Usage Examples

Comprehensive examples demonstrating how to use the AI Abstraction Layer in real-world scenarios.

Table of Contents

Basic Examples

Simple Text Generation

```
import { ConfigFactory, AIServiceFactory, LLMService } from './lib/ai';

async function basicTextGeneration() {
  // Create configuration
  const config = ConfigFactory.openai.llm(process.env.OPENAI_API_KEY!, {
    defaultModel: 'gpt-4',
    defaultTemperature: 0.7,
    defaultMaxTokens: 500
  });

  // Create service factory and LLM service
  const factory = new AIServiceFactory();
  const llmService = await factory.createLLMService(config);

  // Generate text
  const response = await llmService.generate({
    messages: [
      { role: 'system', content: 'You are a helpful AI assistant.' },
      { role: 'user', content: 'Explain quantum computing in simple terms.' }
    ],
    maxTokens: 300,
    temperature: 0.7
  });

  if (response.success) {
    console.log('Generated text:', response.data.choices[0].message.content);
    console.log('Tokens used:', response.data.usage.totalTokens);
  } else {
    console.error('Generation failed:', response.error.message);
  }
}
```

Basic Embeddings

```
import { ConfigFactory, EmbeddingsService } from './lib/ai';

async function basicEmbeddings() {
  const config = ConfigFactory.voyageai.embeddings(process.env.VOYAGEAI_API_KEY!, {
    defaultModel: 'voyage-2',
    batchSize: 50
  });

  const factory = new AIServiceFactory();
  const embeddingsService = await factory.createEmbeddingsService(config);

  // Single text embedding
  const singleResponse = await embeddingsService.embedText(
    'The quick brown fox jumps over the lazy dog'
  );

  if (singleResponse.success) {
    console.log('Embedding dimensions:', singleResponse.data.length);
  }

  // Batch embeddings
  const texts = [
    'Machine learning is a subset of artificial intelligence',
    'Python is a popular programming language',
    'Natural language processing deals with text analysis'
  ];

  const batchResponse = await embeddingsService.embedTexts(texts);

  if (batchResponse.success) {
    console.log('Batch embeddings created:', batchResponse.data.length);

    // Calculate similarity between first two texts
    const similarity = embeddingsService.calculateSimilarity(
      batchResponse.data[0],
      batchResponse.data[1]
    );
    console.log('Similarity score:', similarity);
  }
}
```

Streaming Examples

Basic Streaming

```javascript
async function basicStreaming() {
  const llmService = await createLLMService();

  const stream = await llmService.generateStream({
    messages: [
      { role: 'user', content: 'Write a short story about a robot learning to paint.' }
    ],
    model: 'gpt-4',
    maxTokens: 1000,
    streamingOptions: {
      onStart: (context) => {
        console.log('Stream started:', context.requestId);
      },
      onProgress: (status, chunk) => {
        if (chunk?.choices[0]?.delta?.content) {
          process.stdout.write(chunk.choices[0].delta.content);
        }
      },
      onComplete: (result, metadata) => {
        console.log('\n\nGeneration complete!');
        console.log('Total tokens:', metadata.usage?.totalTokens);
      },
      onError: (error) => {
        console.error('Stream error:', error.message);
      }
    }
  });

  // Alternative: Manual stream processing
  try {
    for await (const chunk of stream) {
      const content = chunk.choices[0]?.delta?.content;
      if (content) {
        process.stdout.write(content);
      }

      if (chunk.choices[0]?.finishReason) {
        console.log('\nFinished:', chunk.choices[0].finishReason);
        break;
      }
    }
  } catch (error) {
    console.error('Streaming error:', error);
  }
}
```

