Interface Documentation

Comprehensive documentation for all AI abstraction layer interfaces and their contracts.

Table of Contents

- Base Interfaces
- LLM Service Interface
- Embeddings Service Interface
- Chat Service Interface
- Factory Interfaces

Base Interfaces

BaseAIService

The foundation interface that all AI services must implement.

```
interface BaseAIService {
  readonly provider: AIProvider;
  readonly serviceType: AIServiceType;
  readonly version: string;

  // Configuration methods
  getConfig(): BaseAIConfig;
  updateConfig(config: Partial<BaseAIConfig>): Promise<void>;

  // Health and status
  healthCheck(): Promise<boolean>;
  getStatus(): Promise<ServiceStatus>;

  // Model information
  listModels(): Promise<AIResponse<string[]>>;
  getModelInfo(model: string): Promise<AIResponse<ModelInfo>>;

  // Lifecycle
  initialize(): Promise<void>;
  destroy(): Promise<void>;
}
```

Key Points:
- All services inherit from this base interface
- Provides common functionality for health monitoring and configuration
- Lifecycle methods ensure proper resource management

Expected Behavior

healthCheck()
- Should return `true` if service is operational
- May perform a lightweight API call to verify connectivity
- Should complete within 5 seconds

getStatus()
- Returns detailed status including latency and rate limits
- Used for monitoring and load balancing decisions

Example Implementation:

```
class OpenAILLMService implements BaseAIService {
  readonly provider = 'openai';
  readonly serviceType = 'llm';
  readonly version = '1.0.0';

  async healthCheck(): Promise<boolean> {
    try {
      const response = await this.api.get('/models');
      return response.status === 200;
    } catch {
      return false;
    }
  }

  async getStatus(): Promise<ServiceStatus> {
    const startTime = Date.now();
    const healthy = await this.healthCheck();
    const latency = Date.now() - startTime;

    return {
      healthy,
      latency,
      rateLimits: this.extractRateLimits(),
      metadata: { lastCheck: new Date().toISOString() }
    };
  }
}
```

LLM Service Interface

LLMService

Extends `BaseAIService` to provide text generation capabilities.

```
interface LLMService extends BaseAIService {
  serviceType: 'llm';

  // Core generation methods
  generate(request: LLMGenerationRequest): Promise<AIResponse<LLMGenerationResponse>>;
  generateStream(request: LLMGenerationRequest): Promise<AIStreamResponse<LLMStreamChun
k>>;
  generateBatch(requests: LLMGenerationRequest[], options?: BatchOptions): Promise<AIR-
esponse<LLMGenerationResponse[]>>;

  // Convenience methods
  chat(messages: AIMessage[], model?: string, options?:
Partial<LLMGenerationRequest>): Promise<AIResponse<LLMGenerationResponse>>;
  chatStream(messages: AIMessage[], model?: string, options?: Partial<LLMGenerationRe-
quest>): Promise<AIStreamResponse<LLMStreamChunk>>;
  complete(prompt: string, model?: string, options?: CompletionOptions): Promise<AIRe-
sponse<string>>;

  // Advanced features
  callFunction(messages: AIMessage[], functions: FunctionDefinition[], model?:
string): Promise<AIResponse<LLMGenerationResponse>>;
  generateFromTemplate(template: string, variables: Record<string, any>, model?:
string): Promise<AIResponse<string>>;
  compressHistory(history: ChatHistory, targetTokens: number): Promise<AIResponse<AIMes
sage[]>>;

  // Utility methods
  countTokens(text: string, model?: string): Promise<AIResponse<number>>;
  estimateCost(tokens: number, model?: string): Promise<AIResponse<number>>;
  isModelSupported(model: string): boolean;
}
```

Request/Response Contracts

LLMGenerationRequest

```
interface LLMGenerationRequest {
  messages: AIMessage[];              // Required: conversation history
  model?: string;                     // Optional: override default model
  temperature?: number;               // 0-2, controls randomness
  maxTokens?: number;                 // Maximum tokens to generate
  topP?: number;                      // 0-1, nucleus sampling
  topK?: number;                      // Top-k sampling
  frequencyPenalty?: number;          // -2 to 2, reduce repetition
  presencePenalty?: number;           // -2 to 2, encourage new topics
  stop?: string | string[];           // Stop sequences
  seed?: number;                      // For reproducible outputs
  streaming?: boolean;                // Enable streaming
  context?: RequestContext;           // Request metadata
  tools?: FunctionDefinition[];       // Available functions
  responseFormat?: { type: 'text' | 'json_object'; schema?: any };
}
```

