Configuration Guide

Complete guide to configuring the AI Abstraction Layer for different environments and use cases.

Table of Contents

Configuration Overview

The AI Abstraction Layer uses a hierarchical configuration system that supports:

- **Multiple configuration sources** (environment variables, files, programmatic)
- **Environment-specific overrides** (development, staging, production)
- **Provider-specific configurations** for each AI service
- **Runtime validation** and type safety
- **Secret management** integration
- **Multi-tenant support** with tenant-level overrides

Configuration Hierarchy

```
1. Environment Variables (highest priority)
2. Configuration Files (ai-config.json)
3. Package.json (aiConfig section)
4. Programmatic Configuration
5. Default Values (lowest priority)
```

Environment-Based Configuration

Environment Detection

The system automatically detects the environment from multiple sources:

```
# Primary environment variable
export AI_ENVIRONMENT=production

# Standard Node.js environment
export NODE_ENV=production

# Generic environment variable
export ENVIRONMENT=production
```

Environment Settings

Each environment has different default settings:

Setting	Development	Staging	Production	Test
Timeout	60s	30s	15s	10s
Retry Attempts	5	3	2	1
Metrics	✅	✅	✅	❌
Caching	❌	✅	✅	❌
Logging	Debug	Info	Warn	Error
Circuit Breaker	❌	✅	✅	❌

Environment Variables

```
# API Keys
OPENAI_API_KEY=sk-...
ANTHROPIC_API_KEY=sk-ant-api03-...
VOYAGEAI_API_KEY=pa-...
COHERE_API_KEY=...
ABACUSAI_API_KEY=...

# API Endpoints (optional)
OPENAI_API_ENDPOINT=https://api.openai.com/v1
ANTHROPIC_API_ENDPOINT=https://api.anthropic.com/v1

# Global Settings
AI_TIMEOUT=30000
AI_RETRY_ATTEMPTS=3
AI_ENABLE_METRICS=true
AI_ENABLE_CACHING=false
AI_LOG_LEVEL=info
AI_RATE_LIMIT_STRATEGY=exponential_backoff

# Environment
AI_ENVIRONMENT=production
```

Provider-Specific Configuration

OpenAI Configuration

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

const openaiConfig = ConfigFactory.openai.llm('your-api-key', {
  // Basic settings
  defaultModel: 'gpt-4',
  defaultTemperature: 0.7,
  defaultMaxTokens: 2000,

  // OpenAI-specific
  organization: 'org-123456789',
  project: 'proj-abc123',

  // Advanced settings
  streamingSupported: true,
  functionCallingSupported: true,
  visionSupported: false,

  // Context windows per model
  contextWindow: {
    'gpt-4': 128000,
    'gpt-4-turbo': 128000,
    'gpt-3.5-turbo': 16384
  },

  // Cost tracking (optional)
  costPerToken: {
    'gpt-4': { input: 0.00003, output: 0.00006 },
    'gpt-3.5-turbo': { input: 0.0000015, output: 0.000002 }
  }
});
```

Anthropic Configuration

```
const anthropicConfig = ConfigFactory.anthropic.llm('your-api-key', {
  defaultModel: 'claude-3-sonnet-20240229',
  defaultTemperature: 0.7,
  defaultMaxTokens: 4000,

  // Anthropic-specific
  version: '2023-06-01',

  // Model support
  supportedModels: [
    'claude-3-opus-20240229',
    'claude-3-sonnet-20240229',
    'claude-3-haiku-20240307'
  ],

  contextWindow: {
    'claude-3-opus-20240229': 200000,
    'claude-3-sonnet-20240229': 200000,
    'claude-3-haiku-20240307': 200000
  }
});
```

Abacus.AI Configuration

```
const abacusaiConfig = ConfigFactory.abacusai.llm('your-api-key', {
  defaultModel: 'gpt-4.1-mini',
  defaultTemperature: 0.7,
  defaultMaxTokens: 3000,

