# 1720690009_crafting_an_ai_powered_hr_assistant

December 29, 2024

## 1 Enhanced AI-Powered HR Assistant for Nestle

### 1.1 Nestlé HR Policy Chatbot Overview

This Jupyter Notebook implements a conversational chatbot designed to answer user queries based on information contained within Nestlé's HR policy documents. It leverages several powerful technologies from the field of Natural Language Processing (NLP) and Large Language Models (LLMs) to achieve this:

1. **Document Loading and Chunking:** The notebook begins by loading the HR policy document (in PDF format) and splitting it into smaller, manageable text chunks. This is crucial because LLMs have limitations on the amount of text they can process at once.
2. **Text Embeddings:** Each text chunk is then converted into a numerical vector representation called an "embedding." These embeddings capture the semantic meaning of the text, allowing the system to understand the relationships between different pieces of information. OpenAI's embedding models are used for this purpose.
3. **Vector Database:** The embeddings are stored in a vector database (Chroma), which allows for efficient similarity search. This means that when a user asks a question, the system can quickly find the most relevant text chunks from the HR policy.
4. **Question Answering with LLM:** The most relevant text chunks are then passed to a large language model (OpenAI's GPT model) along with the user's question. The LLM uses this context to generate a coherent and informative answer.
5. **User Interface:** Finally, a user-friendly interface is created using Gradio, allowing users to easily interact with the chatbot.

In summary, this notebook demonstrates a complete workflow for building a question-answering system over a PDF document using state-of-the-art NLP and LLM techniques. This approach can be generalized to other document types and domains, making it a valuable tool for information retrieval and knowledge management. The notebook is structured with Markdown explanations and code blocks separated by functionality, making it easy to follow and understand the implementation details.

---

### 1.2 Usage and Setup Guide

#### 1.2.1 Prerequisites:

1. **Environment Setup:**

- Python 3.8 or higher
- Required packages installed
- .env file configured

2. **Required Environment Variables:**

```
OPENAI_API_KEY=your_api_key_here
PDF_DOC_PATH=path_to_your_pdf_file
```

#### 1.2.2 Running the Application:

1. Ensure all cells are run in order
2. Wait for the Gradio interface to launch
3. Interface will be available at http://127.0.0.1:7860

#### 1.2.3 Features:

- Real-time question answering
- Conversation history tracking
- Error handling and recovery
- Rate limiting protection

#### 1.2.4 Best Practices:

- Keep API keys secure
- Monitor usage rates
- Regular logging review
- Backup document sources

#### 1.2.5 Troubleshooting:

- Check environment variables
- Verify PDF file accessibility
- Monitor API rate limits
- Review error logs

### 1.3 Installation of Libraries

This code block manages the installation of required Python libraries and their dependencies. Let's examine each component:

#### 1.3.1 Core Dependencies:

- `openai` (>=1.0.0): Core API interface for OpenAI services
  - Required for: API authentication, model interactions
  - Dependencies: requests, typing-extensions
- `langchain` (>=0.1.0): Framework for LLM applications
  - Requires: openai, numpy
  - Components: Chains, agents, memory systems
  - Sub-packages:
    - `langchain-community`: Community integrations

* `langchain-openai`: OpenAI-specific modules

#### 1.3.2 Vector Storage and Document Processing:

- `chromadb`: Vector database for embedding storage
  - Handles: Similarity search, vector indexing
  - Requires: numpy, sqlalchemy
- `pypdf`: PDF document processing
  - Features: Text extraction, document parsing
  - Used for: Loading and processing HR documents

#### 1.3.3 Interface and Utilities:

- `gradio`: Web interface framework
  - Purpose: Interactive chat interface
  - Dependencies: fastapi, websockets
- `tiktoken`: OpenAI's tokenizer
  - Used for: Token counting, context management
  - Critical for: Rate limiting, cost management

#### 1.3.4 System Utilities:

- `python-dotenv`: Environment variable management
  - Security: API key and configuration handling
  - Local development: .env file support
- `psutil`: System resource monitoring
  - Memory tracking
  - Performance metrics

#### 1.3.5 Standard Library Requirements:

- `logging`: Application monitoring and debugging
- `json`: Data serialization
- `datetime`: Timestamp management
- `collections`: Advanced data structures
- `typing`: Type hints and validation
- `pathlib`: Cross-platform path handling

#### 1.3.6 Installation Command:

"'bash pip install openai>=1.0.0 langchain>=0.1.0 langchain-community langchain-openai chromadb pypdf gradio tiktoken python-dotenv psutil

[15]:
```
# Install necessary libraries (run this in your notebook environment if needed)
# !pip install openai langchain langchain-community langchain-openai chromadb␣
↪pypdf gradio tiktoken python-dotenv psutil
```

## 1.4 Core Imports and Error Handling Classes

This section establishes the foundation of our application through comprehensive import management and logging configuration. Let's examine each component in detail:

### 1.4.1 Import Structure and Purpose:

1. **System and OS Operations:**
   - `os`: Environment variables, path operations
   - `sys`: Python runtime operations
   - `platform`: System identification
   - `psutil`: Resource monitoring
2. **Data Type Management:**
   - `typing`: Type hints for code reliability
     - List: For document collections
     - Dict: For configuration and responses
     - Any: For flexible type handling
     - Optional: For nullable parameters
3. **Time and Data Handling:**
   - `datetime`: Timestamp generation and tracking
   - `collections.defaultdict`: Automatic dictionary initialization
   - `time`: Performance measurements
   - `json`: Data serialization
4. **File Operations:**
   - `pathlib.Path`: Cross-platform path handling
     - File existence checking
     - Path manipulation
     - Extension handling

### 1.4.2 External Dependencies:

1. **AI and Language Processing:**
   - `openai`: OpenAI API interface
   - `langchain` components:
     - Document loaders: PDF processing
     - Text splitters: Content chunking
     - Embeddings: Vector creation
     - Chains: Processing pipeline
2. **User Interface:**
   - `gradio`: Web interface components
     - Chatbot: Conversation display
     - Textbox: User input
     - Button: Control elements

### 1.4.3 Logging Configuration:

1. **Handler Setup:**
   - File Handler:
     - Location: 'chatbot.log'

- Purpose: Persistent logging
- Mode: Append

- Stream Handler:
  - Purpose: Real-time console output
  - Level: INFO and above

2. **Format Pattern:**
   - %(asctime)s - %(name)s - %(levelname)s - [%(filename)s:%(lineno)d] - %(message)s
     - `asctime`: Timestamp with milliseconds
     - `name`: Logger component identifier
     - `levelname`: Severity (INFO/WARNING/ERROR)
     - `filename:lineno`: Source location
     - `message`: Log content

