Error Handling Guide

Comprehensive guide to error handling patterns, retry strategies, and recovery mechanisms in the AI Abstraction Layer.

Table of Contents

- Error Types and Categories
- Error Handling Patterns
- Retry Strategies
- Circuit Breaker Pattern
- Error Recovery
- Provider-Specific Error Mapping
- Best Practices

Error Types and Categories

AIError Structure

All errors in the abstraction layer use the standardized `AIError` class:

```
class AIError extends Error {
  public readonly details: AIErrorDetails;
  public readonly isAIError = true;

  constructor(details: AIErrorDetails);

  // Helper methods
  isRetryable(): boolean;
  isRateLimit(): boolean;
  isQuotaExceeded(): boolean;
  isNetworkError(): boolean;
  isCritical(): boolean;
  getRetryDelay(): number;
}
```

Error Categories

Category	Description	Retryable	Examples
`authentication`	API key or auth issues	❌	Invalid API key, expired token
`authorization`	Permission/access issues	❌	Insufficient permissions, blocked access
`rate_limit`	Rate limiting by provider	✅	Too many requests per minute
`quota_exceeded`	Usage limits exceeded	❌	Monthly quota reached, token limit
`invalid_request`	Malformed requests	❌	Invalid parameters, missing fields
`model_not_found`	Requested model unavailable	❌	Model doesn't exist or not accessible
`service_unavailable`	Provider service issues	✅	Server errors, maintenance
`timeout`	Request timeout	✅	Network timeout, slow response
`network`	Network connectivity	✅	Connection refused, DNS issues
`parsing`	Response parsing errors	✅	Invalid JSON, unexpected format
`validation`	Input validation failures	❌	Invalid model parameters
`streaming`	Streaming-specific errors	✅	Stream interruption, connection lost
`provider_error`	Provider-specific issues	✅	Internal provider errors
`unknown`	Unclassified errors	✅	Unexpected errors

Error Handling Patterns

Basic Error Handling

```
import { AIError, isAIError } from './lib/ai';

try {
  const response = await llmService.generate(request);
  console.log('Success:', response.data);
} catch (error) {
  if (isAIError(error)) {
    console.error('AI Error:', error.details.type, error.message);

    // Handle based on error type
    switch (error.details.type) {
      case 'authentication':
        // Handle auth error - maybe refresh token
        break;
      case 'rate_limit':
        // Handle rate limiting - wait and retry
        const retryAfter = error.getRetryDelay();
        setTimeout(() => retryRequest(), retryAfter);
        break;
      case 'quota_exceeded':
        // Handle quota - notify user, upgrade plan
        notifyQuotaExceeded();
        break;
      default:
        // Generic error handling
        logError(error);
    }
  } else {
    // Handle non-AI errors
    console.error('Unexpected error:', error);
  }
}
```

Response-Based Error Handling

```
// Using AIResponse pattern (recommended)
const response = await llmService.generate(request);

if (response.success) {
  // Handle successful response
  console.log('Generated text:', response.data.choices[0].message.content);
} else {
  // Handle error from response
  const error = response.error;

  switch (error.severity) {
    case 'critical':
      alertOpsTeam(error);
      break;
    case 'high':
      logError(error);
      break;
    case 'medium':
      // Retry logic
      if (error.retryable) {
        scheduleRetry(request, error.retryAfter);
      }
      break;
    default:
      // Low severity - just log
      console.warn('Minor error:', error.message);
  }
}
```

Retry Strategies

Built-in Retry Strategies

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

// Pre-configured strategies
const conservative = RetryStrategies.conservative;  // 2 attempts, 1s base delay
const aggressive = RetryStrategies.aggressive;      // 5 attempts, 0.5s base delay
const rateLimitFocused = RetryStrategies.rateLimitOptimized; // 3 attempts, longer
delays
const networkFocused = RetryStrategies.networkOptimized;     // 4 attempts, network-fo-
cused
```

Custom Retry Configuration

```
const customRetryStrategy = new RetryStrategy({
  maxAttempts: 4,
  baseDelay: 2000,
  maxDelay: 30000,
  backoffMultiplier: 2,
  jitter: true,
  retryableErrors: ['rate_limit', 'service_unavailable', 'timeout'],
  onRetry: (attempt, error) => {
    console.log(`Retry attempt ${attempt} for ${error.details.type}`);
  },
  shouldRetry: (error, attempt) => {
    // Custom retry logic
    if (error.details.type === 'quota_exceeded') return false;
    if (attempt >= 3 && error.details.severity === 'low') return false;
    return error.isRetryable();
  }
});

// Use retry strategy
const result = await customRetryStrategy.execute(
  () => llmService.generate(request),
  { provider: 'openai', service: 'llm' }
);
```

Provider-Specific Retry Strategies