  // Abacus.AI-specific
  instanceId: 'your-instance-id', // optional

  // Supported models
  supportedModels: [
    'gpt-4.1-mini',
    'claude-3-opus',
    'claude-3-sonnet',
    'claude-3-haiku'
  ],

  // Streaming support
  streamingSupported: true
});
```

Voyage AI Configuration

```
const voyageaiConfig = ConfigFactory.voyageai.embeddings('your-api-key', {
  defaultModel: 'voyage-2',
  batchSize: 100,
  maxBatchSize: 128,

  supportedModels: [
    'voyage-large-2',
    'voyage-2',
    'voyage-lite-02-instruct'
  ],

  dimensions: {
    'voyage-large-2': 1536,
    'voyage-2': 1024,
    'voyage-lite-02-instruct': 1024
  }
});
```

Cohere Configuration

```
const cohereConfig = ConfigFactory.cohere.llm('your-api-key', {
  defaultModel: 'command-r',
  defaultTemperature: 0.7,
  defaultMaxTokens: 2000,

  // Cohere-specific
  version: 'latest',

  supportedModels: [
    'command-r-plus',
    'command-r',
    'command'
  ]
});
```

Configuration Sources

1. Configuration Files

ai-config.json

```
{
  "openai": {
    "llm": {
      "apiKey": "sk-...",
      "defaultModel": "gpt-4",
      "defaultTemperature": 0.7,
      "streamingSupported": true,
      "organization": "org-123456789"
    },
    "embeddings": {
      "apiKey": "sk-...",
      "defaultModel": "text-embedding-3-small",
      "batchSize": 100
    }
  },
  "anthropic": {
    "llm": {
      "apiKey": "sk-ant-api03-...",
      "defaultModel": "claude-3-sonnet-20240229",
      "version": "2023-06-01"
    }
  },
  "global": {
    "timeout": 30000,
    "retryAttempts": 3,
    "enableMetrics": true,
    "enableCaching": false,
    "logLevel": "info"
  },
  "environment": "production"
}
```

package.json

```
{
  "name": "my-app",
  "version": "1.0.0",
  "aiConfig": {
    "openai": {
      "llm": {
        "defaultModel": "gpt-3.5-turbo",
        "defaultTemperature": 0.8
      }
    },
    "global": {
      "timeout": 25000
    }
  }
}
```

2. Programmatic Configuration

```
import { ConfigLoader, ConfigUtils } from './lib/ai/config';

// Load configuration
const config = await ConfigUtils.loadConfig('openai', 'llm', {
  environment: 'production',
  validateConfig: true,
  loadDefaults: true,
  overrides: {
    defaultModel: 'gpt-4-turbo',
    timeout: 45000
  }
});

// Custom configuration loader
const loader = new ConfigLoader();
loader.addSource({
  name: 'database',
  priority: 75,
  async load() {
    return await loadConfigFromDatabase();
  }
});
```

3. Builder Pattern

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

const config = await ConfigFactory.builder('openai', 'llm')
  .apiKey(process.env.OPENAI_API_KEY!)
  .model('gpt-4')
  .temperature(0.7)
  .maxTokens(2000)
  .streaming(true)
  .timeout(30000)
  .retries(3)
  .metrics(true)
  .caching(false)
  .set('organization', 'org-123456789')
  .build(true); // validate = true
```

4. Multi-Provider Configuration

```
// Create multi-provider setup
const multiConfig = ConfigFactory.multiProvider('llm', 'openai', ['anthropic',
'cohere']);

// Custom multi-provider with specific weights
const weightedConfig = ConfigFactory.createMultiProviderConfig('llm', 'openai', ['an-
thropic'], {
  loadBalancing: {
    strategy: 'weighted',
    weights: {
      openai: 0.7,
      anthropic: 0.3
    }
  },
  healthCheck: {
    enabled: true,
    interval: 30000,
    timeout: 5000
  }
});
```