[16]:
```
import os
import sys
import logging
import platform
import psutil
from typing import List, Dict, Any, Optional
from datetime import datetime
from collections import defaultdict
from functools import wraps
from time import time
from pathlib import Path
import json

# Core dependencies
import openai
from dotenv import load_dotenv

# Document processing
from langchain.document_loaders import PyPDFLoader, UnstructuredFileLoader
from langchain.text_splitter import CharacterTextSplitter

# LangChain components
from langchain_openai import OpenAI, OpenAIEmbeddings
from langchain.vectorstores import Chroma
from langchain.chains import RetrievalQA

# UI
import gradio as gr


# Configure logging
logging.basicConfig(
    level=logging.INFO,
```

```
    format='%(asctime)s - %(name)s - %(levelname)s - [%(filename)s:%(lineno)d]␣
 ↪- %(message)s',
    handlers=[
        logging.FileHandler('chatbot.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)
```

## 1.5 Error Handling and Configuration System

This section implements a comprehensive error handling hierarchy and configuration management system. Let's examine each component in detail:

### 1.5.1 Error Handling Architecture:

1. **Base Error Class (ChatbotError):**
   - **Core Features:**
     - Custom message formatting
     - Error code management
     - Timestamp tracking
     - JSON serialization
     - Logging integration
   - **Implementation Details:**
     - Inherits from Python's Exception
     - Automated timestamp generation
     - Standardized error formatting
     - Logging level configuration
2. **Specialized Error Classes:**

   A. **ConfigurationError:**
   - **Purpose:** Configuration and setup issues
   - **Additional Fields:**
     - config_key: Affected configuration parameter
     - Validation context
     - Setting details

   B. **DocumentLoadError:**
   - **Purpose:** Document processing failures
   - **Additional Fields:**
     - file_path: Problematic document location
     - Format details
     - Processing stage information

   C. **ModelError:**
   - **Purpose:** AI model operation issues
   - **Additional Fields:**

- model_name: Affected AI model
- query: Problematic input
- Context information
- Performance metrics

#### 1.5.2 Configuration Management System:

1. **Default Configuration:**
   - **Core Settings:**
     - CHUNK_SIZE: Text segment length (1000 chars)
     - CHUNK_OVERLAP: Segment overlap (0 chars)
     - MODEL_TEMPERATURE: Response randomness (0)
     - EMBEDDING_MODEL: Vector model selection
   - **Advanced Settings:**
     - MAX_RETRIES: Operation retry limit
     - RATE_LIMIT_PER_SECOND: API call throttling
     - LOG_LEVEL: Logging detail control
2. **Configuration Methods:**
   - **load_from_env:**
     - Environment variable processing
     - Type conversion handling
     - Validation enforcement
     - Error reporting
   - **get_settings:**
     - Configuration serialization
     - Private attribute filtering
     - Format standardization
3. **Type Handling:**
   - String to numeric conversion
   - Boolean interpretation
   - List/dictionary parsing
   - Default value management

#### 1.5.3 Metrics Tracking System:

1. **Performance Metrics:**
   - Operation duration tracking
   - Resource utilization
   - API call monitoring
   - Error rate calculation
2. **Statistical Analysis:**
   - Mean/min/max calculations
   - Error pattern detection
   - Performance trending
   - Resource usage analysis

#### 1.5.4 Implementation Features:

1. **Error Handling:**

- Hierarchical error classification
- Detailed error context capture
- Automated logging integration
- Recovery mechanism support

2. **Configuration Management:**
   - Centralized settings control
   - Environment-based configuration
   - Type-safe value handling
   - Validation enforcement
3. **Metrics Collection:**
   - Real-time performance tracking
   - Statistical aggregation
   - Error pattern analysis
   - Resource utilization monitoring

#### 1.5.5 Best Practices:

1. Consistent error formatting
2. Comprehensive error context
3. Type-safe configuration
4. Detailed logging integration
5. Performance metric tracking
6. Resource usage monitoring

[17]:
```python
class ChatbotError(Exception):
    """Base exception class for chatbot errors"""
    def __init__(self, message: str = None, error_code: str = None):
        self.message = message or "An unexpected error occurred in the chatbot"
        self.error_code = error_code or "CHATBOT_ERROR"
        self.timestamp = datetime.now()

        formatted_message = f"[{self.error_code}] {self.timestamp}: {self.
 ↪message}"
        super().__init__(formatted_message)

    def to_dict(self) -> Dict[str, Any]:
        """Convert error details to dictionary for logging or API responses"""
        return {
            "error_code": self.error_code,
            "message": self.message,
            "timestamp": self.timestamp.isoformat(),
            "error_type": self.__class__.__name__
        }

    def log_error(self, log_level: int = logging.ERROR):
        """Log the error with specified logging level"""
        logger.log(log_level, json.dumps(self.to_dict(), indent=2))
```

```python
class ConfigurationError(ChatbotError):
    """Raised when there are configuration-related issues"""
    def __init__(self, message: str = None, config_key: str = None):
        self.config_key = config_key
        error_message = f"Configuration error{f' for {config_key}' if␣
 ↪config_key else ''}: {message}"
        super().__init__(
            message=error_message,
            error_code="CONFIG_ERROR"
        )

    def to_dict(self) -> Dict[str, Any]:
        """Add configuration-specific details to error dictionary"""
        error_dict = super().to_dict()
        error_dict["config_key"] = self.config_key
        return error_dict

class DocumentLoadError(ChatbotError):
    """Raised when there are issues loading documents"""
    def __init__(self, message: str = None, file_path: str = None):
        self.file_path = file_path
        error_message = f"Failed to load document{f' {file_path}' if file_path␣
 ↪else ''}: {message}"
        super().__init__(
            message=error_message,
            error_code="DOC_LOAD_ERROR"
        )

    def to_dict(self) -> Dict[str, Any]:
        """Add document-specific details to error dictionary"""
        error_dict = super().to_dict()
        error_dict["file_path"] = self.file_path
        return error_dict

class ModelError(ChatbotError):
    """Raised when there are issues with the AI model"""
    def __init__(self, message: str = None, model_name: str = None, query: str␣
 ↪= None):
        self.model_name = model_name
        self.query = query
        error_message = f"Model error{f' for {model_name}' if model_name else␣
 ↪''}: {message}"
        super().__init__(
            message=error_message,
            error_code="MODEL_ERROR"
        )
```

```
    def to_dict(self) -> Dict[str, Any]:
        """Add model-specific details to error dictionary"""
        error_dict = super().to_dict()
        error_dict.update({
            "model_name": self.model_name,
            "query": self.query if self.query else None
        })
        return error_dict
class Config:
    """Configuration settings for the chatbot"""
    CHUNK_SIZE = 1000
    CHUNK_OVERLAP = 0
    MODEL_TEMPERATURE = 0
    EMBEDDING_MODEL = "text-embedding-ada-002"

    @classmethod
    def get_settings(cls) -> Dict[str, Any]:
        """Returns all configuration settings as a dictionary"""
        return {k: v for k, v in cls.__dict__.items()
                if not k.startswith('_')}
```

## 1.6 Environment Setup and API Key Management

This section implements secure environment configuration and API key management with comprehensive validation and error handling. Let's examine the components in detail:

