SDK.Models` makes the code clearer and ensures IntelliSense works properly for model classes.

Step 2: Welcome Banner

```
Console.WriteLine("============================================");
Console.WriteLine("Example 3: Using Chat Service Directly");
Console.WriteLine("============================================\n");
```

Identifies this example. The title "Using Chat Service Directly" emphasizes that we're working with the full Chat API, not simplified helpers.

Step 3: Execute the Example

```
await Example03.RunAsync();
```

Calls the main example method asynchronously. The pattern remains consistent across all examples.

Step 4: Initialize the Client

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

What's happening:

1. Retrieve API key from configuration (environment variables or .env file)
2. Create an `OpenRouterClient` instance

The client object provides access to different service endpoints:

- `client.Chat` - Chat completions
- `client.Models` - List available models
- `client.Embeddings` - Generate embeddings
- And more...

Step 5: Build the Request (THE CORE PART)

```
var chatRequest = new ChatCompletionRequest
{
    Model = ExampleConfig.ModelName,
    Messages = new List<Message>
    {
        new SystemMessage { Content = "You are a helpful assistant." },
        new UserMessage { Content = "What is 2+2?" }
    }
};
```

This is where Example03 differs significantly from previous examples. Let's break it down:

The Request Object:

```
var chatRequest = new ChatCompletionRequest
```

`ChatCompletionRequest` is a strongly-typed class that represents the complete API request. It includes properties like:

- `Model`: Which AI model to use
- `Messages`: The conversation history
- `Temperature`: Creativity level (0-2)
- `MaxTokens`: Response length limit
- `TopP`: Nucleus sampling parameter

- `Stream`: Enable streaming
- And many more advanced options

Model Selection:

```
Model = ExampleConfig.ModelName,
```

Specifies which AI model processes the request. Common options:

- `openai/gpt-4` - Most capable, best reasoning
- `openai/gpt-3.5-turbo` - Fast and economical
- `anthropic/claude-3-opus` - Excellent for analysis
- `google/gemini-pro` - Strong multimodal capabilities

The Messages Array:

```
Messages = new List<Message>
{
    new SystemMessage { Content = "You are a helpful assistant." },
    new UserMessage { Content = "What is 2+2?" }
}
```

This is the heart of the conversation. Let's understand each message:

System Message:

```
new SystemMessage { Content = "You are a helpful assistant." }
```

- **Type**: `SystemMessage`
- **Purpose**: Sets the AI's personality, behavior, and constraints
- **Position**: Typically first in the messages array
- **Visibility**: Not directly shown to the user, but influences all responses

System messages are powerful. They tell the AI:

- How to behave ("You are a helpful assistant")
- What constraints to follow ("Only answer in French")
- What role to play ("You are a Python expert")
- What format to use ("Always respond in JSON")

User Message:

```
new UserMessage { Content = "What is 2+2?" }
```

- **Type**: `UserMessage`
- **Purpose**: Represents input from the end user
- **Content**: The actual question or prompt

Message Flow:

```
System Message (invisible to user):
"You are a helpful assistant."
    ↓
AI internalizes this instruction
    ↓
User Message (visible):
"What is 2+2?"
    ↓
AI responds according to system instruction
    ↓
Assistant Message (response):
"2+2 equals 4."
```

Step 6: Send the Request

```
var chatResponse = await client.Chat.CreateAsync(chatRequest);
```

What happens:

1. The SDK serializes `chatRequest` to JSON
2. Sends HTTP POST to OpenRouter API
3. Waits for the response (async operation)
4. Deserializes JSON response to `ChatCompletionResponse` object
5. Returns the strongly-typed response

The async/await pattern:

- `await` pauses execution until the API responds
- Doesn't block the thread (efficient for I/O-bound operations)
- Exceptions from the API call can be caught in the try-catch block

Step 7: Display Model Information

```
Console.WriteLine($"Model: {chatResponse.Model}");
```

Why display the model?