Next.js API Route with Streaming

```
// pages/api/chat/stream.ts
import { NextRequest } from 'next/server';
import { ConfigFactory, AIServiceFactory } from '@/lib/ai';

export default async function handler(req: NextRequest) {
  if (req.method !== 'POST') {
    return new Response('Method not allowed', { status: 405 });
  }

  const { message, sessionId } = await req.json();

  const config = ConfigFactory.abacusai.llm(process.env.ABACUSAI_API_KEY!, {
    defaultModel: 'gpt-4.1-mini'
  });

  const factory = new AIServiceFactory();
  const llmService = await factory.createLLMService(config);

  const stream = await llmService.generateStream({
    messages: [
      { role: 'system', content: 'You are a helpful AI assistant.' },
      { role: 'user', content: message }
    ],
    streamingOptions: {
      bufferSize: 1024,
      timeout: 30000
    }
  });

  const readableStream = new ReadableStream({
    async start(controller) {
      const encoder = new TextEncoder();

      try {
        for await (const chunk of stream) {
          const data = JSON.stringify({
            id: chunk.id,
            content: chunk.choices[0]?.delta?.content || '',
            finishReason: chunk.choices[0]?.finishReason || null,
            usage: chunk.usage
          });

          controller.enqueue(encoder.encode(`data: ${data}\n\n`));

          if (chunk.choices[0]?.finishReason) {
            controller.enqueue(encoder.encode('data: [DONE]\n\n'));
            break;
          }
        }
      } catch (error) {
        console.error('Stream error:', error);
        controller.enqueue(encoder.encode(`data: ${JSON.stringify({ error: 'Stream
failed' })}\n\n`));
      } finally {
        controller.close();
      }
    }
  });

  return new Response(readableStream, {
    headers: {
      'Content-Type': 'text/plain; charset=utf-8',
      'Cache-Control': 'no-cache',
      'Connection': 'keep-alive'
    }
  });
}
```

React Streaming Component

```
'use client';

import { useState, useEffect } from 'react';

interface ChatStreamingProps {
  message: string;
  onComplete?: (fullResponse: string) => void;
}

export function ChatStreaming({ message, onComplete }: ChatStreamingProps) {
  const [response, setResponse] = useState('');
  const [isStreaming, setIsStreaming] = useState(false);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    if (!message) return;

    let fullResponse = '';
    setIsStreaming(true);
    setError(null);
    setResponse('');

    const streamChat = async () => {
      try {
        const response = await fetch('/api/chat/stream', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ message })
        });

        if (!response.ok) {
          throw new Error('Failed to start stream');
        }

        const reader = response.body?.getReader();
        const decoder = new TextDecoder();

        if (!reader) {
          throw new Error('No reader available');
        }

        while (true) {
          const { done, value } = await reader.read();
          if (done) break;

          const chunk = decoder.decode(value);
          const lines = chunk.split('\n');

          for (const line of lines) {
            if (line.startsWith('data: ')) {
              const data = line.slice(6);

              if (data === '[DONE]') {
                setIsStreaming(false);
                onComplete?.(fullResponse);
                return;
              }

              try {
                const parsed = JSON.parse(data);
                if (parsed.content) {
                  fullResponse += parsed.content;
                  setResponse(fullResponse);
                }
                if (parsed.error) {
                  throw new Error(parsed.error);
                }
              } catch (parseError) {
                console.warn('Failed to parse chunk:', data);
              }
            }
          }
        }
      } catch (err) {
        console.error('Streaming error:', err);
        setError(err instanceof Error ? err.message : 'Unknown error');
        setIsStreaming(false);
      }
    };

    streamChat();
  }, [message, onComplete]);

  return (
    <div className="p-4 border rounded-lg">
      {error && (
        <div className="text-red-500 mb-2">Error: {error}</div>
      )}

      <div className="whitespace-pre-wrap">
        {response}
        {isStreaming && (
          <span className="animate-pulse">▋</span>
        )}
      </div>

      {isStreaming && (
        <div className="mt-2 text-sm text-gray-500">
          AI is thinking...
        </div>
      )}
    </div>
  );
}
```

Multi-Provider Examples

Provider Failover

```typescript
import { MultiProviderFactory, ConfigFactory } from './lib/ai';

async function providerFailoverExample() {
  // Create multi-provider configuration
  const openaiConfig = ConfigFactory.openai.llm(process.env.OPENAI_API_KEY!);
  const anthropicConfig = ConfigFactory.anthropic.llm(process.env.ANTHROPIC_API_KEY!);
  const cohereConfig = ConfigFactory.cohere.llm(process.env.COHERE_API_KEY!);

  const factory = new MultiProviderFactory();

  // Create failover service
  const multiService = await factory.createLoadBalancedService(
    [openaiConfig, anthropicConfig, cohereConfig],
    'failover'
  );

  // The service will automatically try providers in order
  const response = await multiService.generate({
    messages: [{ role: 'user', content: 'Explain machine learning' }]
  });

  if (response.success) {
    console.log('Response from:', response.data.provider);
    console.log('Content:', response.data.choices[0].message.content);
  }
}
```