### 1.6.1 Configuration Class Implementation:

1. **Default Configuration Management:**
   - **Structure:**
     - Dictionary-based storage
     - Type-safe value handling
     - Immutable defaults
     - Runtime override support
   - **Default Values:**
     - Text processing parameters
     - Model configuration
     - Performance limits
     - Logging settings
2. **Configuration Methods:**
   - **Initialization:**
     - Default value copying
     - Load time tracking
     - Validation setup
     - Error handling initialization
   - **Environment Loading:**
     - Variable detection
     - Type conversion

- Validation
- Error reporting

- **Value Access:**
  - Type-safe retrieval
  - Default handling
  - Error checking
  - Logging integration

#### 1.6.2 Environment Variable Management:

1. **Required Variables:**
   - **OpenAI Configuration:**
     - API_KEY: Authentication token
     - Model selection
     - Rate limits
   - **Document Management:**
     - PDF_DOC_PATH: Document location
     - Format requirements
     - Access permissions
2. **Validation Process:**
   - **Security Checks:**
     - API key format validation
     - Path existence verification
     - Permission checking
     - Format validation
   - **Error Handling:**
     - Detailed error messages
     - Recovery suggestions
     - Logging integration
     - Security considerations

#### 1.6.3 Metrics System:

1. **Performance Tracking:**
   - **Duration Metrics:**
     - Operation timing
     - Statistical analysis
     - Threshold monitoring
     - Performance logging
   - **Error Tracking:**
     - Error categorization
     - Pattern detection
     - Impact analysis
     - Resolution tracking
2. **Statistical Analysis:**
   - **Calculations:**
     - Mean/min/max values
     - Error frequencies

- Performance trends
- Resource utilization

#### 1.6.4 Implementation Features:

1. **Security:**
   - Secure variable handling
   - Path sanitization
   - Access control
   - Error masking
2. **Validation:**
   - Type checking
   - Format verification
   - Permission validation
   - Path confirmation
3. **Monitoring:**
   - Performance tracking
   - Error logging
   - Resource monitoring
   - Usage analytics

#### 1.6.5 Error Recovery:

1. **Handling Strategies:**
   - Graceful degradation
   - Retry mechanisms
   - Alternative paths
   - User notification
2. **Logging Integration:**
   - Detailed error context
   - Stack trace capture
   - System state recording
   - Recovery attempts

[18]:
```python
class Config:
    """Configuration settings for the chatbot"""

    # Default configuration values
    DEFAULT_CONFIG = {
        'CHUNK_SIZE': 1000,
        'CHUNK_OVERLAP': 0,
        'MODEL_TEMPERATURE': 0,
        'EMBEDDING_MODEL': "text-embedding-ada-002",
        'MAX_RETRIES': 3,
        'RATE_LIMIT_PER_SECOND': 5,
        'LOG_LEVEL': logging.INFO
    }
```

```
    def __init__(self):
        self._config = self.DEFAULT_CONFIG.copy()
        self._load_time = datetime.now()

    def load_from_env(self) -> None:
        """Load configuration from environment variables"""
        try:
            for key in self.DEFAULT_CONFIG:
                env_value = os.getenv(f'CHATBOT_{key}')
                if env_value:
                    # Convert string values to appropriate types
                    if isinstance(self.DEFAULT_CONFIG[key], int):
                        self._config[key] = int(env_value)
                    elif isinstance(self.DEFAULT_CONFIG[key], float):
                        self._config[key] = float(env_value)
                    else:
                        self._config[key] = env_value

            logger.info("Configuration loaded successfully")

        except ValueError as e:
            raise ConfigurationError(
                message=f"Invalid configuration value: {str(e)}",
                config_key=key
            )

    def get(self, key: str) -> Any:
        """Get configuration value"""
        if key not in self._config:
            raise ConfigurationError(
                message=f"Configuration key not found: {key}",
                config_key=key
            )
        return self._config[key]

    def to_dict(self) -> Dict[str, Any]:
        """Convert configuration to dictionary"""
        return {
            'config': self._config,
            'load_time': self._load_time.isoformat(),
            'age_seconds': (datetime.now() - self._load_time).total_seconds()
        }

def load_env() -> tuple:
    """
    Load and validate environment variables
    Returns:
```

```
        tuple: (api_key, pdf_doc_path)
    Raises:
        ConfigurationError: If required environment variables are missing
    """
    try:
        load_dotenv(verbose=True)

        required_vars = {
            'OPENAI_API_KEY': 'OpenAI API key',
            'PDF_DOC_PATH': 'PDF document path'
        }

        env_vars = {}
        for var, description in required_vars.items():
            value = os.getenv(var)
            if not value:
                raise ConfigurationError(
                    message=f"Missing {description}",
                    config_key=var
                )
            env_vars[var] = value

        # Validate PDF path exists
        pdf_path = Path(env_vars['PDF_DOC_PATH'])
        if not pdf_path.exists():
            raise ConfigurationError(
                message=f"PDF file does not exist: {pdf_path}",
                config_key='PDF_DOC_PATH'
            )

        logger.info("Environment variables loaded successfully")
        return env_vars['OPENAI_API_KEY'], str(pdf_path)

    except Exception as e:
        if not isinstance(e, ChatbotError):
            e = ConfigurationError(
                message=f"Environment loading failed: {str(e)}"
            )
        e.log_error()
        raise

class MetricsTracker:
    """Track performance metrics for the chatbot"""

    def __init__(self):
        self.metrics = defaultdict(list)
```

```
    def record_duration(self, operation: str, duration: float):
        """Record duration of an operation"""
        self.metrics[f"{operation}_duration"].append(duration)
        logger.debug(f"{operation} took {duration:.2f} seconds")

    def record_error(self, error: ChatbotError):
        """Record an error"""
        self.metrics["errors"].append(error.to_dict())
        error.log_error()

    def get_statistics(self) -> Dict[str, Any]:
        """Get statistical summary of metrics"""
        stats = {}
        for metric, values in self.metrics.items():
            if metric.endswith('_duration'):
                if values:
                    stats[metric] = {
                        'mean': sum(values) / len(values),
                        'min': min(values),
                        'max': max(values),
                        'count': len(values)
                    }
            elif metric == "errors":
                stats["error_count"] = len(values)
                stats["error_types"] = defaultdict(int)
                for error in values:
                    stats["error_types"][error["error_type"]] += 1

        return stats

# Global instances
config = Config()
metrics = MetricsTracker()
```

## 1.7 Text Processing and Vector Store Creation System

This section implements a sophisticated document processing pipeline that converts raw text into searchable vector embeddings. Let's examine the complete implementation:

### 1.7.1 Document Loading System:

1. **File Processing Pipeline:**
   - **Input Validation:**
     - File existence verification
     - Format checking
     - Permission validation
     - Size limitations
   - **Loading Strategy:**

    - Format-specific loaders
    - Chunk management
    - Memory optimization
    - Error handling
2. **Retry Mechanism:**
    - **Implementation:**
        - Configurable retry count
        - Exponential backoff
        - Failure tracking
        - Success verification
    - **Error Management:**
        - Detailed error capture
        - Recovery attempts
        - Resource cleanup
        - State restoration