The response includes which model actually processed your request. This is important because:

- OpenRouter might route to a different model if one is unavailable
- You requested a model family, and a specific version was used
- Debugging: Confirming the expected model was used

Example output:

```
Model: openai/gpt-3.5-turbo-0125
```

Step 8: Extract and Display the Response

```
if (chatResponse.Choices?.Count > 0)
{
    var message = chatResponse.Choices[0].Message;
    Console.WriteLine($"Response: {message.Content}");
}
```

Understanding the response structure:

```
chatResponse.Choices?.Count > 0
```

- **Choices**: Array of possible completions
- **? operator**: Null-conditional operator (safe navigation)
- **Count > 0**: Verify at least one choice exists

Why an array?

The API can return multiple responses (when `n > 1` in the request). By default, it returns one choice, but the API is designed to support multiple alternatives.

Extracting the message:

```
var message = chatResponse.Choices[0].Message;
```

- `[0]`: Gets the first (primary) choice
- `.Message`: The actual message object (type: `AssistantMessage`)

Displaying the content:

```
Console.WriteLine($"Response: {message.Content}");
```

The `Content` property contains the actual text response from the AI.

Step 9: Display Token Usage

```
if (chatResponse.Usage != null)
{
    Console.WriteLine($"Tokens used: {chatResponse.Usage.TotalTokens}");
}
```

New in Example03: Token usage tracking!

The `Usage` object contains:

- `PromptTokens`: Tokens in your input (system message + user message)
- `CompletionTokens`: Tokens in the AI's response
- `TotalTokens`: Sum of prompt and completion tokens

Why this matters:

- **Cost tracking**: API pricing is per token

- **Optimization**: Monitor and reduce token usage
- **Debugging**: Understand context window usage
- **Planning**: Estimate costs for production workloads

Example output:

```
Tokens used: 45
```

This means:

- Your input (system + user messages) might be ~25 tokens
- The response might be ~20 tokens
- Total billed: 45 tokens

Step 10: Error Handling

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

Catches any errors during the API call:

- Network failures
- Authentication errors
- Invalid requests
- Rate limiting
- Timeout issues

Best practice: In production, catch specific exception types for better error handling.

Understanding Message Types

The OpenRouter SDK provides three message types. Understanding when and how to use each is crucial for building effective AI applications.

1. SystemMessage

```
new SystemMessage { Content = "You are a helpful assistant." }
```

Purpose:

- Sets the AI's behavior and personality
- Defines constraints and rules
- Not part of the visible conversation
- Influences all subsequent responses

Common Uses:

Setting Personality:

```
new SystemMessage { Content = "You are a friendly customer service representative." }
```

Defining Expertise:

```
new SystemMessage { Content = "You are an expert Python developer with 10 years of experience." }
```

Setting Constraints:

```
new SystemMessage { Content = "You must respond only in valid JSON format." }
```

Defining Role:

```
new SystemMessage { Content = "You are a SQL expert. Help users write database queries." }
```

Setting Tone:

```
new SystemMessage { Content = "You are a professional business analyst. Use formal language." }
```

2. UserMessage

```
new UserMessage { Content = "What is 2+2?" }
```

Purpose:

- Represents input from the end user
- The actual questions or prompts
- What the user sees and types

Common Uses:

Questions:

```
new UserMessage { Content = "How do I connect to a database in C#?" }
```

Commands:

```
new UserMessage { Content = "Translate this to Spanish: Hello, how are you?" }
```

Data for Processing:

```
new UserMessage { Content = "Summarize this article: [long text]" }
```

3. AssistantMessage

```
new AssistantMessage { Content = "The answer is 4." }
```

Purpose:

- Represents previous AI responses
- Used in multi-turn conversations
- Provides context for follow-up questions

Common Uses:

Building Conversation History:

```
Messages = new List<Message>
{
    new UserMessage { Content = "What is 2+2?" },
    new AssistantMessage { Content = "2+2 equals 4." },
    new UserMessage { Content = "What about 3+3?" }
}
```

The AI sees the previous exchange and can provide contextual responses.

System Messages Explained

System messages are one of the most powerful features in chat-based AI. Let's explore them in depth.