Load Balancing

```
async function loadBalancingExample() {
  const multiConfig = ConfigFactory.createMultiProviderConfig('llm', 'openai', ['an-
thropic', 'cohere'], {
    loadBalancing: {
      strategy: 'weighted',
      weights: {
        openai: 0.5,      // 50% of requests
        anthropic: 0.3,   // 30% of requests
        cohere: 0.2       // 20% of requests
      }
    },
    healthCheck: {
      enabled: true,
      interval: 30000,
      timeout: 5000
    }
  });

  const factory = new MultiProviderFactory();
  const balancedService = await factory.createMultiProviderService(multiConfig);

  // Make multiple requests to see load balancing in action
  for (let i = 0; i < 10; i++) {
    const response = await balancedService.generate({
      messages: [{ role: 'user', content: `Request ${i + 1}` }]
    });

    if (response.success) {
      console.log(`Request ${i + 1} handled by:`, response.data.provider);
    }
  }
}
```

Cost-Optimized Provider Selection

```
class CostOptimizedService {
  private providers: Array<{
    service: LLMService;
    costPerToken: number;
    reliability: number;
  }>;

  constructor() {
    this.providers = [
      {
        service: openaiService,
        costPerToken: 0.00003,
        reliability: 0.99
      },
      {
        service: anthropicService,
        costPerToken: 0.000015,
        reliability: 0.98
      },
      {
        service: cohereService,
        costPerToken: 0.000008,
        reliability: 0.95
      }
    ];
  }

  async generateOptimized(
    request: LLMGenerationRequest,
    budget: number,
    prioritizeReliability = false
  ): Promise<AIResponse<LLMGenerationResponse>> {
    const estimatedTokens = this.estimateTokens(request);

    // Filter providers by budget
    const affordableProviders = this.providers.filter(
      p => p.costPerToken * estimatedTokens <= budget
    );

    if (affordableProviders.length === 0) {
      throw new Error('Budget too low for any provider');
    }

    // Sort by reliability or cost
    affordableProviders.sort((a, b) => {
      return prioritizeReliability
        ? b.reliability - a.reliability
        : a.costPerToken - b.costPerToken;
    });

    // Try providers in order
    for (const provider of affordableProviders) {
      try {
        const response = await provider.service.generate(request);
        if (response.success) {
          return response;
        }
      } catch (error) {
        console.warn(`Provider failed:`, error);
        continue;
      }
    }
    throw new Error('All providers failed');
  }

  private estimateTokens(request: LLMGenerationRequest): number {
    // Simple estimation - in practice, use tiktoken or similar
    const messageLength = request.messages
      .map(m => typeof m.content === 'string' ? m.content : '')
      .join(' ')
      .length;

    return Math.ceil(messageLength / 4) + (request.maxTokens || 1000);
  }
}
```

Chat Service Examples

Basic Chat Session

```
import { ConfigFactory, ChatService } from './lib/ai';

async function basicChatSession() {
  // Create chat service with LLM and embeddings
  const chatConfig = ConfigFactory.createChatConfig('openai', 'voyageai', {
    systemPrompt: 'You are a helpful programming assistant.',
    memoryStrategy: 'sliding_window',
    memorySize: 10,
    enableRAG: false,
    streamingEnabled: true
  });

  const factory = new AIServiceFactory();
  const chatService = await factory.createChatService(chatConfig);

  // Create a new session
  const session = await chatService.createSession(
    'user123',
    'tenant1',
    'Programming Help Session'
  );

  if (!session.success) {
    console.error('Failed to create session:', session.error);
    return;
  }

  console.log('Created session:', session.data.id);

  // First message
  const response1 = await chatService.chat({
    message: 'Can you help me understand async/await in JavaScript?',
    sessionId: session.data.id,
    temperature: 0.7
  });

  if (response1.success) {
    console.log('AI:', response1.data.message.content);
  }

  // Follow-up message (context maintained)
  const response2 = await chatService.chat({
    message: 'Can you show me a practical example?',
    sessionId: session.data.id
  });

  if (response2.success) {
    console.log('AI:', response2.data.message.content);
    console.log('Total tokens used:', response2.data.usage.totalTokens);
  }

  // Get session history
  const messages = await chatService.getMessages(session.data.id);
  if (messages.success) {
    console.log('Conversation history:', messages.data.length, 'messages');
  }
}
```