#### 1.7.2 TextProcessor Class Architecture:

1. **Initialization Components:**
    - **Text Splitter Setup:**
        - Chunk size configuration
        - Overlap management
        - Token limits
        - Format handling
    - **Embedding Configuration:**
        - Model selection
        - Parameter optimization
        - Cache initialization
        - Resource allocation
2. **Document Processing:**
    - **Splitting Logic:**
        - Content segmentation
        - Context preservation
        - Token management
        - Format maintenance
    - **Embedding Generation:**
        - Vector creation
        - Dimension management
        - Quality verification
        - Performance optimization

#### 1.7.3 Vector Store Implementation:

1. **Chroma Database Setup:**
    - **Initialization:**
        - Index creation
        - Storage configuration
        - Performance tuning

    - Backup strategy
  - **Management:**
    - Data persistence
    - Index optimization
    - Query preparation
    - Resource efficiency
2. **Performance Features:**
  - **Caching System:**
    - Result caching
    - Cache invalidation
    - Memory management
    - Performance monitoring
  - **Batch Processing:**
    - Chunk batching
    - Parallel processing
    - Resource allocation
    - Error handling

#### 1.7.4 Metrics and Monitoring:

1. **Performance Tracking:**
  - **Time Measurements:**
    - Processing duration
    - Operation latency
    - System overhead
    - Resource usage
  - **Quality Metrics:**
    - Embedding accuracy
    - Processing success rate
    - Error frequency
    - Resource efficiency
2. **Statistical Analysis:**
  - **Data Collection:**
    - Operation timing
    - Resource utilization
    - Error patterns
    - System health

#### 1.7.5 Error Management:

1. **Error Categories:**
  - Document loading failures
  - Processing errors
  - Resource constraints
  - System limitations
2. **Recovery Mechanisms:**
  - Retry strategies
  - Alternative processing

- Resource reallocation
- State recovery

[19]:
```python
def load_document(file_path: str) -> List:
    """
    Load and process document based on file type with enhanced error handling

    Args:
        file_path (str): Path to the document
    Returns:
        List: Loaded document pages
    Raises:
        DocumentLoadError: If document loading fails
    """
    start_time = time()
    try:
        logger.info(f"Starting document load: {file_path}")
        file_path = Path(file_path)

        # Validate file existence and permissions
        if not file_path.exists():
            raise DocumentLoadError(
                message="File does not exist",
                file_path=str(file_path)
            )
        if not file_path.is_file():
            raise DocumentLoadError(
                message="Path is not a file",
                file_path=str(file_path)
            )

        # Select appropriate loader based on file extension
        if file_path.suffix.lower() == '.pdf':
            loader = PyPDFLoader(str(file_path))
        else:
            raise DocumentLoadError(
                message=f"Unsupported file type: {file_path.suffix}",
                file_path=str(file_path)
            )

        # Load document with retries
        max_retries = config.get('MAX_RETRIES')
        for attempt in range(max_retries):
            try:
                documents = loader.load()

                # Log success metrics
```

```
                duration = time() - start_time
                metrics.record_duration('document_load', duration)
                logger.info(
                    f"Successfully loaded document with {len(documents)} pages␣
↪in {duration:.2f} seconds"
                )
                return documents

            except Exception as e:
                if attempt < max_retries - 1:
                    logger.warning(f"Retry {attempt + 1}/{max_retries} failed:␣
↪{str(e)}")
                    continue
                raise DocumentLoadError(
                    message=f"Failed after {max_retries} attempts: {str(e)}",
                    file_path=str(file_path)
                )

    except Exception as e:
        if not isinstance(e, ChatbotError):
            e = DocumentLoadError(
                message=str(e),
                file_path=str(file_path)
            )
        metrics.record_error(e)
        raise

class TextProcessor:
    """Handles text splitting and embedding creation with enhanced error␣
↪handling and metrics"""

    def __init__(self):
        try:
            self.text_splitter = CharacterTextSplitter(
                chunk_size=config.get('CHUNK_SIZE'),
                chunk_overlap=config.get('CHUNK_OVERLAP')
            )
            self.embeddings = OpenAIEmbeddings()
            self._cache = {}
            self._stats = defaultdict(int)
            logger.info("TextProcessor initialized successfully")

        except Exception as e:
            raise ConfigurationError(
                message=f"Failed to initialize TextProcessor: {str(e)}"
            )
```

```python
    def _split_documents(self, documents: List) -> List:
        """
        Split documents into chunks with error handling

        Args:
            documents (List): List of documents to split
        Returns:
            List: Split text chunks
        """
        start_time = time()
        try:
            texts = self.text_splitter.split_documents(documents)
            duration = time() - start_time
            metrics.record_duration('text_splitting', duration)
            self._stats['total_chunks'] = len(texts)
            logger.info(f"Split documents into {len(texts)} chunks in {duration:
↪.2f} seconds")
            return texts

        except Exception as e:
            error = ModelError(
                message=f"Failed to split documents: {str(e)}",
                model_name="text_splitter"
            )
            metrics.record_error(error)
            raise error

    def process_documents(self, documents: List) -> Chroma:
        """
        Process documents into embeddings and store in Chroma

        Args:
            documents (List): List of document pages
        Returns:
            Chroma: Vector store with processed documents
        """
        start_time = time()
        try:
            logger.info("Starting document processing")

            # Split documents
            texts = self._split_documents(documents)

            # Create vector store
            logger.info("Creating vector store")
            db = Chroma.from_documents(texts, self.embeddings)
```

```
            # Record success metrics
            duration = time() - start_time
            metrics.record_duration('document_processing', duration)
            self._stats['processing_time'] = duration

            logger.info(
                f"Successfully created vector store in {duration:.2f} seconds. "
                f"Stats: {json.dumps(self._stats, indent=2)}"
            )
            return db

        except Exception as e:
            if not isinstance(e, ChatbotError):
                e = ModelError(
                    message=f"Failed to process documents: {str(e)}",
                    model_name="document_processor"
                )
            metrics.record_error(e)
            raise e

    def get_stats(self) -> Dict[str, Any]:
        """Get processing statistics"""
        return {
            'stats': self._stats,
            'cache_size': len(self._cache),
            'embedding_model': config.get('EMBEDDING_MODEL')
        }
```

## 1.8 Rate Limiting and Question-Answering System

This section implements the core QA functionality with sophisticated rate limiting protection. Let's examine each component:

### 1.8.1 Rate Limiting Decorator:

1. **Architecture:**

- Rolling window implementation
- Function-specific tracking
- Automated cleanup
- Performance monitoring

2. **Features:**

- Configurable rate limits
- Per-function restrictions
- Timestamp management
- Violation tracking

3. **Error Management:**

- Custom error generation
- Metric recording
- Violation logging
- Recovery handling

#### 1.8.2 QASystem Class Components:

1. **Initialization Framework:**

- Model configuration
- Chain setup
- Retriever integration
- History management

2. **Query Processing Pipeline:**

- Input validation
- Rate limiting
- Context retrieval
- Response generation

3. **Response Management:**

- Format standardization
- History recording
- Performance tracking
- Error handling

