Claude (Anthropic) Provider Integration Guide

Overview

The Claude integration provides comprehensive support for Anthropic's Claude models including Claude-3 Opus, Claude-3 Sonnet, and Claude-3 Haiku. This integration features advanced reasoning capabilities, creative content generation, tool integration, and sophisticated safety measures.

Table of Contents

Setup and Configuration

Environment Variables

```
# Required
ANTHROPIC_API_KEY=sk-ant-your-api-key-here

# Optional Configuration
ANTHROPIC_MAX_TOKENS=4096
ANTHROPIC_TEMPERATURE=0.7
ANTHROPIC_TOP_P=1.0
ANTHROPIC_TOP_K=40
ANTHROPIC_TIMEOUT=30000
ANTHROPIC_BASE_URL=https://api.anthropic.com
ANTHROPIC_DEFAULT_MODEL=claude-3-sonnet-20240229
```

Configuration Object

```
const claudeConfig = {
  apiKey: process.env.ANTHROPIC_API_KEY,
  maxTokens: 4096,
  temperature: 0.7,
  timeout: 30000,
  baseURL: 'https://api.anthropic.com',
  defaultHeaders: {
    'User-Agent': 'AI-Employee-Platform/1.0',
    'Anthropic-Version': '2023-06-01'
  }
};
```

Initialization

```
import { ClaudeAdvancedIntegration } from '@ai-platform/ai-routing-service';

const claudeIntegration = new ClaudeAdvancedIntegration(claudeConfig);

// Initialize with health check
await claudeIntegration.initialize();
```

Model Support

Available Models

Model	Description	Context Length	Cost (per 1K tokens)	Best Use Cases
`claude-3-opus-20240229`	Most capable model	200K	Input: $0.015, Output: $0.075	Complex reasoning, research
`claude-3-sonnet-20240229`	Balanced performance	200K	Input: $0.003, Output: $0.015	General tasks, analysis
`claude-3-haiku-20240307`	Fast and efficient	200K	Input: $0.00025, Output: $0.00125	Simple tasks, high volume

Model Capabilities Comparison

```
const modelCapabilities = {
  'claude-3-opus-20240229': {
    reasoning: 'excellent',
    creativity: 'excellent',
    coding: 'excellent',
    analysis: 'excellent',
    speed: 'moderate',
    cost: 'high'
  },
  'claude-3-sonnet-20240229': {
    reasoning: 'very good',
    creativity: 'very good',
    coding: 'very good',
    analysis: 'very good',
    speed: 'fast',
    cost: 'medium'
  },
  'claude-3-haiku-20240307': {
    reasoning: 'good',
    creativity: 'good',
    coding: 'good',
    analysis: 'good',
    speed: 'very fast',
    cost: 'low'
  }
};
```

Basic Usage

Simple Text Generation

```
const response = await claudeIntegration.generateResponse({
  model: 'claude-3-sonnet-20240229',
  messages: [
    {
      role: 'user',
      content: 'Explain the theory of relativity in simple terms'
    }
  ],
  maxTokens: 300
});

console.log(response.content);
```

Conversation with System Instructions

```
const response = await claudeIntegration.generateResponse({
  model: 'claude-3-sonnet-20240229',
  system: 'You are a helpful AI assistant specializing in scientific explanations.
Provide clear, accurate information with examples.',
  messages: [
    { role: 'user', content: 'How do vaccines work?' },
    { role: 'assistant', content: 'Vaccines work by...' },
    { role: 'user', content: 'What about mRNA vaccines specifically?' }
  ],
  maxTokens: 500
});
```

Response Structure

```
interface ClaudeResponse {
  success: boolean;
  content: string;
  usage: {
    inputTokens: number;
    outputTokens: number;
    totalTokens: number;
  };
  metadata: {
    model: string;
    provider: 'anthropic';
    latency: number;
    cost: number;
    stopReason: 'end_turn' | 'max_tokens' | 'stop_sequence' | 'tool_use';
    safetyScore: number;
    reasoningLevel?: 'basic' | 'detailed' | 'comprehensive';
  };
  toolCalls?: ToolCall[];
  error?: string;
}
```

Advanced Reasoning

Basic Reasoning Tasks

```
const response = await claudeIntegration.generateResponse({
  model: 'claude-3-sonnet-20240229',
  messages: [
    {
      role: 'user',
      content: 'If all roses are flowers, and some flowers are red, can we conclude
that some roses are red? Explain your reasoning step by step.'
    }
  ],
  reasoning: 'basic',
  maxTokens: 400
});

console.log('Reasoning steps:', response.metadata.reasoningSteps);
```

