PersianAI: Persian-English Translation System

Author: PersianAI Team

April 13, 2025

# Executive Summary

PersianAI is a sophisticated translation system designed to provide accurate and context-aware translations between Persian and English languages. The system leverages state-of-the-art AI models from OpenAI and Google, combined with custom processing logic to handle the unique characteristics of Persian text.

Key features include:  
• Real-time translation with WebSocket support  
• Context-aware translation improvements  
• Detailed change tracking and version control  
• Comprehensive API for integration  
• Robust error handling and monitoring

# Introduction

The PersianAI project addresses the growing need for accurate and reliable Persian-English translation services. By combining advanced AI models with specialized processing for Persian language characteristics, the system provides high-quality translations suitable for both casual and professional use.

## Project Goals

• Provide accurate Persian-English translations  
• Handle complex Persian language structures  
• Maintain context and meaning in translations  
• Offer real-time translation capabilities  
• Support integration with other systems

# System Architecture

## Overview

The system follows a microservices architecture with the following components:  
• FastAPI web server  
• Translation processing service  
• Change tracking system  
• WebSocket communication layer  
• External API integrations

## Component Interaction

Components interact through well-defined APIs and message queues, ensuring:  
• Loose coupling between services  
• Scalability and maintainability  
• Efficient resource utilization  
• Reliable error handling

# Implementation Details

## Core Components

The implementation consists of several key components:  
• FastAPI application server  
• Translation processing engine  
• Change tracking system  
• WebSocket server  
• API integration layer

## Technology Stack

The project utilizes modern technologies:  
• Python 3.9+  
• FastAPI framework  
• OpenAI and Gemini APIs  
• PostgreSQL database  
• Redis caching

# Testing and Validation

## Testing Strategy

The testing approach includes:  
• Unit tests for core functionality  
• Integration tests for API endpoints  
• Performance testing  
• Security testing  
• User acceptance testing

## Validation Methods

Translation quality is validated through:  
• Automated quality metrics  
• Manual review by language experts  
• User feedback analysis  
• A/B testing of improvements

## 6.4 Code Documentation

### translation\_bot.py

The translation\_bot.py file serves as the core of our application, implementing the FastAPI server and handling translation requests. This file contains the main application logic, API endpoints, and integration with various AI models.

#### FastAPI Application Setup

The application is initialized using FastAPI, setting up CORS middleware and route handlers:  
 - Creates FastAPI instance with custom title and description  
 - Configures CORS middleware to handle cross-origin requests  
 - Sets up WebSocket connection for real-time translation updates  
 - Implements rate limiting and request validation  
 - Handles API key management and security  
  
 Key implementation details:  
 - Uses environment variables for configuration  
 - Implements custom middleware for request logging  
 - Handles API key rotation and validation  
 - Manages concurrent request processing  
 - Provides detailed error responses

app = FastAPI(  
 title="PersianAI Translation Service",  
 description="Professional Persian to English translation service",  
 version="1.0.0"  
)  
  
# Configure CORS  
app.add\_middleware(  
 CORSMiddleware,  
 allow\_origins=["\*"],  
 allow\_credentials=True,  
 allow\_methods=["\*"],  
 allow\_headers=["\*"],  
)  
  
# Add custom middleware  
@app.middleware("http")  
async def add\_process\_time\_header(request: Request, call\_next):  
 start\_time = time.time()  
 response = await call\_next(request)  
 process\_time = time.time() - start\_time  
 response.headers["X-Process-Time"] = str(process\_time)  
 return response

#### Translation Core Logic

The translation logic includes:  
 - Integration with Azure Translator API for accurate translations  
 - Custom text preprocessing for Persian language  
 - Handling of special characters and formatting  
 - Caching mechanism for improved performance  
 - Support for multiple AI models (OpenAI and Gemini)  
 - Text chunking for handling large documents  
  
 Implementation features:  
 - Smart text chunking based on sentence boundaries  
 - Context preservation between chunks  
 - Parallel processing of chunks  
 - Result aggregation and formatting  
 - Quality validation of translations

async def translate\_text(text: str, model\_type: str = "gpt-3.5-turbo") -> str:  
 """Translate Persian text to English using specified model."""  
 try:  
 # Preprocess text  
 text = preprocess\_persian\_text(text)  
   