How System Messages Work

System messages tell the AI:

- **Who** it is
- **How** it should behave
- **What** constraints to follow
- **Which** format to use

Effective System Message Patterns

Pattern 1: Role Definition

```
new SystemMessage
{
    Content = "You are a SQL database expert helping users write queries."
}
```

Effect: The AI will prioritize SQL knowledge and provide database-specific advice.

Pattern 2: Behavior Constraints

```
new SystemMessage
{
    Content = "You are a helpful assistant. Never write code longer than 20
lines. Always explain your code."
}
```

Effect: Enforces specific output constraints.

Pattern 3: Format Requirements

```
new SystemMessage
{
    Content = @"You are a JSON API. Always respond in this format:
{
    ""answer"": ""your answer here"",
    ""confidence"": 0.95
}"
}
```

Effect: Forces structured output that's easy to parse programmatically.

Pattern 4: Language and Tone

```
new SystemMessage
{
    Content = "You are a kindergarten teacher. Use simple words and be very
patient."
}
```

Effect: Adjusts language complexity and communication style.

Pattern 5: Domain Expertise

```
new SystemMessage
{
    Content = "You are a medical doctor specializing in cardiology. Provide
expert medical information."
}
```

Effect: The AI draws from medical knowledge and uses appropriate terminology.

System Message Best Practices

1. **Be Specific**: Vague instructions lead to inconsistent results
    - ❌ "Be helpful"
    - ✅ "You are a Python tutor. Explain concepts simply with code examples."
2. **Set Clear Boundaries**:

```
"You are a customer service bot. Only answer questions about products.
 If asked about other topics, politely redirect to product information."
```

3. **Define Output Format**:

```
"Always structure your response with: 1) Summary, 2) Details, 3) Next Steps"
```

4. **Keep It Concise**: Long system messages consume tokens
    - Aim for 1-3 sentences when possible
    - Be clear but brief
5. **Test Different Versions**: Small changes can have big impacts
    - Iterate to find what works best
    - A/B test different system prompts

Response Metadata

Example03 introduces response metadata - valuable information about how the API processed your request.

The Response Structure

```
ChatCompletionResponse
├── Model (string)
├── Choices (List<Choice>)
|   └── [0]
|       ├── Message (AssistantMessage)
|       |   └── Content (string)
|       ├── FinishReason (string)
|       └── Index (int)
├── Usage (Usage)
|   ├── PromptTokens (int)
|   ├── CompletionTokens (int)
|   └── TotalTokens (int)
├── Id (string)
└── Created (long)
```

Understanding Each Field

Model

```
chatResponse.Model
```

Example value: `"openai/gpt-3.5-turbo-0125"`

What it tells you:

- Exact model version that processed your request
- Useful for debugging and logging
- Confirms routing decisions

Choices

```
chatResponse.Choices[0]
```

Why an array?

You can request multiple responses:

```
var request = new ChatCompletionRequest
{
    Model = "openai/gpt-3.5-turbo",
    Messages = messages,
    N = 3  // Request 3 different responses
};
```

Then you get:

```
chatResponse.Choices[0].Message.Content  // First alternative
chatResponse.Choices[1].Message.Content  // Second alternative
chatResponse.Choices[2].Message.Content  // Third alternative
```

Message

```
chatResponse.Choices[0].Message
```

Type: `AssistantMessage`

Properties:

- `Content`: The actual text response
- `Role`: Always "assistant"
- `Name`: Optional name field

FinishReason

```
chatResponse.Choices[0].FinishReason
```

Possible values:

- `"stop"`: Natural completion (most common)
- `"length"`: Hit max token limit (response truncated)
- `"content_filter"`: Blocked by content policy
- `"function_call"`: Called a function (advanced usage)

Why it matters:

```
if (chatResponse.Choices[0].FinishReason == "length")
{
    Console.WriteLine("Warning: Response was cut off. Consider increasing
MaxTokens.");
}
```

Usage

```
chatResponse.Usage.PromptTokens       // Tokens in input
chatResponse.Usage.CompletionTokens   // Tokens in output
chatResponse.Usage.TotalTokens        // Total billed
```