Validation and Security

Configuration Validation

```
import { ConfigValidator, ValidationUtils } from './lib/ai/config';

// Validate configuration
const validator = new ConfigValidator();
const result = await validator.validateConfig(config);

if (!result.valid) {
  console.error('Validation errors:');
  result.errors.forEach(error => {
    console.error(`- ${error.field}: ${error.message} (${error.severity})`);
  });

  console.warn('Warnings:');
  result.warnings.forEach(warning => console.warn(`- ${warning}`));
}

// Quick validation
const isValid = await ValidationUtils.quickValidate(config);

// API key validation
const isValidKey = await ValidationUtils.validateApiKey('openai', 'sk-...');
```

Secret Management

```
import { ConfigUtils } from './lib/ai/config';

// Create secrets provider
const secretsProvider = ConfigUtils.createMemorySecretsProvider({
  'openai_key': 'sk-actual-key-value',
  'anthropic_key': 'sk-ant-actual-key'
});

// Load config with secrets resolution
const config = await ConfigUtils.loadConfig('openai', 'llm', {
  secretsProvider,
  overrides: {
    apiKey: 'secret:openai_key' // Will be resolved from secrets provider
  }
});

// Custom secrets provider (e.g., AWS Secrets Manager)
const awsSecretsProvider = {
  async getSecret(key: string): Promise<string | null> {
    const secretsManager = new AWS.SecretsManager();
    try {
      const result = await secretsManager.getSecretValue({ SecretId: key }).promise();
      return result.SecretString || null;
    } catch {
      return null;
    }
  },
  async setSecret(key: string, value: string): Promise<void> {
    // Implementation for setting secrets
  },
  async deleteSecret(key: string): Promise<void> {
    // Implementation for deleting secrets
  }
};
```

Environment Validation

```
import { EnvironmentUtils } from './lib/ai/config';

// Validate current environment setup
const validation = await EnvironmentUtils.validateSetup();

if (!validation.valid) {
  console.error('Environment validation issues:');
  validation.issues.forEach(issue => console.error(`- ${issue}`));
  process.exit(1);
}

// Get environment information
const envInfo = EnvironmentUtils.getInfo();
console.log('Environment info:', envInfo);
```

Multi-Tenant Configuration

Tenant-Specific Overrides

```typescript
interface TenantConfig {
  tenantId: string;
  overrides: Partial<BaseAIConfig>;
  limits: {
    maxTokensPerRequest?: number;
    maxRequestsPerHour?: number;
    allowedModels?: string[];
  };
  features: {
    streamingEnabled?: boolean;
    functionCallingEnabled?: boolean;
    ragEnabled?: boolean;
  };
}

class TenantConfigManager {
  private tenantConfigs = new Map<string, TenantConfig>();

  async getConfigForTenant(
    tenantId: string,
    provider: AIProvider,
    serviceType: AIServiceType
  ): Promise<BaseAIConfig> {
    const baseConfig = await ConfigUtils.loadConfig(provider, serviceType);
    const tenantConfig = this.tenantConfigs.get(tenantId);

    if (!tenantConfig) {
      return baseConfig;
    }

    // Apply tenant overrides
    return {
      ...baseConfig,
      ...tenantConfig.overrides,
      // Add tenant context
      headers: {
        ...baseConfig.headers,
        'X-Tenant-ID': tenantId
      }
    };
  }

  setTenantConfig(tenantId: string, config: TenantConfig): void {
    this.tenantConfigs.set(tenantId, config);
  }
}

// Usage
const tenantManager = new TenantConfigManager();

tenantManager.setTenantConfig('tenant1', {
  tenantId: 'tenant1',
  overrides: {
    defaultModel: 'gpt-3.5-turbo', // Cost-conscious tenant
    defaultMaxTokens: 1000,
    timeout: 20000
  },
  limits: {
    maxTokensPerRequest: 2000,
    maxRequestsPerHour: 1000,
    allowedModels: ['gpt-3.5-turbo', 'text-embedding-3-small']
  },
```

```
  features: {
    streamingEnabled: true,
    functionCallingEnabled: false,
    ragEnabled: false
  }
});
```