#### 1.8.3 Performance Monitoring:

1. **Metric Collection:**

- Query timing
- Success rates
- Resource usage
- Error frequency

2. **Statistical Analysis:**

- Average response time
- Error patterns
- Resource utilization
- System health

#### 1.8.4 History Management:

1. **Query Tracking:**

- Timestamp recording
- Duration monitoring
- Context preservation
- Pattern analysis

2. **Historical Analysis:**

- Performance trends
- Usage patterns
- Error correlations
- System optimization

#### 1.8.5 Implementation Features:

1. **Rate Control:**

- Dynamic limiting
- Adaptive thresholds
- Load balancing
- Resource protection

2. **Error Handling:**

- Custom error types
- Context preservation
- Recovery strategies
- User notification

3. **Performance Optimization:**

- Response caching
- Resource management
- Load distribution
- System monitoring

[20]:
```python
def rate_limit(max_per_second: int):
    """
    Enhanced rate limiting decorator with metrics tracking

    Args:
        max_per_second (int): Maximum number of calls allowed per second
    """
    calls = defaultdict(list)

    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            now = time()
            func_name = func.__name__

            # Clean old calls
            calls[func_name] = [c for c in calls[func_name] if c > now - 1]

            # Check rate limit
            if len(calls[func_name]) >= max_per_second:
                error = ChatbotError(
```

```
                    message=f"Rate limit of {max_per_second} calls per second␣
↪exceeded for {func_name}",
                    error_code="RATE_LIMIT_ERROR"
                )
                metrics.record_error(error)
                raise error

            # Record call
            calls[func_name].append(now)

            # Execute function
            try:
                result = func(*args, **kwargs)
                metrics.record_duration(func_name, time() - now)
                return result
            except Exception as e:
                if not isinstance(e, ChatbotError):
                    e = ModelError(
                        message=str(e),
                        model_name=func_name
                    )
                metrics.record_error(e)
                raise e

        return wrapper
    return decorator

class QASystem:
    """Enhanced question-answering system with metrics and error handling"""

    def __init__(self, vector_store: Chroma):
        """Initialize QA system with vector store"""
        try:
            self.qa_chain = RetrievalQA.from_chain_type(
                llm=OpenAI(temperature=config.get('MODEL_TEMPERATURE')),
                chain_type="stuff",
                retriever=vector_store.as_retriever()
            )
            self.query_history = []
            logger.info("QA system initialized successfully")

        except Exception as e:
            error = ModelError(
                message=f"Failed to initialize QA system: {str(e)}",
                model_name="QASystem"
            )
            metrics.record_error(error)
```

```
            raise error

    @rate_limit(max_per_second=5)
    def get_response(self, query: str) -> str:
        """
        Get response for user query with enhanced error handling

        Args:
            query (str): User's question
        Returns:
            str: Generated response
        """
        start_time = time()
        try:
            # Validate query
            if not query or not query.strip():
                raise ModelError(
                    message="Empty query provided",
                    query=query
                )

            # Process query
            response = self.qa_chain.invoke({"query": query})

            # Record successful query
            duration = time() - start_time
            self.query_history.append({
                'query': query,
                'timestamp': datetime.now().isoformat(),
                'duration': duration
            })

            logger.info(f"Query processed successfully in {duration:.2f}␣
↪seconds")
            return response['result'] if isinstance(response, dict) else␣
↪response

        except Exception as e:
            if not isinstance(e, ChatbotError):
                e = ModelError(
                    message=f"Failed to process query: {str(e)}",
                    model_name="QASystem",
                    query=query
                )
            metrics.record_error(e)
            raise e
```

```python
    def get_stats(self) -> Dict[str, Any]:
        """Get QA system statistics"""
        return {
            'total_queries': len(self.query_history),
            'average_duration': sum(q['duration'] for q in self.query_history) /
↪ len(self.query_history) if self.query_history else 0,
            'query_history': self.query_history[-10:]  # Last 10 queries
        }
```

## 1.9 Chat Interface and User Interaction System

This section implements a sophisticated conversational interface using Gradio, managing user interactions and system responses. Let's examine each component:

### 1.9.1 Chat Interface Architecture:

1. **State Management:**
   - Conversation history tracking
   - Session metrics collection
   - Interface statistics
   - Performance monitoring
2. **Interaction Components:**
   - Message processing pipeline
   - Response generation
   - History updates
   - Error management
3. **Metric Collection:**
   - Interaction counting
   - Error tracking
   - Response timing
   - Resource usage

### 1.9.2 User Interface Components:

1. **Display Elements:**
   - Chat history display
   - Input field configuration
   - Control button layout
   - Statistics panel
2. **Interactive Features:**
   - Real-time message processing
   - Dynamic history updates
   - Error display handling
   - System status indicators
3. **Control Elements:**
   - Conversation reset
   - Statistics display
   - System monitoring

- Error recovery

#### 1.9.3 Response Processing:

1. **Message Handling:**
   - Input validation
   - Context management
   - Response generation
   - History updates
2. **Error Management:**
   - Error detection
   - User notification
   - Recovery procedures
   - State preservation
3. **Performance Tracking:**
   - Response timing
   - Resource monitoring
   - Error frequency
   - System health

#### 1.9.4 Interface Features:

1. **Conversation Management:**
   - History preservation
   - Context maintenance
   - State recovery
   - Session tracking
2. **User Experience:**
   - Responsive updates
   - Clear feedback
   - Error notifications
   - Status indicators
3. **System Integration:**
   - QA system connection
   - Metric collection
   - Error propagation
   - Resource management

#### 1.9.5 Data Management:

1. **History Storage:**
   - Message archiving
   - Context preservation
   - Timestamp tracking
   - State management
2. **Statistics Collection:**
   - Usage metrics
   - Error tracking
   - Performance data

- System health

#### 1.9.6 Interface Design:

1. **Layout Organization:**
   - Clear structure
   - Intuitive controls
   - Status indicators
   - Error displays
2. **User Interaction:**
   - Immediate feedback
   - Clear messaging
   - Error handling
   - Status updates