LLMGenerationResponse

```
interface LLMGenerationResponse {
  id: string;                         // Unique response ID
  model: string;                      // Model used
  choices: Array<{
    index: number;                    // Choice index
    message: AIMessage;               // Generated message
    finishReason: 'stop' | 'length' | 'content_filter' | 'tool_calls';
    logprobs?: any;                   // Token probabilities (if requested)
  }>;
  usage: TokenUsage;                  // Token consumption
  metadata: AIMetadata;               // Request metadata
  created: string;                    // Creation timestamp
}
```

Usage Examples

Basic Text Generation

```
const response = await llmService.generate({
  messages: [
    { role: 'system', content: 'You are a helpful assistant.' },
    { role: 'user', content: 'Explain quantum computing.' }
  ],
  model: 'gpt-4',
  maxTokens: 500,
  temperature: 0.7
});

console.log(response.data?.choices[0]?.message.content);
```

Streaming Generation

```
const stream = await llmService.generateStream({
  messages: [{ role: 'user', content: 'Write a long story' }],
  streamingOptions: {
    onProgress: (status, chunk) => {
      if (chunk?.choices[0]?.delta?.content) {
        process.stdout.write(chunk.choices[0].delta.content);
      }
    },
    onComplete: (result) => console.log('\nGeneration complete')
  }
});

for await (const chunk of stream) {
  // Process stream chunks
}
```

Function Calling

```
const functions: FunctionDefinition[] = [
  {
    name: 'get_weather',
    description: 'Get current weather for a location',
    parameters: {
      type: 'object',
      properties: {
        location: { type: 'string', description: 'City name' }
      },
      required: ['location']
    }
  }
];

const response = await llmService.callFunction(
  [{ role: 'user', content: 'What\'s the weather in New York?' }],
  functions,
  'gpt-4'
);

// Handle function calls in response
const toolCalls = response.data?.choices[0]?.message.toolCalls;
if (toolCalls) {
  for (const call of toolCalls) {
    if (call.function.name === 'get_weather') {
      const args = JSON.parse(call.function.arguments);
      const weather = await getWeatherData(args.location);
      // Continue conversation with function result
    }
  }
}
```

Embeddings Service Interface

EmbeddingsService

Provides text embedding and semantic search capabilities.

```
interface EmbeddingsService extends BaseAIService {
  serviceType: 'embeddings';

  // Core embedding methods
  embed(request: EmbeddingRequest): Promise<AIResponse<EmbeddingResponse>>;
  embedBatch(texts: string[], model?: string, options?: BatchEmbeddingOptions):
Promise<AIResponse<EmbeddingResponse>>;
  embedDocuments(documents: Document[], model?: string, chunkingOptions?: ChunkingOp-
tions): Promise<AIResponse<EmbeddedDocument[]>>;

  // Convenience methods
  embedText(text: string, model?: string): Promise<AIResponse<number[]>>;
  embedTexts(texts: string[], model?: string): Promise<AIResponse<number[][]>>;

  // Similarity operations
  calculateSimilarity(embedding1: number[], embedding2: number[]): number;
  findMostSimilar(queryEmbedding: number[], candidateEmbeddings: number[][], k?:
number): Array<{ index: number; score: number }>;

  // Vector operations
  normalizeEmbedding(embedding: number[]): number[];
  combineEmbeddings(embeddings: number[][], weights?: number[], method?: 'average' | 'w
eighted_average' | 'max'): number[];

  // Text processing
  chunkText(text: string, options: ChunkingOptions): Promise<AIResponse<TextChunk[]>>;
  preprocessText(text: string, options?: PreprocessingOptions): string;

  // Model information
  getEmbeddingDimensions(model: string): Promise<AIResponse<number>>;
  getMaxInputLength(model: string): Promise<AIResponse<number>>;

  // Vector store integration
  createVectorStore(name: string, config?: VectorStoreConfig): Promise<AIResponse<Vec-
torStore>>;
}
```