Chat with Streaming

```
async function streamingChatExample() {
  const chatService = await createChatService();
  const sessionId = 'existing-session-id';

  const stream = await chatService.chatStream({
    message: 'Write a comprehensive guide on React hooks',
    sessionId,
    streamingOptions: {
      onProgress: (status, chunk) => {
        if (chunk?.delta?.content) {
          process.stdout.write(chunk.delta.content);
        }

        // Show retrieval status for RAG
        if (chunk?.context?.stage === 'retrieving') {
          console.log('\n[Searching knowledge base...]');
        } else if (chunk?.context?.stage === 'generating') {
          console.log('\n[Generating response...]');
        }
      },
      onComplete: (result) => {
        console.log('\n\nResponse complete');
        if (result.context?.retrievedDocuments) {
          console.log('Sources used:', result.context.retrievedDocuments.length);
        }
      }
    }
  });

  // Alternative: Manual processing
  let fullResponse = '';
  for await (const chunk of stream) {
    if (chunk.delta.content) {
      fullResponse += chunk.delta.content;
    }

    if (chunk.finishReason) {
      console.log('\nFinal response length:', fullResponse.length);
      break;
    }
  }
}
```

Memory Management

```
async function memoryManagementExample() {
  const chatService = await createChatService();
  const sessionId = 'long-conversation-session';

  // Check memory usage
  const memoryUsage = await chatService.getMemoryUsage(sessionId);
  if (memoryUsage.success) {
    console.log('Memory usage:', memoryUsage.data);

    if (memoryUsage.data.totalTokens > 8000) {
      console.log('Memory getting full, compressing...');

      // Compress session memory
      const compressed = await chatService.compressSessionMemory(sessionId, 'summary');
      if (compressed.success) {
        console.log('Memory compressed successfully');
      }
    }
  }

  // Continue conversation
  const response = await chatService.chat({
    message: 'What did we discuss earlier about machine learning?',
    sessionId,
    memoryOptions: {
      strategy: 'summary',
      maxTokens: 6000
    }
  });

  if (response.success) {
    console.log('AI response:', response.data.message.content);

    // Check if memory was used
    if (response.data.context?.memoryUsed) {
      console.log('Used', response.data.context.memoryUsed.length, 'previous messages')
;
    }
  }
}
```

RAG Implementation

Basic RAG Setup

```
import { Document } from './lib/ai/interfaces';

async function basicRAGExample() {
  const chatService = await createChatService(); // With embeddings enabled
  const sessionId = await createSession();

  // Add documents to knowledge base
  const documents: Document[] = [
    {
      id: '1',
      content: 'React hooks are functions that let you use state and other React fea-
tures in functional components. The most common hooks are useState and useEffect.',
      metadata: { source: 'react-docs', category: 'hooks' }
    },
    {
      id: '2',
      content: 'useEffect is used for side effects like data fetching, subscriptions,
or manually changing the DOM. It runs after the render is committed to the screen.',
      metadata: { source: 'react-docs', category: 'hooks' }
    },
    {
      id: '3',
      content: 'useState returns a stateful value and a function to update it. The
state is preserved between re-renders.',
      metadata: { source: 'react-docs', category: 'hooks' }
    }
  ];

  // Add to knowledge base
  const added = await chatService.addToKnowledgeBase(
    'react-knowledge',
    documents,
    'tenant1'
  );

  if (added.success) {
    console.log('Added documents:', added.data.length);
  }

  // Enable RAG for session
  await chatService.enableRAG(sessionId, {
    enabled: true,
    knowledgeBase: 'react-knowledge',
    maxResults: 3,
    threshold: 0.7,
    rerank: true,
    includeContext: true
  });

  // Ask question with RAG
  const response = await chatService.chat({
    message: 'How do I use useState in React?',
    sessionId,
    ragOptions: {
      enabled: true,
      maxResults: 2,
      threshold: 0.8
    }
  });

  if (response.success) {
    console.log('AI Response:', response.data.message.content);
    // Show retrieved sources
    if (response.data.context?.retrievedDocuments) {
      console.log('\nSources:');
      response.data.context.retrievedDocuments.forEach((doc, index) => {
        console.log(`${index + 1}. Score: ${doc.score}, Content: ${doc.document.con-
tent.substring(0, 100)}...`);
      });
    }
  }
}
```