Example:

```
Prompt: "What is 2+2?" (system + user messages)
PromptTokens: 20

Completion: "2+2 equals 4."
CompletionTokens: 8

TotalTokens: 28
```

Running the Example

Using Visual Studio

1. Open `OpenRouter.SDK.sln` in Visual Studio
2. Right-click `Example03.ChatServiceDirect` → Set as Startup Project
3. Press **F5** to run

Using .NET CLI

```
cd Examples/Example03.ChatServiceDirect
dotnet run
```

Using PowerShell

```
cd
"c:\Users\subhr\source\repos\OpenRouter.SDK\OpenRouter.SDK\Examples\Example03.ChatServiceDirect"
dotnet run
```

Expected Output

```
==========================================
Example 3: Using Chat Service Directly
==========================================

=== Example 3: Using Chat Service Directly ===
Model: openai/gpt-3.5-turbo-0125
Response: 2+2 equals 4.
Tokens used: 28

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

What to Look For

✅ **Success indicators:**

- Model name is displayed
- Response answers the question
- Token count is shown
- No error messages

📊 **Metadata insights:**

- Model name confirms which AI processed your request
- Token count helps estimate costs
- Response quality reflects system message influence

Comparing the Examples

Let's compare how Examples 01, 02, and 03 differ:

Example 01: Simple Text Completion

```
var request = new ChatCompletionRequest
{
    Model = ExampleConfig.ModelName,
    Messages = new List<Message>
    {
        new UserMessage
        {
            Role = "user",
            Content = "How many r's are in the word 'strawberry'?"
        }
    }
};

var response = await client.Chat.CreateAsync(request);
Console.WriteLine($"Response:\n{response.Choices[0].Message.Content}");
```

Focus: Basic request-response
Message types: User only
Metadata: Not examined
Error handling: Comprehensive

Example 02: Streaming Completion

```
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);
}
```

Focus: Real-time streaming
Message types: User only (helper method)
Metadata: Not accessible in streaming
User experience: Immediate feedback

Example 03: Chat Service Direct

```
var chatRequest = new ChatCompletionRequest
{
    Model = ExampleConfig.ModelName,
    Messages = new List<Message>
    {
        new SystemMessage { Content = "You are a helpful assistant." },
        new UserMessage { Content = "What is 2+2?" }
    }
};

var chatResponse = await client.Chat.CreateAsync(chatRequest);

Console.WriteLine($"Model: {chatResponse.Model}");
Console.WriteLine($"Response: {chatResponse.Choices[0].Message.Content}");
Console.WriteLine($"Tokens used: {chatResponse.Usage.TotalTokens}");
```

Focus: Full API control
Message types: System + User
Metadata: Full access to model, usage, etc.
Control: Maximum flexibility

When to Use Each

Use Example01 pattern when:

- Simple question-answer scenarios
- Don't need system messages
- Metadata not important
- Want simplest possible code

Use Example02 pattern when:

- Building chat interfaces
- User experience is priority
- Real-time feedback needed
- Long-form content generation

Use Example03 pattern when:

- Need system messages for behavior control
- Want to track token usage
- Building production applications
- Need full response metadata
- Foundation for multi-turn conversations

Building Multi-Turn Conversations

Example03 sets the foundation for multi-turn conversations. Here's how to extend it:

Pattern: Maintaining Conversation History

```
var conversationHistory = new List<Message>
{
    new SystemMessage { Content = "You are a helpful math tutor." }
};

// Turn 1
conversationHistory.Add(new UserMessage { Content = "What is 2+2?" });

var response1 = await client.Chat.CreateAsync(new ChatCompletionRequest
{
    Model = ExampleConfig.ModelName,
    Messages = conversationHistory
});

conversationHistory.Add(response1.Choices[0].Message);
Console.WriteLine($"AI: {response1.Choices[0].Message.Content}");