[21]:
```python
class ChatInterface:
    """Enhanced chat interface with error handling and metrics"""

    def __init__(self, qa_system: QASystem):
        self.qa_system = qa_system
        self.conversation_history = []
        self.interface_metrics = {
            'total_interactions': 0,
            'error_count': 0,
            'start_time': datetime.now().isoformat()
        }

    def respond(self, message: str, history: List) -> tuple:
        """
        Process user message with enhanced error handling

        Args:
            message (str): User's message
            history (List): Conversation history
        Returns:
            tuple: (message, updated_history)
        """
        self.interface_metrics['total_interactions'] += 1

        try:
            response = self.qa_system.get_response(message)
            self.conversation_history.append({
                'user_message': message,
                'bot_response': response,
                'timestamp': datetime.now().isoformat()
            })
            return response, history + [[message, response]]
```

```
        except Exception as e:
            self.interface_metrics['error_count'] += 1
            error_message = f"Error: {str(e)}"
            if not isinstance(e, ChatbotError):
                metrics.record_error(ChatbotError(error_message))
            return error_message, history + [[message, error_message]]

    def create_interface(self) -> gr.Blocks:
        """Create enhanced Gradio interface"""
        with gr.Blocks() as demo:
            # Header
            gr.Markdown("# Nestlé HR Chatbot")

            # Chat interface
            chatbot = gr.Chatbot()
            msg = gr.Textbox(
                label="Ask about Nestle's HR policies",
                placeholder="Type your question here...",
                show_label=True
            )

            # Control buttons
            with gr.Row():
                clear = gr.Button("Clear Conversation")
                show_stats = gr.Button("Show Statistics")

            # Statistics display
            stats_output = gr.JSON(label="System Statistics", visible=False)

            # Event handlers
            msg.submit(
                self.respond,
                [msg, chatbot],
                [msg, chatbot]
            )

            clear.click(
                lambda: (None, None),
                None,
                [msg, chatbot],
                queue=False
            )

            show_stats.click(
                lambda: {
                    'interface': self.interface_metrics,
                    'qa_system': self.qa_system.get_stats(),
```

```
                    'overall_metrics': metrics.get_statistics()
                },
                None,
                stats_output
            )

        return demo
```

## 1.10 Application State Management and Initialization

This section orchestrates the overall application lifecycle, managing state, initialization, and system health. Let's examine each component:

### 1.10.1 Application State Manager:

1. **State Tracking:**
   - Component initialization status
   - System health monitoring
   - Resource allocation
   - Runtime metrics
2. **Health Management:**
   - Status updates
   - Health checks
   - Error detection
   - Recovery procedures
3. **Component Registry:**
   - Initialization tracking
   - Dependency management
   - Resource allocation
   - State preservation

### 1.10.2 Initialization Process:

1. **Environment Setup:**
   - Configuration loading
   - Variable validation
   - Resource allocation
   - System preparation
2. **Component Initialization:**
   - Sequential startup
   - Dependency resolution
   - State verification
   - Error handling
3. **Resource Management:**
   - Memory allocation
   - Connection establishment
   - Cache initialization
   - System optimization

#### 1.10.3 Health Monitoring:

1. **Status Tracking:**
   - Component health
   - System performance
   - Resource usage
   - Error conditions
2. **Metrics Collection:**
   - Performance data
   - Resource utilization
   - Error frequency
   - System statistics

#### 1.10.4 Error Management:

1. **Error Detection:**
   - Component failures
   - Resource issues
   - System problems
   - Performance degradation
2. **Recovery Procedures:**
   - Error handling
   - State recovery
   - Resource cleanup
   - System restoration

#### 1.10.5 System Features:

1. **State Management:**
   - Comprehensive tracking
   - Status monitoring
   - Resource oversight
   - Performance analysis
2. **Health Reporting:**
   - Status updates
   - Error notifications
   - Performance metrics
   - System analytics
3. **Resource Control:**
   - Allocation management
   - Usage monitoring
   - Optimization
   - Cleanup procedures

#### 1.10.6 Implementation Benefits:

1. **System Reliability:**
   - Consistent state tracking
   - Error detection

- Recovery mechanisms
- Health monitoring

2. **Performance Optimization:**
   - Resource management
   - Usage tracking
   - System tuning
   - Performance metrics
3. **Maintenance Support:**
   - Status reporting
   - Error tracking
   - System analytics
   - Resource monitoring