 # Split into chunks if needed  
 chunks = await chunk\_text(text)  
   
 # Process chunks in parallel  
 tasks = []  
 for chunk in chunks:  
 if model\_type in ["gpt-3.5-turbo", "gpt-4"]:  
 task = process\_openai\_translation(chunk, model\_type)  
 else:  
 task = process\_gemini\_translation(chunk)  
 tasks.append(task)  
   
 # Wait for all translations  
 translations = await asyncio.gather(\*tasks)  
   
 # Combine and validate results  
 result = combine\_translations(translations)  
 validate\_translation(result)  
   
 return result  
 except Exception as e:  
 logger.error(f"Translation error: {str(e)}")  
 raise

### changes.py

The changes.py file manages version control and tracks modifications to translations. It implements a sophisticated diff system that can detect and visualize changes at the word level, making it easy to review and validate translations.

#### Change Tracking System

Features of the change tracking system:  
 - Version history maintenance  
 - Diff generation between versions  
 - Rollback capability for translations  
 - Audit trail for all modifications  
 - Word-level change detection  
 - HTML-based diff visualization  
 - Change statistics and metrics  
  
 Key components:  
 - Sequence matching algorithm for word-level diffs  
 - HTML generation with color coding  
 - Change statistics calculation  
 - Version comparison utilities  
 - Rollback functionality

def track\_changes(original: str, modified: str) -> Dict[str, Any]:  
 """Track changes between original and modified text."""  
 # Split into words while preserving formatting  
 orig\_words = split\_into\_words(original)  
 mod\_words = split\_into\_words(modified)  
   
 # Create sequence matcher  
 matcher = SequenceMatcher(None, orig\_words, mod\_words)  
   
 # Process changes  
 changes = {  
 'replacements': 0,  
 'deletions': 0,  
 'insertions': 0,  
 'details': []  
 }  
   
 # Analyze each operation  
 for op, i1, i2, j1, j2 in matcher.get\_opcodes():  
 if op == 'replace':  
 changes['replacements'] += 1  
 changes['details'].append({  
 'type': 'replace',  
 'original': ' '.join(orig\_words[i1:i2]),  
 'modified': ' '.join(mod\_words[j1:j2])  
 })  
 elif op == 'delete':  
 changes['deletions'] += 1  
 changes['details'].append({  
 'type': 'delete',  
 'original': ' '.join(orig\_words[i1:i2])  
 })  
 elif op == 'insert':  
 changes['insertions'] += 1  
 changes['details'].append({  
 'type': 'insert',  
 'modified': ' '.join(mod\_words[j1:j2])  
 })  
   
 return changes

### improvements.py

The improvements.py file contains logic for enhancing translation quality through various optimization techniques and custom prompts. It implements sophisticated text processing algorithms to improve the accuracy and naturalness of translations.

#### Translation Enhancement Features

Key improvement features include:  
 - Context-aware translation refinement  
 - Persian language-specific optimizations  
 - Quality metrics calculation  
 - Suggestion system for better translations  
 - Custom prompt templates for different content types  
 - Error correction and grammar improvement  
  
 Implementation details:  
 - Custom prompt engineering for different content types  
 - Context window management  
 - Quality scoring system  
 - Grammar correction rules  
 - Style consistency checks

def enhance\_translation(text: str, context: Dict[str, Any] = None) -> str:  
 """Enhance translation quality using context and rules."""  
 try:  
 # Apply Persian-specific rules  
 text = apply\_persian\_rules(text)  
   
 # Generate context-aware prompt  
 prompt = generate\_context\_prompt(text, context)  
   
 # Get model suggestions  
 suggestions = get\_model\_suggestions(prompt)  
   
 # Apply improvements  
 improved\_text = apply\_improvements(text, suggestions)  
   
 # Validate quality  
 quality\_score = calculate\_quality\_score(improved\_text)  
   
 if quality\_score < QUALITY\_THRESHOLD:  
 # Apply additional improvements if needed  
 improved\_text = apply\_additional\_improvements(improved\_text)  
   
 return improved\_text  
 except Exception as e:  
 logger.error(f"Enhancement error: {str(e)}")  
 raise

### Test Files

The project includes a comprehensive test suite spread across multiple files, each focusing on different aspects of the system. These tests ensure reliability and maintainability of the codebase.