Advanced RAG with Custom Retrieval

```
class AdvancedRAGService {
  private chatService: ChatService;
  private embeddingsService: EmbeddingsService;

  constructor(chatService: ChatService, embeddingsService: EmbeddingsService) {
    this.chatService = chatService;
    this.embeddingsService = embeddingsService;
  }

  async hybridSearch(
    query: string,
    knowledgeBase: string,
    options: {
      semanticWeight: number;
      keywordWeight: number;
      maxResults: number;
      rerank: boolean;
    }
  ): Promise<Document[]> {
    // 1. Semantic search using embeddings
    const semanticResults = await this.chatService.searchKnowledgeBase(
      query,
      knowledgeBase,
      undefined,
      {
        maxResults: options.maxResults * 2,
        threshold: 0.6
      }
    );

    // 2. Keyword search (simple TF-IDF simulation)
    const keywordResults = await this.keywordSearch(query, knowledgeBase);

    // 3. Combine and rerank results
    const combinedResults = this.combineResults(
      semanticResults.success ? semanticResults.data : [],
      keywordResults,
      options.semanticWeight,
      options.keywordWeight
    );

    // 4. Rerank if requested
    if (options.rerank) {
      return this.rerankResults(query, combinedResults.slice(0, options.maxResults));
    }

    return combinedResults.slice(0, options.maxResults);
  }

  async chatWithAdvancedRAG(
    message: string,
    sessionId: string,
    ragOptions: {
      knowledgeBase: string;
      semanticWeight: number;
      keywordWeight: number;
      maxResults: number;
      contextTemplate?: string;
    }
  ): Promise<string> {
    // Retrieve relevant documents
    const relevantDocs = await this.hybridSearch(message, ragOptions.knowledgeBase, {
      semanticWeight: ragOptions.semanticWeight,
      keywordWeight: ragOptions.keywordWeight,
      maxResults: ragOptions.maxResults,
      rerank: true
    });

    // Create context from retrieved documents
    const context = this.createContext(relevantDocs, ragOptions.contextTemplate);

    // Enhanced prompt with context
    const enhancedMessage = `Context information:
${context}

User question: ${message}

Please answer the question using the provided context. If the context doesn't contain
relevant information, say so.`;

    // Chat with context
    const response = await this.chatService.chat({
      message: enhancedMessage,
      sessionId,
      temperature: 0.3 // Lower temperature for factual responses
    });

    return response.success ? response.data.message.content! : 'Sorry, I couldn\'t gen-
erate a response.';
  }

  private async keywordSearch(query: string, knowledgeBase: string):
Promise<Document[]> {
    // Implement keyword search logic
    // This is a simplified version - in practice, use full-text search
    const words = query.toLowerCase().split(' ');
    // Return documents that contain query keywords
    return []; // Placeholder
  }

  private combineResults(
    semanticResults: any[],
    keywordResults: Document[],
    semanticWeight: number,
    keywordWeight: number
  ): Document[] {
    // Combine and score results
    const combined = new Map<string, { doc: Document; score: number }>();

    // Add semantic results
    semanticResults.forEach(result => {
      const id = result.document.id;
      combined.set(id, {
        doc: result.document,
        score: result.score * semanticWeight
      });
    });

    // Add keyword results
    keywordResults.forEach(doc => {
      const id = doc.id;
      const existing = combined.get(id);
      if (existing) {
        existing.score += keywordWeight;
      } else {
        combined.set(id, {
          doc,
          score: keywordWeight
        });
      }
    });

    // Sort by combined score
    return Array.from(combined.values())
      .sort((a, b) => b.score - a.score)
      .map(item => item.doc);
  }

  private async rerankResults(query: string, documents: Document[]):
Promise<Document[]> {
    // Use cross-encoder for reranking (placeholder)
    // In practice, use a reranking model like Cohere's rerank API
    return documents; // Placeholder
  }

  private createContext(documents: Document[], template?: string): string {
    if (template) {
      return template.replace('{documents}', documents.map(doc => doc.content).join('\n
\n'));
    }

    return documents
      .map((doc, index) => `Document ${index + 1}:\n${doc.content}`)
      .join('\n\n---\n\n');
  }
}
```