[22]:
```python
class ApplicationState:
    """Manages global application state and initialization"""

    def __init__(self):
        self.startup_time = datetime.now()
        self.initialized = False
        self.components = {}
        self.health_status = {
            'status': 'initializing',
            'last_check': self.startup_time.isoformat()
        }

    def update_health(self, status: str, message: str = None):
        """Update application health status"""
        self.health_status.update({
            'status': status,
            'last_check': datetime.now().isoformat(),
            'message': message
        })
        logger.info(f"Health status updated: {status} - {message}")

    def get_status(self) -> Dict[str, Any]:
        """Get complete application status"""
        return {
            'health': self.health_status,
            'uptime_seconds': (datetime.now() - self.startup_time).
 ↪total_seconds(),
            'initialized': self.initialized,
            'components': list(self.components.keys()),
            'config': config.to_dict(),
            'metrics': metrics.get_statistics()
        }

def initialize_application() -> ApplicationState:
```

```
    """
    Initialize all application components with comprehensive error handling

    Returns:
        ApplicationState: Application state manager
    """
    app_state = ApplicationState()

    try:
        logger.info("Starting application initialization")

        # Load environment variables and configuration
        logger.info("Loading environment and configuration")
        api_key, pdf_path = load_env()
        config.load_from_env()
        app_state.components['config'] = config

        # Load and process documents
        logger.info("Loading documents")
        documents = load_document(pdf_path)
        app_state.components['documents'] = f"Loaded {len(documents)} pages"

        # Initialize text processor
        logger.info("Initializing text processor")
        processor = TextProcessor()
        vector_store = processor.process_documents(documents)
        app_state.components['text_processor'] = processor
        app_state.components['vector_store'] = vector_store

        # Initialize QA system
        logger.info("Initializing QA system")
        qa_system = QASystem(vector_store)
        app_state.components['qa_system'] = qa_system

        # Initialize chat interface
        logger.info("Setting up chat interface")
        interface = ChatInterface(qa_system)
        app_state.components['interface'] = interface

        # Mark initialization as complete
        app_state.initialized = True
        app_state.update_health('healthy', 'Application initialized␣
↪successfully')

        logger.info("Application initialization completed successfully")
        return app_state
```

```
    except Exception as e:
        error_message = f"Application initialization failed: {str(e)}"
        logger.error(error_message)
        app_state.update_health('error', error_message)

        if not isinstance(e, ChatbotError):
            e = ChatbotError(error_message)
        metrics.record_error(e)
        raise

def cleanup_resources():
    """Cleanup application resources"""
    try:
        logger.info("Cleaning up application resources")
        # Add cleanup logic here (e.g., closing file handles, database␣
 ↪connections)

    except Exception as e:
        logger.error(f"Error during cleanup: {str(e)}")

    finally:
        logger.info("Application shutdown complete")

def main():
    """Main application entry point with enhanced error handling and␣
 ↪monitoring"""
    app_state = None

    try:
        # Initialize application
        app_state = initialize_application()

        # Create and launch interface
        demo = app_state.components['interface'].create_interface()

        # Configure shutdown handling
        def handle_shutdown():
            logger.info("Shutdown initiated")
            cleanup_resources()

        # Launch interface with shutdown handling
        demo.launch(
            share=False,  # Set to True to create a public URL
            prevent_thread_lock=True
        )

        # Log successful startup
```

```
        logger.info(
            "Application started successfully\n" +
            json.dumps(app_state.get_status(), indent=2)
        )

    except Exception as e:
        error_message = f"Application startup failed: {str(e)}"
        logger.error(error_message)

        if app_state:
            app_state.update_health('error', error_message)

        if not isinstance(e, ChatbotError):
            e = ChatbotError(error_message)
        metrics.record_error(e)

        raise

    finally:
        cleanup_resources()

if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        logger.critical(f"Fatal error: {str(e)}")
        sys.exit(1)
```

```
2024-12-29 18:57:54,867 - __main__ - INFO - [2334330675.py:43] - Starting
application initialization
2024-12-29 18:57:54,867 - __main__ - INFO - [2334330675.py:46] - Loading
environment and configuration
2024-12-29 18:57:54,869 - __main__ - INFO - [465277660.py:92] - Environment
variables loaded successfully
2024-12-29 18:57:54,869 - __main__ - INFO - [465277660.py:33] - Configuration
loaded successfully
2024-12-29 18:57:54,869 - __main__ - INFO - [2334330675.py:52] - Loading
documents
2024-12-29 18:57:54,869 - __main__ - INFO - [5827190.py:14] - Starting document
load: dataset/the_nestle_hr_policy_pdf_2012.pdf
2024-12-29 18:57:55,133 - __main__ - INFO - [5827190.py:47] - Successfully
loaded document with 8 pages in 0.26 seconds
2024-12-29 18:57:55,133 - __main__ - INFO - [2334330675.py:57] - Initializing
text processor
2024-12-29 18:57:55,144 - __main__ - INFO - [5827190.py:82] - TextProcessor
initialized successfully
2024-12-29 18:57:55,145 - __main__ - INFO - [5827190.py:126] - Starting document
processing
```

```
2024-12-29 18:57:55,145 - __main__ - INFO - [5827190.py:104] - Split documents
into 7 chunks in 0.00 seconds
2024-12-29 18:57:55,145 - __main__ - INFO - [5827190.py:132] - Creating vector
store
2024-12-29 18:57:55,551 - httpx - INFO - [_client.py:1025] - HTTP Request: POST
https://api.openai.com/v1/embeddings "HTTP/1.1 200 OK"
2024-12-29 18:57:55,674 - __main__ - INFO - [5827190.py:140] - Successfully
created vector store in 0.53 seconds. Stats: {
  "total_chunks": 7,
  "processing_time": 0.5295591354370117
}
2024-12-29 18:57:55,675 - __main__ - INFO - [2334330675.py:64] - Initializing QA
system
2024-12-29 18:57:55,693 - __main__ - INFO - [3138667305.py:60] - QA system
initialized successfully
2024-12-29 18:57:55,693 - __main__ - INFO - [2334330675.py:69] - Setting up chat
interface
2024-12-29 18:57:55,693 - __main__ - INFO - [2334330675.py:20] - Health status
updated: healthy - Application initialized successfully
2024-12-29 18:57:55,694 - __main__ - INFO - [2334330675.py:77] - Application
initialization completed successfully
/opt/homebrew/anaconda3/envs/1720690009_20241228_v2/lib/python3.10/site-
packages/gradio/components/chatbot.py:242: UserWarning: You have not specified a
value for the `type` parameter. Defaulting to the 'tuples' format for chatbot
messages, but this is deprecated and will be removed in a future version of
Gradio. Please set type='messages' instead, which uses openai-style dictionaries
with 'role' and 'content' keys.
  warnings.warn(
2024-12-29 18:57:55,812 - httpx - INFO - [_client.py:1025] - HTTP Request: GET
http://127.0.0.1:7861/gradio_api/startup-events "HTTP/1.1 200 OK"
2024-12-29 18:57:55,816 - httpx - INFO - [_client.py:1025] - HTTP Request: HEAD
http://127.0.0.1:7861/ "HTTP/1.1 200 OK"

* Running on local URL:  http://127.0.0.1:7861

To create a public link, set `share=True` in `launch()`.

<IPython.core.display.HTML object>

2024-12-29 18:57:55,819 - __main__ - INFO - [2334330675.py:125] - Application
started successfully
{
  "health": {
    "status": "healthy",
    "last_check": "2024-12-29T18:57:55.693949",
    "message": "Application initialized successfully"
  },
  "uptime_seconds": 0.95167,
  "initialized": true,
```

```
  "components": [
    "config",
    "documents",
    "text_processor",
    "vector_store",
    "qa_system",
    "interface"
  ],
  "config": {
    "config": {
      "CHUNK_SIZE": 1000,
      "CHUNK_OVERLAP": 0,
      "MODEL_TEMPERATURE": 0,
      "EMBEDDING_MODEL": "text-embedding-ada-002",
      "MAX_RETRIES": 3,
      "RATE_LIMIT_PER_SECOND": 5,
      "LOG_LEVEL": 20
    },
    "load_time": "2024-12-29T18:57:54.789042",
    "age_seconds": 1.029908
  },
  "metrics": {
    "document_load_duration": {
      "mean": 0.2634608745574951,
      "min": 0.2634608745574951,
      "max": 0.2634608745574951,
      "count": 1
    },
    "text_splitting_duration": {
      "mean": 8.893013000488281e-05,
      "min": 8.893013000488281e-05,
      "max": 8.893013000488281e-05,
      "count": 1
    },
    "document_processing_duration": {
      "mean": 0.5295591354370117,
      "min": 0.5295591354370117,
      "max": 0.5295591354370117,
      "count": 1
    }
  }
}
2024-12-29 18:57:55,819 - __main__ - INFO - [2334330675.py:93] - Cleaning up
application resources
2024-12-29 18:57:55,819 - __main__ - INFO - [2334330675.py:100] - Application
shutdown complete
```

## 1.11 Testing and System Reporting Utilities

This section implements comprehensive testing procedures and system reporting capabilities. Let's examine each component of the testing and reporting framework:

### 1.11.1 Test Runner System:

1. **Environment Testing:**
   - API key validation
   - Path verification
   - Configuration testing
   - Permission checking
2. **Document Processing Tests:**
   - Loading verification
   - Document processing
   - Chunking validation
   - Format handling
3. **QA System Testing:**
   - Query processing
   - Response validation
   - Rate limiting checks
   - Error handling

### 1.11.2 System Reporting Framework:

1. **Configuration Reporting:**
   - Current settings
   - Environment state
   - System parameters
   - Runtime configuration
2. **Performance Analytics:**
   - Operation timing
   - Resource utilization
   - Response latency
   - System load
3. **Error Analysis:**
   - Error frequency
   - Error patterns
   - Impact assessment
   - Resolution tracking

### 1.11.3 System Metrics:

1. **Resource Monitoring:**
   - Memory usage
   - CPU utilization
   - Storage metrics
   - Network status
2. **Performance Statistics:**

- Response times
- Processing speed
- Resource efficiency
- System throughput

#### 1.11.4 Implementation Features:

1. **Test Execution:**
   - Sequential testing
   - Dependency handling
   - State verification
   - Error capture
2. **Report Generation:**
   - Metrics compilation
   - Statistics calculation
   - Status summaries
   - Performance analysis
3. **System Analysis:**
   - Trend identification
   - Pattern recognition
   - Performance evaluation
   - Resource assessment