```
import { ProviderRetryStrategies, getRetryStrategy } from './lib/ai/error-handling';

// Get provider-specific strategy
const openaiStrategy = ProviderRetryStrategies.openai;
const anthropicStrategy = ProviderRetryStrategies.anthropic;

// Or get automatically based on provider
const strategy = getRetryStrategy('abacusai');

// Use with service
const result = await strategy.execute(() => service.generate(request));
```

Decorator Pattern for Automatic Retry

```
import { withRetry } from './lib/ai/error-handling';

class MyLLMService implements LLMService {
  @withRetry(RetryStrategies.aggressive)
  async generate(request: LLMGenerationRequest): Promise<AIResponse<LLMGenerationRe-
sponse>> {
    // Implementation with automatic retry
    return this.makeAPICall(request);
  }

  @withRetry(ProviderRetryStrategies.openai)
  async generateStream(request: LLMGenerationRequest): Promise<AIStreamResponse<LLMStre
amChunk>> {
    // Streaming with retry
    return this.makeStreamingCall(request);
  }
}
```

Circuit Breaker Pattern

Basic Circuit Breaker Usage

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

// Get circuit breaker for a service
const circuitBreaker = globalCircuitBreakerManager.getCircuitBreaker('openai', 'llm', {
  failureThreshold: 5,        // Open after 5 failures
  resetTimeout: 60000,        // Try to close after 1 minute
  monitoringWindow: 300000,   // 5-minute monitoring window
  minimumRequests: 10,        // Need 10 requests before considering circuit state
  halfOpenMaxAttempts: 3      // Allow 3 attempts in half-open state
});

// Use circuit breaker
try {
  const result = await circuitBreaker.execute(() => llmService.generate(request));
  console.log('Success:', result);
} catch (error) {
  if (error.details?.type === 'service_unavailable' && error.message.includes('circuit
breaker')) {
    console.log('Service is currently unavailable due to circuit breaker');
    // Handle circuit open scenario
    provideFallbackResponse();
  }
}
```

Circuit Breaker Monitoring

```
// Check circuit breaker status
const metrics = circuitBreaker.getMetrics();
console.log('Circuit state:', metrics.state);
console.log('Failure rate:', metrics.failureRate);
console.log('Time until retry:', metrics.timeUntilRetry);

// Get all circuit breaker states
const allStates = globalCircuitBreakerManager.getAllStates();
console.log('All circuit breakers:', allStates);

// Reset circuit breaker if needed
if (metrics.state === 'open') {
  circuitBreaker.forceClose(); // Use with caution
}
```

Automatic Circuit Breaker with Decorators

```
import { withCircuitBreaker } from './lib/ai/error-handling';

class ReliableLLMService {
  @withCircuitBreaker({
    failureThreshold: 3,
    resetTimeout: 30000
  })
  async generate(request: LLMGenerationRequest) {
    return this.llmService.generate(request);
  }
}
```

Error Recovery

Recovery Strategies

The abstraction layer provides multiple recovery strategies:

1. **Fallback Provider**: Switch to alternative provider
2. **Model Fallback**: Use simpler/cheaper model
3. **Request Simplification**: Reduce request complexity
4. **Cache Recovery**: Use cached response
5. **Degraded Mode**: Provide minimal response

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

// Use global recovery manager
const recoveryResult = await globalRecoveryManager.recover(
  error,
  () => llmService.generate(request),
  {
    service: llmService,
    provider: 'openai',
    serviceType: 'llm',
    fallbackServices: [anthropicService, cohereService],
    metadata: { originalModel: 'gpt-4' }
  }
);

if (recoveryResult.success) {
  console.log('Recovered successfully:', recoveryResult.data);
  console.log('Recovery strategy used:', recoveryResult.strategy);
} else {
  console.error('All recovery strategies failed:', recoveryResult.error);
}
```

Custom Recovery Strategies

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

class CustomFallbackStrategy implements RecoveryStrategy {
  name = 'custom_fallback';
  priority = 85;

  canRecover(error: AIError): boolean {
    return error.details.type === 'quota_exceeded' &&
           error.details.provider === 'openai';
  }

  async recover<T>(
    error: AIError,
    originalOperation: () => Promise<T>,
    context: RecoveryContext
  ): Promise<T> {
    // Switch to Anthropic when OpenAI quota exceeded
    const anthropicConfig = getAnthropicConfig();
    const anthropicService = await createAnthropicService(anthropicConfig);

    // Modify request for Anthropic
    const modifiedRequest = adaptRequestForAnthropic(originalOperation);
    return modifiedRequest();
  }
}

// Register custom strategy
const recoveryManager = new ErrorRecoveryManager();
recoveryManager.registerStrategy(new CustomFallbackStrategy());
```