Document Processing Pipeline

```
import { ChunkingOptions } from './lib/ai/interfaces';

class DocumentProcessor {
  private embeddingsService: EmbeddingsService;
  private chatService: ChatService;

  constructor(embeddingsService: EmbeddingsService, chatService: ChatService) {
    this.embeddingsService = embeddingsService;
    this.chatService = chatService;
  }

  async processDocument(
    content: string,
    metadata: Record<string, any>,
    knowledgeBase: string
  ): Promise<void> {
    // 1. Preprocess text
    const cleanedContent = this.preprocessText(content);

    // 2. Chunk the document
    const chunks = await this.embeddingsService.chunkText(cleanedContent, {
      strategy: 'semantic',
      maxChunkSize: 1000,
      overlapSize: 100,
      preserveSentences: true
    });

    if (!chunks.success) {
      throw new Error('Failed to chunk document');
    }

    // 3. Create documents from chunks
    const documents: Document[] = chunks.data.map((chunk, index) => ({
      id: `${metadata.id}_chunk_${index}`,
      content: chunk.content,
      metadata: {
        ...metadata,
        chunkIndex: index,
        startPosition: chunk.startPosition,
        endPosition: chunk.endPosition,
        parentDocumentId: metadata.id
      },
      parentDocumentId: metadata.id
    }));

    // 4. Generate embeddings and add to knowledge base
    const embeddedDocs = await this.embeddingsService.embedDocuments(documents);

    if (embeddedDocs.success) {
      await this.chatService.addToKnowledgeBase(
        knowledgeBase,
        embeddedDocs.data,
        metadata.tenantId
      );

      console.log(`Processed document ${metadata.id} into ${documents.length} chunks`);
    }
  }

  async processPDFFile(
    fileBuffer: Buffer,
    metadata: Record<string, any>,
    knowledgeBase: string
  ): Promise<void> {
    // Extract text from PDF using LLM API
    const base64Content = fileBuffer.toString('base64');

    const extractionResponse = await this.llmService.generate({
      messages: [{
        role: 'user',
        content: [{
          type: 'file',
          file: {
            filename: metadata.filename || 'document.pdf',
            file_data: `data:application/pdf;base64,${base64Content}`
          }
        }, {
          type: 'text',
          text:
'Extract all text content from this PDF file. Return only the extracted text without
any formatting or commentary.'
        }]
      }]
    });

    if (extractionResponse.success) {
      const extractedText = extractionResponse.data.choices[0].message.content!;
      await this.processDocument(extractedText, metadata, knowledgeBase);
    } else {
      throw new Error('Failed to extract text from PDF');
    }
  }

  private preprocessText(content: string): string {
    return content
      // Remove excessive whitespace
      .replace(/\s+/g, ' ')
      // Remove empty lines
      .replace(/\n\s*\n/g, '\n')
      // Trim
      .trim();
  }

  async batchProcessDocuments(
    documents: Array<{ content: string; metadata: Record<string, any> }>,
    knowledgeBase: string,
    batchSize = 10
  ): Promise<void> {
    for (let i = 0; i < documents.length; i += batchSize) {
      const batch = documents.slice(i, i + batchSize);
      const promises = batch.map(doc =>
        this.processDocument(doc.content, doc.metadata, knowledgeBase)
      );

      await Promise.allSettled(promises);
      console.log(`Processed batch ${Math.floor(i / batchSize) + 1}/${Math.ceil(documen
ts.length / batchSize)}`);

      // Small delay to avoid rate limits
      await new Promise(resolve => setTimeout(resolve, 1000));
    }
  }
}
```

Error Handling Examples

Comprehensive Error Handling

```
import { RetryStrategy, ErrorRecoveryManager, CircuitBreaker } from './lib/ai/error-
handling';

async function robustAIService() {
  const config = ConfigFactory.openai.llm(process.env.OPENAI_API_KEY!);
  const llmService = await factory.createLLMService(config);

  // Set up retry strategy
  const retryStrategy = new RetryStrategy({
    maxAttempts: 3,
    baseDelay: 2000,
    backoffMultiplier: 2,
    jitter: true,
    onRetry: (attempt, error) => {
      console.log(`Retry attempt ${attempt} for error: ${error.details.type}`);
    }
  });

  // Set up circuit breaker
  const circuitBreaker = new CircuitBreaker('openai', 'llm', {
    failureThreshold: 5,
    resetTimeout: 60000
  });

  // Set up error recovery
  const recoveryManager = new ErrorRecoveryManager();

  const request: LLMGenerationRequest = {
    messages: [{ role: 'user', content: 'Generate a poem about AI' }]
  };

  try {
    // Execute with all error handling layers
    const result = await circuitBreaker.execute(async () => {
      return await retryStrategy.execute(async () => {
        const response = await llmService.generate(request);
        if (!response.success) {
          throw response.error;
        }
        return response.data;
      });
    });

    console.log('Success:', result.choices[0].message.content);

  } catch (error) {
    console.log('Primary service failed, attempting recovery...');

    // Attempt error recovery
    const recoveryResult = await recoveryManager.recover(
      error,
      () => llmService.generate(request),
      {
        service: llmService,
        provider: 'openai',
        serviceType: 'llm',
        fallbackServices: [/* other services */]
      }
    );

    if (recoveryResult.success) {
      console.log('Recovered successfully using:', recoveryResult.strategy);
```

```
        console.log('Result:', recoveryResult.data);
      } else {
        console.error('All recovery attempts failed:', recoveryResult.error);

        // Final fallback - provide graceful degradation
        return {
          success: false,
          message: 'I apologize, but I\'m currently experiencing technical difficulties.
Please try again later.'
        };
      }
    }
  }
```