// Turn 2 - AI remembers previous context
conversationHistory.Add(new UserMessage { Content = "What about 3+3?" });

var response2 = await client.Chat.CreateAsync(new ChatCompletionRequest
{
    Model = ExampleConfig.ModelName,
    Messages = conversationHistory
});

conversationHistory.Add(response2.Choices[0].Message);
Console.WriteLine($"AI: {response2.Choices[0].Message.Content}");
```

Output:

```
AI: 2+2 equals 4.
AI: 3+3 equals 6.
```

The AI can reference previous messages because we include the full history.

Pattern: Simple Chat Loop

```
var messages = new List<Message>
{
    new SystemMessage { Content = "You are a helpful assistant." }
};

while (true)
{
    Console.Write("You: ");
    var userInput = Console.ReadLine();

    if (string.IsNullOrEmpty(userInput) || userInput == "quit")
        break;

    messages.Add(new UserMessage { Content = userInput });

    var response = await client.Chat.CreateAsync(new ChatCompletionRequest
    {
        Model = ExampleConfig.ModelName,
        Messages = messages
    });

    var aiMessage = response.Choices[0].Message;
    messages.Add(aiMessage);

    Console.WriteLine($"AI: {aiMessage.Content}\n");
}
```

Interactive conversation:

```
You: What is the capital of France?
AI: The capital of France is Paris.

You: What is it famous for?
AI: Paris is famous for the Eiffel Tower, the Louvre Museum, Notre-Dame
Cathedral...

You: quit
```

Token Usage and Costs

Understanding token usage is critical for managing costs in production applications.

What Are Tokens?

Tokens are pieces of words. Roughly:

- **1 token** ≈ 4 characters
- **1 token** ≈ 0.75 words

Examples:

- "Hello" = 1 token
- "Hello, how are you?" = 6 tokens
- "artificial intelligence" = 2-3 tokens

Calculating Costs

Example pricing (varies by model):

- GPT-3.5-turbo: $0.0015 per 1K prompt tokens, $0.002 per 1K completion tokens
- GPT-4: $0.03 per 1K prompt tokens, $0.06 per 1K completion tokens

From Example03 output:

```
Tokens used: 28
```

Cost breakdown:

```
Prompt tokens: ~20
Completion tokens: ~8

For GPT-3.5-turbo:
Prompt cost: 20 * $0.0015 / 1000 = $0.00003
Completion cost: 8 * $0.002 / 1000 = $0.000016
Total: ~$0.000046 (negligible)
```

For 1000 similar requests:

```
1000 * $0.000046 = $0.046 (about 5 cents)
```

Optimizing Token Usage

1. Keep system messages concise:

```
// Wasteful - 50+ tokens
new SystemMessage { Content = "You are an extremely helpful, friendly, and
knowledgeable assistant who always provides detailed, comprehensive, and accurate
responses..." }

// Efficient - ~10 tokens
new SystemMessage { Content = "You are a helpful assistant." }
```

2. Limit max tokens:

```
var request = new ChatCompletionRequest
{
    Model = ExampleConfig.ModelName,
    Messages = messages,
    MaxTokens = 100  // Prevent runaway responses
};
```

3. Trim conversation history:

```
// Keep only last 10 messages
if (conversationHistory.Count > 10)
{
    // Keep system message
    var systemMsg = conversationHistory[0];
    var recentMessages = conversationHistory.Skip(conversationHistory.Count -
9).ToList();
    conversationHistory = new List<Message> { systemMsg };
    conversationHistory.AddRange(recentMessages);
}
```

4. Monitor usage:

```
int totalTokensUsed = 0;

var response = await client.Chat.CreateAsync(request);
totalTokensUsed += response.Usage.TotalTokens;

Console.WriteLine($"Session total: {totalTokensUsed} tokens");
```

Best Practices

1. Always Include System Messages

```
// Good
Messages = new List<Message>
{
    new SystemMessage { Content = "You are a helpful coding assistant." },
    new UserMessage { Content = "How do I loop in C#?" }
}

// Less controlled - behavior may vary
Messages = new List<Message>
{
    new UserMessage { Content = "How do I loop in C#?" }
}
```

2. Check Response Before Using