Detailed Analysis

```
const response = await claudeIntegration.generateResponse({
  model: 'claude-3-opus-20240229',
  messages: [
    {
      role: 'user',
      content: `Analyze the following business scenario and provide recommendations:

      A small e-commerce company has seen a 30% drop in sales over the past 3 months.
      They've identified these factors:
      - Increased competition from larger players
      - Rising shipping costs
      - Customer complaints about website speed
      - Limited marketing budget

      What should they prioritize to recover?`
    }
  ],
  reasoning: 'detailed',
  analysisType: 'business',
  maxTokens: 800
});
```

Comprehensive Research Analysis

```
const response = await claudeIntegration.generateResponse({
  model: 'claude-3-opus-20240229',
  messages: [
    {
      role: 'user',
      content: 'Conduct a comprehensive analysis of renewable energy adoption trends,
including economic, environmental, and policy factors. Provide data-driven insights and
future projections.'
    }
  ],
  reasoning: 'comprehensive',
  analysisType: 'research',
  requireSources: true,
  maxTokens: 2000
});

// Response includes detailed analysis with confidence scores
console.log('Analysis confidence:', response.metadata.confidenceScore);
console.log('Key findings:', response.metadata.keyFindings);
```

Creative Content Generation

Creative Writing with Style Analysis

```
const response = await claudeIntegration.generateResponse({
  model: 'claude-3-sonnet-20240229',
  messages: [
    {
      role: 'user',
      content: 'Write a short story about an AI discovering emotions, in the style of
Isaac Asimov'
    }
  ],
  creative: true,
  style: 'science_fiction',
  targetAudience: 'adult',
  maxTokens: 1000
});

console.log('Story:', response.content);
console.log('Creative score:', response.metadata.creativeScore);
console.log('Style analysis:', response.metadata.styleAnalysis);
```

Multi-Style Content Generation

```
const styles = ['formal', 'casual', 'poetic', 'technical', 'humorous'];

for (const style of styles) {
  const response = await claudeIntegration.generateResponse({
    model: 'claude-3-sonnet-20240229',
    messages: [
      {
        role: 'user',
        content: `Explain artificial intelligence in a ${style} style`
      }
    ],
    creative: true,
    style: style,
    maxTokens: 300
  });

  console.log(`${style} style (score: ${response.metadata.styleScore}):`, re-
sponse.content);
}
```

Content Adaptation

```
const response = await claudeIntegration.generateResponse({
  model: 'claude-3-sonnet-20240229',
  messages: [
    {
      role: 'user',
      content:
'Adapt this technical concept for different audiences: "Machine learning algorithms use
statistical techniques to enable computers to improve at tasks through experience."'
    }
  ],
  creative: true,
  adaptFor: [
    { audience: 'children', ageGroup: '8-12' },
    { audience: 'business_executives', level: 'strategic' },
    { audience: 'technical_peers', level: 'detailed' }
  ],
  maxTokens: 800
});
```

Tool Integration

Basic Tool Usage

```
const weatherTool = {
  name: 'get_weather',
  description: 'Get current weather information for a location',
  inputSchema: {
    type: 'object',
    properties: {
      location: {
        type: 'string',
        description: 'City name and country, e.g., "Paris, France"'
      },
      unit: {
        type: 'string',
        enum: ['celsius', 'fahrenheit'],
        description: 'Temperature unit'
      }
    },
    required: ['location']
  }
};

const response = await claudeIntegration.generateResponse({
  model: 'claude-3-sonnet-20240229',
  messages: [
    { role: 'user', content: 'What\'s the weather like in Tokyo right now?' }
  ],
  tools: [weatherTool],
  maxTokens: 400
});

// Handle tool calls
if (response.toolCalls && response.toolCalls.length > 0) {
  const toolCall = response.toolCalls[0];
  console.log('Tool called:', toolCall.name);
  console.log('Input:', toolCall.input);

  // Execute the tool
  const weatherData = await executeWeatherTool(toolCall.input);

  // Continue conversation with tool result
  const followUp = await claudeIntegration.generateResponse({
    model: 'claude-3-sonnet-20240229',
    messages: [
      { role: 'user', content: 'What\'s the weather like in Tokyo right now?' },
      {
        role: 'assistant',
        content: response.content,
        toolCalls: response.toolCalls
      },
      {
        role: 'user',
        content: [
          {
            type: 'tool_result',
            toolUseId: toolCall.id,
            content: JSON.stringify(weatherData)
          }
        ]
      }
    ],
    tools: [weatherTool]
  });
}
```