Provider-Specific Error Mapping

Automatic Error Mapping

```
import { mapProviderError, globalErrorMapper } from './lib/ai/error-handling';

// Automatically map provider errors
try {
  const response = await fetch('https://api.openai.com/v1/chat/completions', {
    // ... request config
  });

  if (!response.ok) {
    throw await response.json();
  }
} catch (providerError) {
  // Map to standardized AIError
  const aiError = mapProviderError(providerError, 'openai', 'llm', 'gpt-4');

  // Now you have a standardized error
  console.log('Error type:', aiError.details.type);
  console.log('Retryable:', aiError.isRetryable());
  console.log('Retry after:', aiError.getRetryDelay());
}
```

Custom Error Mappings

```
import { registerErrorMappings, ErrorMapping } from './lib/ai/error-handling';

const customMappings: ErrorMapping[] = [
  {
    condition: (error) => error.code === 'CUSTOM_RATE_LIMIT',
    mapTo: 'rate_limit',
    severity: 'medium',
    retryable: true,
    extractRetryAfter: (error) => error.retryAfterSeconds
  },
  {
    condition: (error) => error.message?.includes('model overloaded'),
    mapTo: 'service_unavailable',
    severity: 'high',
    retryable: true
  }
];

// Register for custom provider
registerErrorMappings('custom', customMappings);
```

Error Analytics and Monitoring

Error Statistics

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

// Get error statistics
const stats = globalErrorTracker.getStatistics(24); // Last 24 hours

console.log('Total errors:', stats.totalErrors);
console.log('Error rate per hour:', stats.errorRate);
console.log('Errors by type:', stats.errorsByType);
console.log('Errors by provider:', stats.errorsByProvider);
console.log('Mean time between errors:', stats.meanTimeBetweenErrors);

// Analyze specific error
const analysis = ErrorAnalyzer.analyzeError(error);
console.log('Error analysis:', analysis);
console.log('Suggested action:', analysis.suggestedAction);
console.log('User-friendly message:', analysis.userMessage);
```

Error Reporting

```
import { ErrorReporting } from './lib/ai/error-handling';

// Check if error should be reported
if (ErrorReporting.shouldReportError(error)) {
  const report = ErrorReporting.createErrorReport(error, {
    userId: 'user123',
    sessionId: 'session456',
    operation: 'text_generation'
  });

  // Send to monitoring service
  await sendToMonitoring(report);
}
```

Streaming Error Handling

Stream Error Recovery

```
async function handleStreamingWithRecovery(request: LLMGenerationRequest) {
  let retryCount = 0;
  const maxRetries = 3;

  while (retryCount < maxRetries) {
    try {
      const stream = await llmService.generateStream({
        ...request,
        streamingOptions: {
          onError: (error) => {
            console.error('Stream error:', error);

            if (error.details.type === 'streaming' && retryCount < maxRetries) {
              // Attempt stream recovery
              setTimeout(() => {
                retryCount++;
                handleStreamingWithRecovery(request);
              }, 2000);
            }
          },
          onComplete: (result) => {
            console.log('Stream completed successfully');
          }
        }
      });

      // Process stream
      for await (const chunk of stream) {
        processChunk(chunk);
      }

      return; // Success - exit retry loop

    } catch (error) {
      retryCount++;

      if (retryCount >= maxRetries) {
        throw error;
      }

      // Wait before retry
      await new Promise(resolve => setTimeout(resolve, 1000 * retryCount));
    }
  }
}
```

Stream Timeout Handling

```
import { withTimeout } from './lib/ai/utils';

async function streamWithTimeout(request: LLMGenerationRequest, timeoutMs: number = 300
00) {
  try {
    const stream = await withTimeout(
      llmService.generateStream(request),
      timeoutMs
    );

    const chunks: LLMStreamChunk[] = [];
    const startTime = Date.now();

    for await (const chunk of stream) {
      chunks.push(chunk);

      // Check for individual chunk timeout
      if (Date.now() - startTime > timeoutMs) {
        throw new AIError({
          type: 'timeout',
          message: 'Stream processing timeout',
          provider: 'openai',
          service: 'llm',
          severity: 'medium',
          retryable: true,
          timestamp: new Date().toISOString()
        });
      }
    }

    return chunks;
  } catch (error) {
    if (error.name === 'TimeoutError') {
      throw mapProviderError(error, 'openai', 'llm');
    }
    throw error;
  }
}
```

Best Practices

1. Error Categorization