#### test\_changes.py

The test\_changes.py file contains unit tests for the change tracking system, ensuring accurate detection and visualization of text modifications.

#### Testing Framework

Testing features include:  
 - Unit tests for text difference detection  
 - Tests for HTML diff generation  
 - Validation of change statistics  
 - Edge case handling tests  
 - Performance benchmarks  
 - Error handling verification  
  
 Test coverage:  
 - Basic text comparison  
 - Complex formatting scenarios  
 - Unicode character handling  
 - Large text processing  
 - Error conditions

#### test\_connections.py

The test\_connections.py file verifies API connectivity and integration, ensuring reliable communication with external services.

#### API Testing Features

API testing capabilities include:  
 - Connection validation for all external APIs  
 - Response format verification  
 - Error handling testing  
 - Rate limit testing  
 - Timeout handling verification  
 - API key validation tests  
  
 Test scenarios:  
 - Successful API calls  
 - Error responses  
 - Rate limiting  
 - Network issues  
 - Authentication failures

#### test\_gemini.py

The test\_gemini.py file specifically tests Google's Gemini model integration, ensuring proper functionality of the Gemini-based translation features.

#### Gemini Model Testing

Testing features for Gemini integration:  
 - Model initialization verification  
 - Response quality assessment  
 - Configuration validation  
 - Error handling for model-specific issues  
 - Performance monitoring  
 - Resource cleanup verification  
  
 Test aspects:  
 - Model loading  
 - Response formatting  
 - Error scenarios  
 - Resource management  
 - Performance metrics

# 6.5 Function Documentation

## 6.5.1 Translation Bot Functions

### Core Functions

The core functions handle the main application setup and configuration:

create\_app() Function:

Purpose: Creates and configures the FastAPI application with all necessary middleware and settings.

Key Features:

• Initializes FastAPI with custom title and description  
• Sets up CORS middleware for cross-origin requests  
• Configures rate limiting and request validation  
• Establishes WebSocket connection handling  
• Sets up API key management and security

def create\_app():  
 """Creates and configures the FastAPI application.  
   
 Returns:  
 FastAPI: Configured FastAPI application instance.  
 """

add\_process\_time\_header() Function:

Purpose: Adds processing time information to response headers for performance monitoring.

Key Features:

• Measures request processing time  
• Adds timing information to response headers  
• Helps with performance monitoring  
• Enables response time tracking

def add\_process\_time\_header(request: Request, call\_next):  
 """Adds processing time header to response.  
   
 Args:  
 request (Request): FastAPI request object  
 call\_next: Next middleware function  
   
 Returns:  
 Response: Response with processing time header  
 """

### Translation Functions

The translation functions handle the core translation logic:

translate\_text() Function:

Purpose: Main translation endpoint that processes translation requests.

Key Features:

• Handles text translation requests  
• Supports multiple AI models  
• Manages text chunking for large inputs  
• Implements caching for performance  
• Provides real-time progress updates

async def translate\_text(request: Request):  
 """Handles text translation requests.  
   
 Args:  
 request (Request): FastAPI request object containing text and model type  
   
 Returns:  
 dict: Contains translated text  
 """

process\_openai\_translation() Function:

Purpose: Processes translations using OpenAI models.

Key Features:

• Integrates with OpenAI API  
• Handles model-specific parameters  
• Manages API rate limiting  
• Implements error handling  
• Provides fallback options

async def process\_openai\_translation(text: str, model: str) -> str:  
 """Processes translation using OpenAI API.  
   
 Args:  
 text (str): Text to translate  
 model (str): OpenAI model to use  
   
 Returns:  
 str: Translated text  
 """

process\_gemini\_translation() Function:

Purpose: Processes translations using Google's Gemini model.

Key Features:

• Integrates with Gemini API  
• Handles model-specific configurations  
• Manages API responses  
• Implements error handling  
• Provides model-specific optimizations

async def process\_gemini\_translation(text: str, model: str) -> str:  
 """Processes translation using Gemini API.  
   
 Args:  
 text (str): Text to translate  
 model (str): Gemini model to use  
   
 Returns:  
 str: Translated text  
 """

### Text Processing Functions

The text processing functions handle text manipulation and optimization:

chunk\_text() Function:

Purpose: Splits large text into manageable chunks for processing.