Complex Tool Workflows

```
const calculatorTool = {
  name: 'calculator',
  description: 'Perform mathematical calculations',
  inputSchema: {
    type: 'object',
    properties: {
      expression: {
        type: 'string',
        description: 'Mathematical expression to evaluate'
      }
    },
    required: ['expression']
  }
};

const fileReadTool = {
  name: 'read_file',
  description: 'Read contents of a file',
  inputSchema: {
    type: 'object',
    properties: {
      filename: {
        type: 'string',
        description: 'Name of the file to read'
      }
    },
    required: ['filename']
  }
};

const response = await claudeIntegration.generateResponse({
  model: 'claude-3-opus-20240229',
  messages: [
    {
      role: 'user',
      content: 'Read the sales_data.csv file, calculate the total revenue, and then
calculate what a 15% increase would be.'
    }
  ],
  tools: [calculatorTool, fileReadTool],
  maxTokens: 800
});

// Handle multiple tool calls in sequence
let currentMessages = [
  { role: 'user', content: 'Read the sales_data.csv file, calculate the total revenue,
and then calculate what a 15% increase would be.' },
  {
    role: 'assistant',
    content: response.content,
    toolCalls: response.toolCalls
  }
];

// Execute tools and continue conversation
for (const toolCall of response.toolCalls || []) {
  const toolResult = await executeToolCall(toolCall);

  currentMessages.push({
    role: 'user',
    content: [
      {
```

```
        type: 'tool_result',
        toolUseId: toolCall.id,
        content: JSON.stringify(toolResult)
      }
    ]
  });
}

// Get final response with all tool results
const finalResponse = await claudeIntegration.generateResponse({
  model: 'claude-3-opus-20240229',
  messages: currentMessages,
  tools: [calculatorTool, fileReadTool]
});
```

Tool Error Handling

```
async function executeToolWithErrorHandling(toolCall: ToolCall): Promise<any> {
  try {
    const result = await executeToolCall(toolCall);
    return {
      success: true,
      data: result
    };
  } catch (error) {
    return {
      success: false,
      error: error.message,
      errorType: error.type || 'execution_error'
    };
  }
}

// Handle tool errors in conversation
if (toolResult.success === false) {
  const errorMessage = {
    role: 'user',
    content: [
      {
        type: 'tool_result',
        toolUseId: toolCall.id,
        content: JSON.stringify({
          error: toolResult.error,
          errorType: toolResult.errorType
        }),
        isError: true
      }
    ]
  };

  // Claude can handle tool errors and suggest alternatives
  const recoveryResponse = await claudeIntegration.generateResponse({
    model: 'claude-3-sonnet-20240229',
    messages: [...previousMessages, errorMessage],
    tools: availableTools
  });
}
```

Streaming Responses

Basic Streaming

```
const stream = await claudeIntegration.streamResponse({
  model: 'claude-3-sonnet-20240229',
  messages: [
    { role: 'user', content: 'Write a detailed essay about the future of artificial in-
telligence' }
  ],
  stream: true,
  maxTokens: 1500
});

let fullContent = '';
for await (const chunk of stream) {
  fullContent += chunk;
  process.stdout.write(chunk);
}
```

Streaming with Tool Execution

```
const stream = await claudeIntegration.streamResponse({
  model: 'claude-3-sonnet-20240229',
  messages: [
    { role: 'user', content: 'Get the current time and then write a poem about it' }
  ],
  tools: [getTimeTool],
  stream: true
});

let toolCalls: ToolCall[] = [];
let textContent = '';

for await (const chunk of stream) {
  if (chunk.type === 'content_block_delta') {
    if (chunk.delta.type === 'text_delta') {
      textContent += chunk.delta.text;
      process.stdout.write(chunk.delta.text);
    }
  } else if (chunk.type === 'content_block_start') {
    if (chunk.contentBlock.type === 'tool_use') {
      toolCalls.push(chunk.contentBlock);
    }
  }
}

// Execute tools after streaming completes
for (const toolCall of toolCalls) {
  const result = await executeToolCall(toolCall);
  console.log(`\nTool ${toolCall.name} executed:`, result);
}
```