Error Analytics Dashboard

```
import { globalErrorTracker, ErrorAnalyzer } from './lib/ai/error-handling';

class ErrorDashboard {
  async generateReport(timeRangeHours = 24) {
    const stats = globalErrorTracker.getStatistics(timeRangeHours);

    return {
      overview: {
        totalErrors: stats.totalErrors,
        errorRate: stats.errorRate,
        meanTimeBetweenErrors: stats.meanTimeBetweenErrors
      },
      breakdown: {
        byType: Object.entries(stats.errorsByType)
          .sort(([,a], [,b]) => b - a)
          .map(([type, count]) => ({ type, count, percentage: (count / stats.totalError
s) * 100 })),

        byProvider: Object.entries(stats.errorsByProvider)
          .sort(([,a], [,b]) => b - a)
          .map(([provider, count]) => ({ provider, count, percentage: (count / stats.to
talErrors) * 100 })),

        byService: Object.entries(stats.errorsByService)
          .sort(([,a], [,b]) => b - a)
          .map(([service, count]) => ({ service, count, percentage: (count / stats.tota
lErrors) * 100 }))
      },
      recentErrors: stats.recentErrors.map(error => ({
        ...ErrorAnalyzer.analyzeError(error),
        timestamp: error.details.timestamp
      })),
      recommendations: this.generateRecommendations(stats)
    };
  }

  private generateRecommendations(stats: any): string[] {
    const recommendations: string[] = [];

    // High error rate
    if (stats.errorRate > 10) {
      recommendations.push('High error rate detected. Consider implementing circuit
breakers or reducing request frequency.');
    }

    // Common error types
    const topErrorType = Object.entries(stats.errorsByType)[0];
    if (topErrorType && topErrorType[1] > stats.totalErrors * 0.4) {
      recommendations.push(`${topErrorType[0]} errors are prevalent. Review ${topError-
Type[0]} handling strategies.`);
    }

    // Provider issues
    const topErrorProvider = Object.entries(stats.errorsByProvider)[0];
    if (topErrorProvider && topErrorProvider[1] > stats.totalErrors * 0.6) {
      recommendations.push(`Consider diversifying providers. $
{topErrorProvider[0]} has high error count.`);
    }

    return recommendations;
  }
}
```

```javascript
// Usage
const dashboard = new ErrorDashboard();
const report = await dashboard.generateReport(24);
console.log(JSON.stringify(report, null, 2));
```

Integration Examples

Express.js Integration

```
import express from 'express';
import { ConfigFactory, AIServiceFactory } from './lib/ai';

const app = express();
app.use(express.json());

// Initialize AI services
let llmService: LLMService;
let chatService: ChatService;

async function initializeAI() {
  const llmConfig = ConfigFactory.openai.llm(process.env.OPENAI_API_KEY!);
  const chatConfig = ConfigFactory.createChatConfig('openai', 'voyageai');

  const factory = new AIServiceFactory();
  llmService = await factory.createLLMService(llmConfig);
  chatService = await factory.createChatService(chatConfig);
}

// Text generation endpoint
app.post('/api/generate', async (req, res) => {
  try {
    const { prompt, options = {} } = req.body;

    const response = await llmService.generate({
      messages: [{ role: 'user', content: prompt }],
      ...options
    });

    if (response.success) {
      res.json({
        success: true,
        text: response.data.choices[0].message.content,
        usage: response.data.usage
      });
    } else {
      res.status(400).json({
        success: false,
        error: response.error.message
      });
    }
  } catch (error) {
    res.status(500).json({
      success: false,
      error: 'Internal server error'
    });
  }
});

// Streaming endpoint
app.post('/api/stream', async (req, res) => {
  try {
    const { prompt, options = {} } = req.body;

    res.writeHead(200, {
      'Content-Type': 'text/event-stream',
      'Cache-Control': 'no-cache',
      'Connection': 'keep-alive'
    });

    const stream = await llmService.generateStream({
      messages: [{ role: 'user', content: prompt }],
      ...options
    });

    for await (const chunk of stream) {
      const data = JSON.stringify({
        content: chunk.choices[0]?.delta?.content || '',
        finishReason: chunk.choices[0]?.finishReason
      });

      res.write(`data: ${data}\n\n`);

      if (chunk.choices[0]?.finishReason) {
        res.write('data: [DONE]\n\n');
        break;
      }
    }

    res.end();
  } catch (error) {
    res.write(`data: ${JSON.stringify({ error: 'Stream failed' })}\n\n`);
    res.end();
  }
});

// Chat endpoint
app.post('/api/chat', async (req, res) => {
  try {
    const { message, sessionId, userId } = req.body;

    // Create session if not provided
    let session = sessionId;
    if (!session) {
      const newSession = await chatService.createSession(userId);
      if (newSession.success) {
        session = newSession.data.id;
      }
    }

    const response = await chatService.chat({
      message,
      sessionId: session,
      userId
    });

    if (response.success) {
      res.json({
        success: true,
        message: response.data.message.content,
        sessionId: session,
        usage: response.data.usage
      });
    } else {
      res.status(400).json({
        success: false,
        error: response.error.message
      });
    }
  } catch (error) {
    res.status(500).json({
      success: false,
      error: 'Internal server error'
    });
  }
```

```
});

// Initialize and start server
initializeAI().then(() => {
  app.listen(3000, () => {
    console.log('AI API server running on port 3000');
  });
});
```