Dynamic Configuration Updates

```
class DynamicConfigManager {
  private configCache = new Map<string, any>();
  private configListeners = new Map<string, Function[]>();

  async updateConfig(
    provider: AIProvider,
    serviceType: AIServiceType,
    updates: Partial<BaseAIConfig>
  ): Promise<void> {
    const configKey = `${provider}:${serviceType}`;
    const currentConfig = this.configCache.get(configKey) || {};
    const newConfig = { ...currentConfig, ...updates };

    // Validate new configuration
    const validation = await new ConfigValidator().validateConfig(newConfig);
    if (!validation.valid) {
      throw new Error(`Invalid configuration: ${validation.errors.map(e => e.message).j
oin(', ')}`);
    }

    // Update cache
    this.configCache.set(configKey, newConfig);

    // Notify listeners
    const listeners = this.configListeners.get(configKey) || [];
    listeners.forEach(listener => listener(newConfig));
  }

  onConfigChange(
    provider: AIProvider,
    serviceType: AIServiceType,
    callback: (config: BaseAIConfig) => void
  ): void {
    const configKey = `${provider}:${serviceType}`;
    const listeners = this.configListeners.get(configKey) || [];
    listeners.push(callback);
    this.configListeners.set(configKey, listeners);
  }
}
```

Configuration Templates and Presets

Development Preset

```
import { EnvironmentPresets } from './lib/ai/config';

// Development configuration with verbose logging and longer timeouts
const devConfig = EnvironmentPresets.development.openai.llm('your-api-key');
// Results in:
// - timeout: 60000
// - retryAttempts: 5
// - enableMetrics: true
// - enableCaching: false
// - logLevel: 'debug'
```

Production Preset

```
const prodConfig = EnvironmentPresets.production.openai.llm('your-api-key');
// Results in:
// - timeout: 15000
// - retryAttempts: 2
// - enableMetrics: true
// - enableCaching: true
// - logLevel: 'warn'
// - rateLimitStrategy: 'exponential_backoff'
```

Custom Templates

```typescript
import { ConfigTemplate } from './lib/ai/config';

const customTemplate: ConfigTemplate = {
  name: 'High Performance LLM',
  description: 'Optimized for high-throughput applications',
  provider: 'openai',
  serviceType: 'llm',
  template: {
    provider: 'openai',
    service: 'llm',
    apiKey: '',
    defaultModel: 'gpt-3.5-turbo',
    defaultTemperature: 0.5,
    defaultMaxTokens: 1000,
    timeout: 10000,
    retryAttempts: 1,
    enableMetrics: true,
    enableCaching: true,
    rateLimitStrategy: 'none'
  },
  requiredFields: ['apiKey'],
  optionalFields: ['defaultModel', 'timeout']
};

// Register template
const factory = new ConfigFactory();
factory.registerTemplate(customTemplate);

// Use template
const config = factory.createServiceConfig('openai', 'llm', {
  apiKey: 'your-key'
});
```

Best Practices

1. Environment Separation

```
// Use different configurations for different environments
const getConfig = (env: string) => {
  const baseConfig = {
    provider: 'openai' as const,
    service: 'llm' as const,
    apiKey: process.env.OPENAI_API_KEY!
  };

  switch (env) {
    case 'development':
      return {
        ...baseConfig,
        defaultModel: 'gpt-3.5-turbo',
        timeout: 60000,
        retryAttempts: 5,
        enableMetrics: true
      };
    case 'production':
      return {
        ...baseConfig,
        defaultModel: 'gpt-4',
        timeout: 15000,
        retryAttempts: 2,
        enableMetrics: true,
        enableCaching: true
      };
    default:
      return baseConfig;
  }
};
```