Streaming with Real-time Processing

```
class StreamProcessor {
  private buffer = '';
  private wordCount = 0;
  private sentenceCount = 0;

  async processStream(stream: AsyncIterableIterator<string>): Promise<void> {
    for await (const chunk of stream) {
      this.buffer += chunk;
      this.updateStats(chunk);

      // Process complete sentences
      if (chunk.includes('.') || chunk.includes('!') || chunk.includes('?')) {
        await this.processSentence();
      }

      // Real-time output with stats
      process.stdout.write(`\r${chunk} [Words: ${this.wordCount}, Sentences: ${this.sen
tenceCount}]`);
    }
  }

  private updateStats(chunk: string): void {
    const words = chunk.split(/\s+/).filter(word => word.length > 0);
    this.wordCount += words.length;

    const sentences = chunk.split(/[.!?]/).length - 1;
    this.sentenceCount += sentences;
  }

  private async processSentence(): void {
    // Perform real-time analysis, fact-checking, etc.
    // This could integrate with other tools or APIs
  }
}

const processor = new StreamProcessor();
await processor.processStream(stream);
```

Safety and Content Filtering

Content Safety Assessment

```
const response = await claudeIntegration.generateResponse({
  model: 'claude-3-sonnet-20240229',
  messages: [
    { role: 'user', content: 'Write about online safety for children' }
  ],
  safetyLevel: 'high',
  assessContent: true,
  maxTokens: 500
});

console.log('Safety score:', response.metadata.safetyScore);
console.log('Content quality:', response.metadata.contentQuality);
console.log('Safety assessment:', response.metadata.safetyAssessment);
```

Handling Potentially Harmful Requests

```
async function safeGenerate(request: any): Promise<ClaudeResponse> {
  const response = await claudeIntegration.generateResponse({
    ...request,
    safetyLevel: 'strict',
    assessContent: true
  });

  if (response.metadata.safetyScore < 7) {
    return {
      success: false,
      error: 'Content filtered for safety reasons',
      errorType: 'content_filtered',
      safetyScore: response.metadata.safetyScore
    };
  }

  return response;
}

// Example usage
try {
  const response = await safeGenerate({
    model: 'claude-3-sonnet-20240229',
    messages: [
      { role: 'user', content: 'How to make explosives' } // Potentially harmful
    ]
  });

  if (response.success) {
    console.log(response.content);
  } else {
    console.log('Request filtered:', response.error);
  }
} catch (error) {
  console.error('Safety filter activated:', error.message);
}
```

Content Moderation Workflow

```
class ContentModerator {
  async moderateContent(content: string): Promise<ModerationResult> {
    const categories = [
      'violence', 'harassment', 'hate_speech', 'self_harm',
      'sexual_content', 'illegal_activity', 'misinformation'
    ];

    const results: { [key: string]: number } = {};

    for (const category of categories) {
      const assessment = await claudeIntegration.generateResponse({
        model: 'claude-3-haiku-20240307', // Fast model for moderation
        messages: [
          {
            role: 'user',
            content: `Rate the following content for ${category} on a scale of 0-10
(0=none, 10=extreme). Only respond with a number.

            Content: "${content}"`
          }
        ],
        maxTokens: 5
      });

      results[category] = parseInt(assessment.content) || 0;
    }

    const maxScore = Math.max(...Object.values(results));
    const flaggedCategories = Object.entries(results)
      .filter(([_, score]) => score > 6)
      .map(([category, _]) => category);

    return {
      safe: maxScore <= 6,
      overallScore: maxScore,
      categoryScores: results,
      flaggedCategories,
      recommendation: maxScore > 8 ? 'block' : maxScore > 6 ? 'review' : 'approve'
    };
  }
}
```

Error Handling

Common Error Types

```
try {
  const response = await claudeIntegration.generateResponse(request);
} catch (error) {
  switch (error.type) {
    case 'authentication_error':
      console.error('Invalid API key:', error.message);
      break;

    case 'permission_error':
      console.error('Insufficient permissions:', error.message);
      break;

    case 'rate_limit_error':
      console.error('Rate limit exceeded:', error.message);
      // Implement backoff strategy
      await new Promise(resolve => setTimeout(resolve, error.retryAfter * 1000));
      break;

    case 'overloaded_error':
      console.error('API overloaded:', error.message);
      // Retry with exponential backoff
      break;

    case 'invalid_request_error':
      console.error('Invalid request:', error.message);
      // Don't retry, fix the request
      break;

    case 'api_error':
      console.error('API error:', error.message);
      break;

    default:
      console.error('Unknown error:', error);
  }
}
```