#### 1.11.5 Testing Components:

1. **Functionality Testing:**
   - Core operations
   - Error handling
   - Recovery procedures
   - State management
2. **Performance Testing:**
   - Response timing
   - Resource usage
   - System limits
   - Load handling

#### 1.11.6 Reporting Features:

1. **System Status:**
   - Component health
   - Resource state
   - Error conditions
   - Performance metrics
2. **Analysis Tools:**
   - Statistical processing
   - Trend analysis
   - Pattern detection
   - Performance evaluation

[23]:
```python
def run_tests():
    """Basic application tests"""
    try:
        logger.info("Starting application tests")

        # Test environment loading
        api_key, pdf_path = load_env()
        assert api_key, "API key not loaded"
        assert pdf_path, "PDF path not loaded"

        # Test document loading
        documents = load_document(pdf_path)
        assert documents, "Documents not loaded"

        # Test text processing
        processor = TextProcessor()
        vector_store = processor.process_documents(documents)
        assert vector_store, "Vector store not created"

        # Test QA system
        qa_system = QASystem(vector_store)
        test_query = "What are the working hours?"
        response = qa_system.get_response(test_query)
        assert response, "No response received"

        logger.info("All tests passed successfully")
        return True

    except Exception as e:
        logger.error(f"Test failed: {str(e)}")
        return False

def generate_system_report() -> Dict[str, Any]:
    """Generate comprehensive system report"""
    return {
        'timestamp': datetime.now().isoformat(),
        'configuration': config.to_dict(),
        'metrics': metrics.get_statistics(),
        'environment': {
            'python_version': sys.version,
            'platform': platform.platform(),
            'memory_usage': psutil.Process().memory_info().rss / 1024 / 1024  #␣
 ↪MB
        },
        'log_summary': {
            'error_count': len([r for r in metrics.metrics['errors']]),
            'last_errors': metrics.metrics['errors'][-5:]  # Last 5 errors
```

```
        }
    }
```

[24]:
```
print('\n=======================')
print(f'Performing run tests')
print('=======================\n')

run_tests()

print('\n=======================')
print(f'Generating System Report')
print('=======================\n')
generate_system_report()
```

```
2024-12-29 18:57:55,832 - __main__ - INFO - [2109332838.py:4] - Starting
application tests
2024-12-29 18:57:55,834 - __main__ - INFO - [465277660.py:92] - Environment
variables loaded successfully
2024-12-29 18:57:55,834 - __main__ - INFO - [5827190.py:14] - Starting document
load: dataset/the_nestle_hr_policy_pdf_2012.pdf
2024-12-29 18:57:56,001 - __main__ - INFO - [5827190.py:47] - Successfully
loaded document with 8 pages in 0.17 seconds


=======================
Performing run tests
=======================


2024-12-29 18:57:56,012 - __main__ - INFO - [5827190.py:82] - TextProcessor
initialized successfully
2024-12-29 18:57:56,012 - __main__ - INFO - [5827190.py:126] - Starting document
processing
2024-12-29 18:57:56,012 - __main__ - INFO - [5827190.py:104] - Split documents
into 7 chunks in 0.00 seconds
2024-12-29 18:57:56,013 - __main__ - INFO - [5827190.py:132] - Creating vector
store
2024-12-29 18:57:56,270 - httpx - INFO - [_client.py:1025] - HTTP Request: POST
https://api.openai.com/v1/embeddings "HTTP/1.1 200 OK"
2024-12-29 18:57:56,341 - __main__ - INFO - [5827190.py:140] - Successfully
created vector store in 0.33 seconds. Stats: {
  "total_chunks": 7,
  "processing_time": 0.3287811279296875
}
2024-12-29 18:57:56,358 - __main__ - INFO - [3138667305.py:60] - QA system
initialized successfully
2024-12-29 18:57:56,741 - httpx - INFO - [_client.py:1025] - HTTP Request: POST
https://api.openai.com/v1/embeddings "HTTP/1.1 200 OK"
2024-12-29 18:57:57,164 - httpx - INFO - [_client.py:1025] - HTTP Request: POST
```

```
https://api.openai.com/v1/completions "HTTP/1.1 200 OK"
2024-12-29 18:57:57,166 - __main__ - INFO - [3138667305.py:100] - Query
processed successfully in 0.81 seconds
2024-12-29 18:57:57,166 - __main__ - INFO - [2109332838.py:26] - All tests
passed successfully

2024-12-29 18:57:56,117 - httpx - INFO - [_client.py:1025] - HTTP Request: GET
https://api.gradio.app/pkg-version "HTTP/1.1 200 OK"


========================
Generating System Report
========================


[24]: {'timestamp': '2024-12-29T18:57:57.168198',
       'configuration': {'config': {'CHUNK_SIZE': 1000,
         'CHUNK_OVERLAP': 0,
         'MODEL_TEMPERATURE': 0,
         'EMBEDDING_MODEL': 'text-embedding-ada-002',
         'MAX_RETRIES': 3,
         'RATE_LIMIT_PER_SECOND': 5,
         'LOG_LEVEL': 20},
        'load_time': '2024-12-29T18:57:54.789042',
        'age_seconds': 2.379163},
       'metrics': {'document_load_duration': {'mean': 0.21483445167541504,
         'min': 0.16620802879333496,
         'max': 0.2634608745574951,
         'count': 2},
        'text_splitting_duration': {'mean': 8.64267349243164e-05,
         'min': 8.392333984375e-05,
         'max': 8.893013000488281e-05,
         'count': 2},
        'document_processing_duration': {'mean': 0.4291701316833496,
         'min': 0.3287811279296875,
         'max': 0.5295591354370117,
         'count': 2},
        'get_response_duration': {'mean': 0.8077270984649658,
         'min': 0.8077270984649658,
         'max': 0.8077270984649658,
         'count': 1}},
       'environment': {'python_version': '3.10.12 | packaged by conda-forge | (main,
      Jun 23 2023, 22:41:52) [Clang 15.0.7 ]',
         'platform': 'macOS-15.2-arm64-arm-64bit',
         'memory_usage': 288.796875},
       'log_summary': {'error_count': 0, 'last_errors': []}}

2024-12-29 18:59:46,850 - httpx - INFO - [_client.py:1025] - HTTP Request: POST
https://api.openai.com/v1/embeddings "HTTP/1.1 200 OK"
```

```
2024-12-29 18:59:49,246 - httpx - INFO - [_client.py:1025] - HTTP Request: POST
https://api.openai.com/v1/completions "HTTP/1.1 200 OK"
2024-12-29 18:59:49,253 - __main__ - INFO - [3138667305.py:100] - Query
processed successfully in 2.66 seconds
2024-12-29 19:00:28,601 - httpx - INFO - [_client.py:1025] - HTTP Request: POST
https://api.openai.com/v1/embeddings "HTTP/1.1 200 OK"
2024-12-29 19:00:30,805 - httpx - INFO - [_client.py:1025] - HTTP Request: POST
https://api.openai.com/v1/completions "HTTP/1.1 200 OK"
2024-12-29 19:00:30,809 - __main__ - INFO - [3138667305.py:100] - Query
processed successfully in 2.39 seconds
```