Key Features:

• Smart text splitting based on sentence boundaries  
• Maintains context between chunks  
• Handles special characters  
• Preserves formatting  
• Optimizes chunk size

async def chunk\_text(text: str, chunk\_size: int) -> List[str]:  
 """Splits text into chunks of specified size.  
   
 Args:  
 text (str): Text to split  
 chunk\_size (int): Maximum size of each chunk  
   
 Returns:  
 List[str]: List of text chunks  
 """

## 6.5.2 Changes Module Functions

The changes module functions handle version control and change tracking:

process\_gemini\_edit() Function:

Purpose: Processes text editing using Gemini model.

Key Features:

• Handles text editing requests  
• Integrates with Gemini API  
• Tracks changes made  
• Generates change statistics  
• Provides edit history

async def process\_gemini\_edit(text: str) -> str:  
 """Processes text editing using Gemini API.  
   
 Args:  
 text (str): Text to edit  
   
 Returns:  
 str: Edited text  
 """

process\_openai\_edit() Function:

Purpose: Processes text editing using OpenAI models.

Key Features:

• Handles text editing requests  
• Integrates with OpenAI API  
• Tracks modifications  
• Generates change reports  
• Maintains edit history

async def process\_openai\_edit(text: str, model: str = ModelType.GPT35.value) -> str:  
 """Processes text editing using OpenAI API.  
   
 Args:  
 text (str): Text to edit  
 model (str): OpenAI model to use  
   
 Returns:  
 str: Edited text  
 """

## 6.5.3 Improvements Module

The improvements module contains prompt templates and enhancement logic:

TRANSLATION\_PROMPT:

Purpose: Template for Persian to English translation.

Key Features:

• Optimized for Persian language  
• Handles cultural context  
• Maintains text formatting  
• Preserves meaning  
• Supports multiple content types

EDIT\_PROMPT:

Purpose: Template for basic Persian text editing.

Key Features:

• Basic text improvements  
• Grammar corrections  
• Style enhancements  
• Format preservation  
• Context awareness

EDIT\_PROMPT\_DETAILED:

Purpose: Template for comprehensive Persian text editing.

Key Features:

• Detailed text analysis  
• Advanced improvements  
• Style optimization  
• Format enhancement  
• Context preservation

## 6.5.4 Test Functions

The test functions ensure system reliability and functionality:

### Test Changes Functions

Functions for testing the change tracking system:

generate\_test\_text() Function:

Purpose: Generates sample Persian text for testing.

Key Features:

• Creates test data  
• Simulates real text  
• Tests various scenarios  
• Validates functionality  
• Measures performance

async def generate\_test\_text(word\_count: int) -> str:  
 """Generates sample Persian text for testing.  
   
 Args:  
 word\_count (int): Number of words to generate  
   
 Returns:  
 str: Generated test text  
 """

test\_chunking() Function:

Purpose: Tests the text chunking mechanism.

Key Features:

• Validates chunk size  
• Tests boundary conditions  
• Checks context preservation  
• Verifies formatting  
• Measures performance

async def test\_chunking() -> None:  
 """Tests the text chunking mechanism with various sizes.  
   
 Tests different word counts and chunk sizes to ensure proper text splitting.  
 """

### Test Connections Functions

Functions for testing API connectivity:

test\_openai\_connection() Function:

Purpose: Tests connection to OpenAI API.

Key Features:

• Validates API keys  
• Tests connectivity  
• Checks response format  
• Verifies error handling  
• Measures response time

async def test\_openai\_connection() -> bool:  
 """Tests connection to OpenAI API.  
   
 Returns:  
 bool: True if connection successful, False otherwise  
 """

test\_gemini\_connection() Function:

Purpose: Tests connection to Gemini API.

Key Features:

• Validates API keys  
• Tests connectivity  
• Checks response format  
• Verifies error handling  
• Measures response time

async def test\_gemini\_connection() -> bool:  
 """Tests connection to Gemini API.  
   
 Returns:  
 bool: True if connection successful, False otherwise  
 """

### API Client Management Functions

get\_sync\_openai\_client() Function:

Purpose: Initializes and caches a synchronous OpenAI client with optimized settings.