Usage Examples

Basic Embeddings

```
const response = await embeddingsService.embed({
  input: 'The quick brown fox jumps over the lazy dog',
  model: 'text-embedding-3-small'
});

const embedding = response.data?.data[0]?.embedding;
console.log('Embedding dimensions:', embedding?.length);
```

Semantic Search

```
const documents = [
  { id: '1', content: 'Machine learning is a subset of AI', metadata: { category:
'AI' } },
  { id: '2', content: 'Python is a programming language', metadata: { category: 'Pro-
gramming' } },
  { id: '3', content: 'Neural networks are used in deep learning', metadata: {
category: 'AI' } }
];

// Embed documents
const embeddedDocs = await embeddingsService.embedDocuments(documents);

// Search query
const queryEmbedding = await embeddingsService.embedText('artificial intelligence');

// Find similar documents
const similarities = embeddingsService.findMostSimilar(
  queryEmbedding.data!,
  embeddedDocs.data!.map(doc => doc.embedding),
  2
);

console.log('Most similar documents:', similarities);
```

Document Chunking

```
const longText = 'Very long document content...';

const chunks = await embeddingsService.chunkText(longText, {
  strategy: 'sentence',
  maxChunkSize: 500,
  overlapSize: 50,
  preserveSentences: true
});

console.log(`Document split into ${chunks.data?.length} chunks`);
```

Chat Service Interface

ChatService

High-level conversational AI interface with RAG support.

```
interface ChatService extends BaseAIService {
  serviceType: 'chat';

  readonly llmService: LLMService;
  readonly embeddingsService?: EmbeddingsService;

  // Core chat methods
  chat(request: ChatRequest): Promise<AIResponse<ChatResponse>>;
  chatStream(request: ChatRequest): Promise<AIStreamResponse<ChatStreamChunk>>;

  // Session management
  createSession(userId?: string, tenantId?: string, title?: string):
Promise<AIResponse<ChatSession>>;
  getSession(sessionId: string): Promise<AIResponse<ChatSession | null>>;
  updateSession(sessionId: string, updates: Partial<ChatSession>): Promise<AIResponse<C
hatSession>>;
  deleteSession(sessionId: string): Promise<AIResponse<boolean>>;

  // Message management
  addMessage(sessionId: string, message: AIMessage): Promise<AIResponse<boolean>>;
  getMessages(sessionId: string, limit?: number, offset?: number): Promise<AIResponse<A
IMessage[]>>;

  // Memory management
  compressSessionMemory(sessionId: string): Promise<AIResponse<boolean>>;
  clearSessionMemory(sessionId: string): Promise<AIResponse<boolean>>;

  // RAG functionality
  enableRAG(sessionId: string, options: RAGOptions): Promise<AIResponse<boolean>>;
  addToKnowledgeBase(name: string, documents: Document[], tenantId?: string): Promise<A
IResponse<string[]>>;
  searchKnowledgeBase(query: string, name?: string, tenantId?: string): Promise<AIRe-
sponse<SimilaritySearchResult[]>>;

  // Tool management
  registerTool(tool: ChatTool): Promise<AIResponse<boolean>>;
  listTools(): Promise<AIResponse<ChatTool[]>>;

  // Conversation features
  generateTitle(sessionId: string): Promise<AIResponse<string>>;
  summarizeConversation(sessionId: string): Promise<AIResponse<string>>;
  suggestFollowUpQuestions(sessionId: string): Promise<AIResponse<string[]>>;
}
```

Usage Examples

Basic Chat

```
// Create session
const session = await chatService.createSession('user123', 'tenant1', 'AI Discussion');

// Send message
const response = await chatService.chat({
  message: 'What is machine learning?',
  sessionId: session.data!.id,
  model: 'gpt-4',
  temperature: 0.7,
  maxTokens: 500
});

console.log('AI Response:', response.data?.message.content);
```