```
// Always categorize errors appropriately
function handleError(error: AIError) {
  const priority = ErrorUtils.getErrorPriority(error);

  switch (priority) {
    case 'critical':
      // Immediate attention required
      alertOpsTeam(error);
      logError(error, 'error');
      break;
    case 'high':
      // Important but not critical
      logError(error, 'warn');
      scheduleInvestigation(error);
      break;
    case 'medium':
      // Can be handled automatically
      if (error.isRetryable()) {
        scheduleRetry(error);
      }
      logError(error, 'info');
      break;
    case 'low':
      // Just log for monitoring
      logError(error, 'debug');
      break;
  }
}
```

2. Graceful Degradation

```
async function generateWithFallback(request: LLMGenerationRequest): Promise<string> {
  try {
    const response = await primaryLLMService.generate(request);
    return response.data!.choices[0].message.content!;
  } catch (error) {
    console.warn('Primary service failed, trying fallback:', error.message);

    try {
      // Try simpler model
      const fallbackResponse = await primaryLLMService.generate({
        ...request,
        model: 'gpt-3.5-turbo', // Simpler model
        maxTokens: Math.min(request.maxTokens || 1000, 500) // Reduce tokens
      });
      return fallbackResponse.data!.choices[0].message.content!;
    } catch (fallbackError) {
      console.warn('Fallback failed, using secondary service:', fallbackError.message);

      try {
        const secondaryResponse = await secondaryLLMService.generate(request);
        return secondaryResponse.data!.choices[0].message.content!;
      } catch (secondaryError) {
        // Last resort - return error message
        return 'I apologize, but I\'m currently experiencing technical difficulties.
Please try again later.';
      }
    }
  }
}
```

3. Error Context Preservation

```
function enrichError(error: AIError, context: Record<string, any>): AIError {
  return new AIError({
    ...error.details,
    context: {
      ...error.details.context,
      ...context,
      stackTrace: error.stack,
      timestamp: new Date().toISOString()
    }
  });
}

// Usage
try {
  await llmService.generate(request);
} catch (error) {
  const enrichedError = enrichError(error, {
    userId: currentUser.id,
    requestId: request.requestId,
    model: request.model,
    tokenCount: request.maxTokens
  });

  throw enrichedError;
}
```

4. Rate Limit Respect

```
class RateLimitAwareLLMService {
  private rateLimitInfo: RateLimitInfo | null = null;

  async generate(request: LLMGenerationRequest): Promise<AIResponse<LLMGenerationRe-
sponse>> {
    // Check rate limit before making request
    if (this.rateLimitInfo && this.isRateLimited()) {
      const waitTime = this.getWaitTime();
      console.log(`Rate limited. Waiting ${waitTime}ms`);
      await new Promise(resolve => setTimeout(resolve, waitTime));
    }

    try {
      const response = await this.makeRequest(request);

      // Update rate limit info from response headers
      this.updateRateLimitInfo(response);

      return response;
    } catch (error) {
      if (isAIError(error) && error.isRateLimit()) {
        this.rateLimitInfo = {
          requestsRemaining: 0,
          tokensRemaining: 0,
          resetTime: new Date(Date.now() + error.getRetryDelay()).toISOString()
        };
      }
      throw error;
    }
  }

  private isRateLimited(): boolean {
    if (!this.rateLimitInfo) return false;

    const resetTime = new Date(this.rateLimitInfo.resetTime);
    return Date.now() < resetTime.getTime() &&
           (this.rateLimitInfo.requestsRemaining <= 0 || this.rateLimit-
Info.tokensRemaining <= 0);
  }
}
```

5. Error Monitoring Integration

```
// Integrate with monitoring services
class MonitoringAwareErrorHandler {
  private readonly metricsCollector: MetricsCollector;
  private readonly alertManager: AlertManager;

  async handleError(error: AIError, context: any) {
    // Collect metrics
    this.metricsCollector.increment('ai.errors.total', {
      provider: error.details.provider,
      service: error.details.service,
      type: error.details.type,
      severity: error.details.severity
    });

    // Send alerts for critical errors
    if (error.isCritical()) {
      await this.alertManager.sendAlert({
        level: 'critical',
        message: error.message,
        context,
        error: error.details
      });
    }

    // Track error patterns
    const errorPattern = `${error.details.provider}:${error.details.type}`;
    const recentErrorCount = this.getRecentErrorCount(errorPattern, 300000); // 5
minutes

    if (recentErrorCount > 10) {
      await this.alertManager.sendAlert({
        level: 'warning',
        message: `High error rate detected: ${errorPattern}`,
        context: { count: recentErrorCount, timeWindow: '5 minutes' }
      });
    }
  }
}
```

This comprehensive error handling system ensures robust operation of AI services with proper error categorization, automatic recovery, and monitoring capabilities.