2. Secret Management

```
// Never hardcode API keys
const BAD_EXAMPLE = {
  apiKey: 'sk-1234567890abcdef...' // DON'T DO THIS
};

// Use environment variables
const GOOD_EXAMPLE = {
  apiKey: process.env.OPENAI_API_KEY!
};

// Use secrets manager for production
const PRODUCTION_EXAMPLE = {
  apiKey: 'secret:production/openai/api-key'
};
```

3. Configuration Validation

```
// Always validate configuration before use
async function createService(config: LLMConfig): Promise<LLMService> {
  // Validate first
  const validation = await new ConfigValidator().validateConfig(config);
  if (!validation.valid) {
    throw new Error(`Invalid configuration: ${validation.errors.map(e => e.message).joi
n(', ')}`);
  }

  // Create service
  return new OpenAILLMService(config);
}
```

4. Configuration Hot Reloading

```
class HotReloadConfigManager {
  private watchers = new Map<string, any>();

  watchConfigFile(filePath: string, callback: (config: any) => void): void {
    const fs = require('fs');

    const watcher = fs.watch(filePath, (eventType: string) => {
      if (eventType === 'change') {
        try {
          const newConfig = JSON.parse(fs.readFileSync(filePath, 'utf8'));
          callback(newConfig);
        } catch (error) {
          console.error('Failed to reload config:', error);
        }
      }
    });

    this.watchers.set(filePath, watcher);
  }

  stopWatching(filePath: string): void {
    const watcher = this.watchers.get(filePath);
    if (watcher) {
      watcher.close();
      this.watchers.delete(filePath);
    }
  }
}
```

5. Cost-Aware Configuration

```
interface CostConfig {
  provider: AIProvider;
  model: string;
  costPerToken: { input: number; output: number };
  dailyBudget?: number;
  warningThreshold?: number;
}

class CostAwareConfigManager {
  private costs = new Map<string, number>();

  async createConfigWithBudget(
    provider: AIProvider,
    serviceType: AIServiceType,
    budget: number
  ): Promise<BaseAIConfig> {
    const config = await ConfigUtils.loadConfig(provider, serviceType);

    // Choose model based on budget
    if (budget < 10) {
      // Low budget - use cheaper models
      if (provider === 'openai') {
        (config as LLMConfig).defaultModel = 'gpt-3.5-turbo';
      }
    } else if (budget > 100) {
      // High budget - use premium models
      if (provider === 'openai') {
        (config as LLMConfig).defaultModel = 'gpt-4';
      }
    }

    return config;
  }

  trackUsage(provider: AIProvider, tokens: number, costPerToken: number): void {
    const key = `${provider}:${new Date().toDateString()}`;
    const currentCost = this.costs.get(key) || 0;
    this.costs.set(key, currentCost + (tokens * costPerToken));
  }
}
```

6. Configuration Monitoring

```
class ConfigMonitor {
  private configHistory = new Map<string, any[]>();

  logConfigChange(
    provider: AIProvider,
    serviceType: AIServiceType,
    oldConfig: any,
    newConfig: any
  ): void {
    const key = `${provider}:${serviceType}`;
    const history = this.configHistory.get(key) || [];

    history.push({
      timestamp: new Date().toISOString(),
      oldConfig,
      newConfig,
      changes: this.getConfigDiff(oldConfig, newConfig)
    });

    this.configHistory.set(key, history.slice(-10)); // Keep last 10 changes
  }

  private getConfigDiff(oldConfig: any, newConfig: any): Record<string, any> {
    const diff: Record<string, any> = {};

    for (const key in newConfig) {
      if (oldConfig[key] !== newConfig[key]) {
        diff[key] = { old: oldConfig[key], new: newConfig[key] };
      }
    }

    return diff;
  }

  getConfigHistory(provider: AIProvider, serviceType: AIServiceType): any[] {
    return this.configHistory.get(`${provider}:${serviceType}`) || [];
  }
}
```

This configuration system provides flexibility, security, and maintainability for AI service configurations across different environments and use cases.