```
// Safe
if (chatResponse.Choices?.Count > 0)
{
    var content = chatResponse.Choices[0].Message.Content;
    ProcessContent(content);
}

// Risky - could throw NullReferenceException
var content = chatResponse.Choices[0].Message.Content;
```

3. Monitor Token Usage

```
var response = await client.Chat.CreateAsync(request);

if (response.Usage != null)
{
    logger.LogInformation($"Tokens: {response.Usage.TotalTokens}");

    if (response.Usage.TotalTokens > 1000)
    {
        logger.LogWarning("High token usage detected");
    }
}
```

4. Handle Finish Reasons

```
var choice = chatResponse.Choices[0];

switch (choice.FinishReason)
{
    case "stop":
        // Normal completion
        break;
    case "length":
        Console.WriteLine("Response truncated. Consider increasing MaxTokens.");
        break;
    case "content_filter":
        Console.WriteLine("Response blocked by content filter.");
        break;
}
```

5. Structure Your Requests

```
// Create reusable request builder
private static ChatCompletionRequest BuildRequest(
    string systemPrompt,
    string userMessage,
    int maxTokens = 150)
{
    return new ChatCompletionRequest
```

```
    {
        Model = ExampleConfig.ModelName,
        Messages = new List<Message>
        {
            new SystemMessage { Content = systemPrompt },
            new UserMessage { Content = userMessage }
        },
        MaxTokens = maxTokens,
        Temperature = 0.7
    };
}
```

6. Log Important Metadata

```
var response = await client.Chat.CreateAsync(request);

logger.LogInformation(new
{
    Model = response.Model,
    TokensUsed = response.Usage?.TotalTokens,
    FinishReason = response.Choices?[0].FinishReason,
    Timestamp = DateTime.UtcNow
});
```

Common Issues and Solutions

Issue 1: System Message Not Working

Symptom: AI doesn't follow system instructions

Causes:

- System message too vague
- Conflicting user instructions
- Model limitations

Solutions:

```
// Be more specific
new SystemMessage
{
    Content = "You must respond only in JSON. No other text allowed."
}

// Add examples
new SystemMessage
{
    Content = @"You are a JSON API. Example response:
{""answer"": ""42"", ""confidence"": 0.95}"
}
```

Issue 2: High Token Usage

Symptom: Costs higher than expected

Solution:

```
// Set limits
var request = new ChatCompletionRequest
{
    Model = "openai/gpt-3.5-turbo",  // Use cheaper model
    Messages = messages,
    MaxTokens = 100,  // Limit response length
    Temperature = 0.3  // More focused responses
};

// Monitor
Console.WriteLine($"Tokens: {response.Usage.TotalTokens}");
```

Issue 3: Null Reference Errors

Symptom: Application crashes

Solution:

```
// Defensive coding
if (chatResponse?.Choices?.Count > 0)
{
    var message = chatResponse.Choices[0]?.Message;
    if (message?.Content != null)
    {
        Console.WriteLine(message.Content);
    }
}
```

Issue 4: Inconsistent Responses

Symptom: Different results for same input

Cause: Temperature setting (randomness)

Solution:

```
var request = new ChatCompletionRequest
{
    Model = ExampleConfig.ModelName,
    Messages = messages,
    Temperature = 0  // Deterministic responses
};
```

Challenge Exercises

1. Modify the system message to make the AI respond as a pirate
2. Add a follow-up question that references the first response
3. Track and display token usage for a 5-turn conversation
4. Implement error handling for different FinishReason values
5. Create a system message that enforces JSON output format

Additional Resources

- **OpenRouter Chat API Docs:** https://openrouter.ai/docs/chat
- **Message Types Guide:** https://openrouter.ai/docs/messages
- **Token Counting:** https://platform.openai.com/tokenizer
- **Model Pricing:** https://openrouter.ai/models
- **Best Practices:** https://openrouter.ai/docs/best-practices

---

This knowledge forms the foundation for building sophisticated AI-powered applications. The direct Chat API access gives you the control needed for complex scenarios while maintaining clean, maintainable code.

Happy coding!