Streaming Chat

```
const stream = await chatService.chatStream({
  message: 'Explain quantum computing in detail',
  sessionId: sessionId,
  streamingOptions: {
    onProgress: (status, chunk) => {
      if (chunk?.delta?.content) {
        process.stdout.write(chunk.delta.content);
      }
    }
  }
});

for await (const chunk of stream) {
  // Handle streaming response
}
```

RAG-Enhanced Chat

```
// Add documents to knowledge base
await chatService.addToKnowledgeBase('company-docs', [
  { id: '1', content: 'Company policy document...', metadata: { type: 'policy' } },
  { id: '2', content: 'Product documentation...', metadata: { type: 'product' } }
]);

// Enable RAG for session
await chatService.enableRAG(sessionId, {
  enabled: true,
  knowledgeBase: 'company-docs',
  maxResults: 5,
  threshold: 0.7,
  rerank: true
});

// Chat with RAG
const response = await chatService.chat({
  message: 'What is our return policy?',
  sessionId: sessionId,
  ragOptions: {
    enabled: true,
    maxResults: 3
  }
});

// Response includes retrieved context
console.log('Sources:', response.data?.context?.retrievedDocuments);
```

Tool Integration

```
// Register custom tool
await chatService.registerTool({
  name: 'calculate',
  description: 'Perform mathematical calculations',
  parameters: {
    type: 'object',
    properties: {
      expression: { type: 'string', description: 'Math expression to evaluate' }
    },
    required: ['expression']
  },
  execute: async (args) => {
    // Safe evaluation of math expression
    return eval(args.expression);
  },
  enabled: true
});

// Chat with tools enabled
const response = await chatService.chat({
  message: 'What is 2 + 2 * 3?',
  sessionId: sessionId,
  toolsEnabled: true
});
```

Factory Interfaces

AIServiceFactory

Creates and manages AI service instances.

```
interface AIServiceFactory {
  // Service creation
  createLLMService(config: LLMConfig): Promise<LLMService>;
  createEmbeddingsService(config: EmbeddingsConfig): Promise<EmbeddingsService>;
  createChatService(config: ChatConfig): Promise<ChatService>;

  // Generic creation
  createService<T extends BaseAIService>(type: AIServiceType, config: BaseAIConfig): Pr
omise<T>;

  // Service discovery
  getSupportedProviders(serviceType: AIServiceType): AIProvider[];
  getProviderCapabilities(provider: AIProvider, serviceType: AIServiceType): Promise<Pr
oviderCapabilities>;

  // Validation
  validateConfig(config: BaseAIConfig): Promise<ConfigValidationResult>;
  testConnection(config: BaseAIConfig): Promise<ConnectionTestResult>;
}
```

Usage Examples

Service Creation

```
const factory = new AIServiceFactory();

// Create OpenAI LLM service
const openaiConfig = ConfigFactory.openai.llm('your-api-key');
const llmService = await factory.createLLMService(openaiConfig);

// Create Voyage AI embeddings service
const voyageConfig = ConfigFactory.voyageai.embeddings('your-api-key');
const embeddingsService = await factory.createEmbeddingsService(voyageConfig);

// Create chat service with both
const chatConfig = ConfigFactory.createChatConfig('openai', 'voyageai');
const chatService = await factory.createChatService(chatConfig);
```

Service Discovery

```
// Get supported providers
const llmProviders = factory.getSupportedProviders('llm');
console.log('LLM providers:', llmProviders);

// Get provider capabilities
const capabilities = await factory.getProviderCapabilities('openai', 'llm');
console.log('OpenAI capabilities:', capabilities);
```

Error Handling Contracts

All interface methods return `AIResponse<T>` which includes error information:

```
type AIResponse<T> = {
  success: true;
  data: T;
  metadata?: Record<string, any>;
  timestamp: string;
} | {
  success: false;
  error: AIErrorDetails;
  timestamp: string;
};
```

Error Handling Pattern

```
const response = await service.someMethod(params);

if (response.success) {
  // Handle successful response
  console.log('Data:', response.data);
} else {
  // Handle error
  console.error('Error:', response.error.message);

  if (response.error.retryable) {
    // Retry logic
  }
}
```

Next Steps

- Error Handling Guide (./error-handling.md)
- Configuration Guide (./configuration.md)
- Examples (./examples.md)
- Best Practices (../README.md#best-practices)