@lru\_cache()  
def get\_sync\_openai\_client():  
 """Get or create synchronous OpenAI client instance."""  
 try:  
 return OpenAI(  
 api\_key=OPENAI\_API\_KEY,  
 timeout=httpx.Timeout(60.0, connect=10.0, read=30.0, write=30.0),  
 max\_retries=3  
 )  
 except Exception as e:  
 logger.error(f"Error initializing sync OpenAI client: {str(e)}")  
 raise

get\_async\_openai\_client() Function:

Purpose: Initializes and caches an asynchronous OpenAI client for concurrent operations.

@lru\_cache()  
def get\_async\_openai\_client():  
 """Get or create async OpenAI client instance."""  
 try:  
 return AsyncOpenAI(  
 api\_key=OPENAI\_API\_KEY,  
 timeout=httpx.Timeout(60.0, connect=10.0, read=30.0, write=30.0),  
 max\_retries=3  
 )  
 except Exception as e:  
 logger.error(f"Error initializing async OpenAI client: {str(e)}")  
 raise

validate\_api\_keys() Function:

Purpose: Validates the presence of required API keys with descriptive error messages.

def validate\_api\_keys():  
 """Validate API keys and raise descriptive errors."""  
 missing\_keys = []  
 if not OPENAI\_API\_KEY:  
 missing\_keys.append("OpenAI API key")  
 if not GEMINI\_API\_KEY:  
 missing\_keys.append("Gemini API key")  
   
 if missing\_keys:  
 raise ValueError(f"Missing required API keys: {', '.join(missing\_keys)}")

initialize\_api\_clients() Function:

Purpose: Initializes and tests API clients to ensure they are working properly.

async def initialize\_api\_clients():  
 """Initialize API clients and verify they work."""  
 try:  
 # Test OpenAI client  
 client = get\_sync\_openai\_client()  
 models = client.models.list()  
 logger.info("OpenAI client initialized successfully")  
 logger.info(f"Available OpenAI models: {[model.id for model in models.data]}")  
  
 # Test Gemini model  
 model = get\_gemini\_model()  
 logger.info("Gemini model initialized successfully")  
   
 return True  
 except Exception as e:  
 logger.error(f"Failed to initialize API clients: {str(e)}")  
 return False

### Model Types and Configuration

ModelType Class:

Purpose: Defines available AI models and their characteristics.

class ModelType(str, Enum):  
 GPT35 = "gpt-3.5-turbo"  
 GPT4 = "gpt-4"  
 GEMINI\_FLASH = "models/gemini-1.5-flash-8b"  
 GEMINI\_PRO = "models/gemini-1.5-pro-latest"  
   
 @property  
 def description(self) -> str:  
 descriptions = {  
 self.GPT35: "Fast and reliable processing with good accuracy",  
 self.GPT4: "Most accurate processing, better understanding of context and nuances",  
 self.GEMINI\_FLASH: "Gemini 1.5 Flash - Fast and efficient model for text processing",  
 self.GEMINI\_PRO: "Gemini 1.5 Pro - Advanced model with better understanding"  
 }  
 return descriptions.get(self, "Unknown model")  
   
 @property  
 def max\_tokens(self) -> int:  
 limits = {  
 self.GPT35: 30000,  
 self.GPT4: 50000,  
 self.GEMINI\_FLASH: 1\_000\_000,  
 self.GEMINI\_PRO: 2000000  
 }  
 return limits.get(self, 30000)

EditMode Class:

Purpose: Defines text editing modes (fast vs. detailed).

class EditMode(str, Enum):  
 FAST = "fast"  
 DETAILED = "detailed"  
  
class EditRequest(BaseModel):  
 text: str  
 model: str  
 mode: EditMode = EditMode.FAST  
  
class EditResponse(BaseModel):  
 edited\_text: str  
 technical\_explanation: str

### Additional Text Processing Functions

validate\_word\_count() Function:

Purpose: Ensures edited text length stays within acceptable bounds.

def validate\_word\_count(original\_text: str, edited\_text: str, tolerance: int = 50) -> bool:  
 """Validate that edited text word count is within tolerance of original."""  
 original\_count = len(original\_text.split())  
 edited\_count = len(edited\_text.split())  
 return abs(original\_count - edited\_count) <= tolerance

process\_gemini\_edit() Function:

Purpose: Processes text editing requests using the Gemini model.

async def process\_gemini\_edit(text: str, mode: EditMode = EditMode.FAST) -> str:  
 """Process text editing using Gemini model."""  
 try:  
 model = get\_gemini\_model()  
 prompt = EDIT\_PROMPT if mode == EditMode.FAST else EDIT\_PROMPT\_DETAILED  
 response = await model.generate\_content(f"{prompt}  
  
Text: {text}")  
 return response.text  
 except Exception as e:  
 logger.error(f"Gemini editing error: {str(e)}")  
 raise

process\_openai\_edit() Function:

Purpose: Processes text editing requests using OpenAI models.