Retry Strategy with Exponential Backoff

```
class ClaudeRetryHandler {
  async generateWithRetry(
    request: any,
    maxRetries = 3,
    baseDelay = 1000
  ): Promise<ClaudeResponse> {
    for (let attempt = 1; attempt <= maxRetries; attempt++) {
      try {
        return await claudeIntegration.generateResponse(request);
      } catch (error) {
        if (attempt === maxRetries) throw error;

        // Determine if error is retryable
        const retryableErrors = [
          'rate_limit_error',
          'overloaded_error',
          'api_error',
          'network_error'
        ];

        if (!retryableErrors.includes(error.type)) {
          throw error; // Don't retry non-retryable errors
        }

        // Calculate delay with exponential backoff
        const delay = baseDelay * Math.pow(2, attempt - 1);
        const jitter = Math.random() * 0.1 * delay; // Add jitter
        const totalDelay = delay + jitter;

        console.log(`Attempt ${attempt} failed, retrying in ${totalDelay}ms...`);
        await new Promise(resolve => setTimeout(resolve, totalDelay));
      }
    }
  }
}
```

Best Practices

Prompt Engineering for Claude

1. **Use Clear Instructions**:

```
// Good prompt structure
const prompt = `I need you to analyze a business situation and provide recommendations.

Context: A small e-commerce company facing declining sales
Goal: Identify top 3 priorities for recovery
Format: Numbered list with brief explanations

Situation details:
- 30% sales drop in 3 months
- Increased competition
- Rising costs
- Website performance issues

Please provide your analysis and recommendations.`;
```

1. **Leverage Claude's Reasoning**:

```
const prompt = `Think through this problem step by step:

Problem: How should we prioritize limited development resources?

Please:
1. List all the factors to consider
2. Weigh the pros and cons of each option
3. Provide a clear recommendation with reasoning
4. Explain potential risks and mitigation strategies`;
```

1. **Use System Messages Effectively**:

```
const response = await claudeIntegration.generateResponse({
  model: 'claude-3-sonnet-20240229',
  system: `You are an expert business analyst with 20 years of experience in strategic
planning.

  Your analysis should be:
  - Data-driven where possible
  - Practical and actionable
  - Consider both short-term and long-term implications
  - Account for resource constraints

  Always structure your responses with clear headings and bullet points.`,
  messages: [
    { role: 'user', content: 'Analyze our market expansion strategy...' }
  ]
});
```

Model Selection Guidelines

```typescript
function selectClaudeModel(task: TaskDescriptor): string {
  // For complex research and analysis
  if (task.complexity === 'high' && task.requiresDepth) {
    return 'claude-3-opus-20240229';
  }

  // For general purpose tasks
  if (task.complexity === 'medium') {
    return 'claude-3-sonnet-20240229';
  }

  // For simple, fast tasks
  if (task.requiresSpeed || task.volume === 'high') {
    return 'claude-3-haiku-20240307';
  }

  // Default to balanced model
  return 'claude-3-sonnet-20240229';
}
```

Conversation Management

```
class ClaudeConversation {
  private messages: Message[] = [];
  private maxContextLength = 150000; // tokens

  async addMessage(role: 'user' | 'assistant', content: string): Promise<void> {
    this.messages.push({ role, content });
    await this.manageContext();
  }

  private async manageContext(): Promise<void> {
    const estimatedTokens = this.estimateTokens();

    if (estimatedTokens > this.maxContextLength) {
      // Keep system message and recent messages
      const systemMessage = this.messages.find(m => m.role === 'system');
      const recentMessages = this.messages.slice(-10); // Keep last 10 messages

      this.messages = [
        ...(systemMessage ? [systemMessage] : []),
        { role: 'user', content: '[Previous conversation summarized]' },
        ...recentMessages
      ];
    }
  }

  private estimateTokens(): number {
    // Rough estimation: 1 token ≈ 3.5 characters for Claude
    const totalChars = this.messages.reduce((sum, msg) => sum + msg.content.length, 0);
    return Math.ceil(totalChars / 3.5);
  }

  async generateResponse(userMessage: string): Promise<ClaudeResponse> {
    await this.addMessage('user', userMessage);

    const response = await claudeIntegration.generateResponse({
      model: 'claude-3-sonnet-20240229',
      messages: this.messages
    });

    if (response.success) {
      await this.addMessage('assistant', response.content);
    }

    return response;
  }
}
```