React Hook Integration

```
// hooks/useAIGeneration.ts
import { useState, useCallback } from 'react';

interface UseAIGenerationOptions {
  streaming?: boolean;
  onChunk?: (chunk: string) => void;
  onComplete?: (fullResponse: string) => void;
  onError?: (error: string) => void;
}

export function useAIGeneration(options: UseAIGenerationOptions = {}) {
  const [isGenerating, setIsGenerating] = useState(false);
  const [response, setResponse] = useState('');
  const [error, setError] = useState<string | null>(null);

  const generate = useCallback(async (prompt: string, generateOptions: any = {}) => {
    setIsGenerating(true);
    setError(null);
    setResponse('');

    try {
      if (options.streaming) {
        // Streaming request
        const response = await fetch('/api/stream', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ prompt, options: generateOptions })
        });

        const reader = response.body?.getReader();
        const decoder = new TextDecoder();
        let fullResponse = '';

        if (reader) {
          while (true) {
            const { done, value } = await reader.read();
            if (done) break;

            const chunk = decoder.decode(value);
            const lines = chunk.split('\n');

            for (const line of lines) {
              if (line.startsWith('data: ')) {
                const data = line.slice(6);
                if (data === '[DONE]') {
                  options.onComplete?.(fullResponse);
                  return;
                }

                try {
                  const parsed = JSON.parse(data);
                  if (parsed.content) {
                    fullResponse += parsed.content;
                    setResponse(fullResponse);
                    options.onChunk?.(parsed.content);
                  }
                } catch (e) {
                  // Skip invalid JSON
                }
              }
            }
          }
        }
      } else {
        // Non-streaming request
        const response = await fetch('/api/generate', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ prompt, options: generateOptions })
        });

        const result = await response.json();

        if (result.success) {
          setResponse(result.text);
          options.onComplete?.(result.text);
        } else {
          throw new Error(result.error);
        }
      }
    } catch (err) {
      const errorMessage = err instanceof Error ? err.message : 'Unknown error';
      setError(errorMessage);
      options.onError?.(errorMessage);
    } finally {
      setIsGenerating(false);
    }
  }, [options]);

  const clear = useCallback(() => {
    setResponse('');
    setError(null);
  }, []);

  return {
    generate,
    clear,
    isGenerating,
    response,
    error
  };
}

// Usage in component
function AIGeneratorComponent() {
  const { generate, isGenerating, response, error } = useAIGeneration({
    streaming: true,
    onChunk: (chunk) => {
      // Handle individual chunks if needed
    },
    onComplete: (fullResponse) => {
      console.log('Generation complete:', fullResponse);
    }
  });

  const handleGenerate = () => {
    generate('Write a short story about AI', {
      temperature: 0.8,
      maxTokens: 500
    });
  };

  return (
    <div>
      <button onClick={handleGenerate} disabled={isGenerating}>
```

```
        {isGenerating ? 'Generating...' : 'Generate Story'}
      </button>

      {error && <div className="error">Error: {error}</div>}

      <div className="response">
        {response}
        {isGenerating && <span className="cursor">▌</span>}
      </div>
    </div>
  );
}
```

These examples demonstrate practical implementations of the AI Abstraction Layer in various scenarios, from basic usage to complex multi-provider setups with streaming, RAG, and error handling.