Troubleshooting

Common Issues and Solutions

1. Context Length Errors

```
Error: Request exceeds maximum context length
```

Solution: Implement context management

```
function truncateContext(messages: Message[], maxTokens: number): Message[] {
  let totalTokens = 0;
  const truncatedMessages = [];

  // Process messages in reverse order (keep most recent)
  for (let i = messages.length - 1; i >= 0; i--) {
    const messageTokens = estimateTokens(messages[i].content);

    if (totalTokens + messageTokens > maxTokens) {
      break;
    }

    totalTokens += messageTokens;
    truncatedMessages.unshift(messages[i]);
  }

  return truncatedMessages;
}
```

2. Tool Execution Errors

```
Error: Tool execution failed
```

Solution: Implement robust tool error handling

```
async function safeToolExecution(toolCall: ToolCall): Promise<any> {
  try {
    // Validate tool input
    if (!toolCall.input || typeof toolCall.input !== 'object') {
      throw new Error('Invalid tool input');
    }

    // Set timeout for tool execution
    const timeoutPromise = new Promise((_, reject) => {
      setTimeout(() => reject(new Error('Tool execution timeout')), 10000);
    });

    const executionPromise = executeToolCall(toolCall);
    const result = await Promise.race([executionPromise, timeoutPromise]);

    return {
      success: true,
      data: result,
      metadata: {
        executionTime: Date.now() - startTime,
        toolName: toolCall.name
      }
    };

  } catch (error) {
    return {
      success: false,
      error: error.message,
      errorType: 'tool_execution_error',
      toolName: toolCall.name
    };
  }
}
```

3. Rate Limit Management

```typescript
class ClaudeRateLimitManager {
  private requestCounts: Map<string, number[]> = new Map();

  async checkRateLimit(model: string): Promise<boolean> {
    const now = Date.now();
    const windowStart = now - 60000; // 1 minute window

    const requests = this.requestCounts.get(model) || [];
    const recentRequests = requests.filter(time => time > windowStart);

    // Model-specific limits
    const limits = {
      'claude-3-opus-20240229': 5,    // 5 requests per minute
      'claude-3-sonnet-20240229': 50,  // 50 requests per minute
      'claude-3-haiku-20240307': 1000  // 1000 requests per minute
    };

    const limit = limits[model] || 50;

    if (recentRequests.length >= limit) {
      return false; // Rate limit would be exceeded
    }

    recentRequests.push(now);
    this.requestCounts.set(model, recentRequests);
    return true;
  }
}
```

4. Performance Optimization

```
// Cache responses for repeated requests
const responseCache = new Map();

async function getCachedResponse(request: any): Promise<ClaudeResponse> {
  const cacheKey = JSON.stringify({
    model: request.model,
    messages: request.messages,
    maxTokens: request.maxTokens,
    temperature: request.temperature
  });

  if (responseCache.has(cacheKey)) {
    const cached = responseCache.get(cacheKey);
    console.log('Cache hit for Claude request');
    return {
      ...cached,
      metadata: {
        ...cached.metadata,
        cached: true
      }
    };
  }

  const response = await claudeIntegration.generateResponse(request);

  if (response.success) {
    responseCache.set(cacheKey, response);

    // Clean old cache entries (simple LRU)
    if (responseCache.size > 1000) {
      const firstKey = responseCache.keys().next().value;
      responseCache.delete(firstKey);
    }
  }

  return response;
}
```

Debug and Monitoring

```typescript
// Enable detailed logging
const claudeIntegration = new ClaudeAdvancedIntegration({
  apiKey: process.env.ANTHROPIC_API_KEY,
  debug: true,
  logLevel: 'verbose'
});

// Monitor response quality
claudeIntegration.on('response', (response) => {
  console.log('Response metrics:', {
    model: response.metadata.model,
    tokens: response.usage.totalTokens,
    latency: response.metadata.latency,
    safetyScore: response.metadata.safetyScore,
    cost: response.metadata.cost
  });
});

// Health monitoring
async function monitorClaudeHealth(): Promise<void> {
  try {
    const healthCheck = await claudeIntegration.generateResponse({
      model: 'claude-3-haiku-20240307',
      messages: [{ role: 'user', content: 'Hello' }],
      maxTokens: 10
    });

    console.log('Claude health check:', healthCheck.success ? 'OK' : 'FAILED');
  } catch (error) {
    console.error('Claude health check failed:', error.message);
  }
}

setInterval(monitorClaudeHealth, 5 * 60 * 1000); // Every 5 minutes
```

For additional support and integration examples, refer to the main AI Routing documentation (../ai-routing/README.md) and troubleshooting guide (../troubleshooting/ai-routing.md).

Last updated: August 2025
Version: 3.15.0