Purpose: Processes text editing requests using OpenAI models.

async def process\_openai\_edit(text: str, model: str = ModelType.GPT35.value,   
 mode: EditMode = EditMode.FAST) -> str:  
 """Process text editing using OpenAI model."""  
 try:  
 client = get\_async\_openai\_client()  
 prompt = EDIT\_PROMPT if mode == EditMode.FAST else EDIT\_PROMPT\_DETAILED  
 response = await client.chat.completions.create(  
 model=model,  
 messages=[  
 {"role": "system", "content": prompt},  
 {"role": "user", "content": text}  
 ]  
 )  
 return response.choices[0].message.content  
 except Exception as e:  
 logger.error(f"OpenAI editing error: {str(e)}")  
 raise

is\_title() Function:

Purpose: Detects if a text segment is likely a title based on characteristics.

def is\_title(text: str, is\_paragraph\_start: bool = False) -> bool:  
 """Determine if text is likely a title."""  
 # Title characteristics  
 characteristics = [  
 len(text.split()) <= 10, # Short length  
 text.strip().endswith((':', '؛', '.')), # Ends with certain punctuation  
 text.isupper(), # All caps (for English)  
 any(char.isdigit() for char in text[:2]), # Starts with number  
 is\_paragraph\_start # At start of paragraph  
 ]  
 return any(characteristics)

# 6.12 API Endpoints Documentation

## Root Endpoint

Main translation endpoint:

# POST /  
 - Purpose: Main translation endpoint  
 - Request Body:  
 - text: Text to translate  
 - model\_type: "openai" or "gemini"  
 - edit\_mode: "basic" or "detailed"  
 - Response:  
 - translated\_text: Translated content  
 - changes: Change statistics  
 - html\_diff: Visual diff of changes

## Edit Endpoint

Text editing endpoint:

# POST /edit  
 - Purpose: Edit and improve text  
 - Request Body:  
 - text: Text to edit  
 - model\_type: "openai" or "gemini"  
 - edit\_mode: "basic" or "detailed"  
 - Response:  
 - edited\_text: Improved text  
 - changes: Change statistics  
 - html\_diff: Visual diff of changes

## WebSocket Endpoint

Real-time updates endpoint:

# WebSocket /ws  
 - Purpose: Real-time translation updates  
 - Events:  
 - translation\_progress: Progress updates  
 - translation\_complete: Final results  
 - error: Error notifications  
 - Data:  
 - progress: Completion percentage  
 - current\_chunk: Current processing chunk  
 - status: Operation status

## Health Check Endpoint

System health monitoring:

# GET /health  
 - Purpose: System health check  
 - Response:  
 - status: "healthy" or "unhealthy"  
 - api\_status: API connectivity status  
 - model\_status: Model availability  
 - system\_metrics: Performance metrics

# 6.13 Error Handling and Troubleshooting

## Common Error Types

The application handles various types of errors:

### API Errors

• OpenAI API errors (rate limiting, authentication)  
• Gemini API errors  
• Network connectivity issues

### Processing Errors

• Translation errors  
• Memory overflow errors  
• Input validation errors

### System Errors

• Configuration errors  
• Database errors  
• Resource exhaustion

# 6.14 Performance Optimization

## Text Processing Optimization

The application implements several optimization strategies for text processing:

# Text chunking for large inputs  
 async def chunk\_text(text: str, max\_chunk\_size: int = 1000) -> List[str]:  
 """Split text into manageable chunks while preserving context."""  
 chunks = []  
 current\_chunk = []  
 current\_size = 0  
   
 for sentence in text.split('.'):  
 sentence\_size = len(sentence)  
 if current\_size + sentence\_size <= max\_chunk\_size:  
 current\_chunk.append(sentence)  
 current\_size += sentence\_size  
 else:  
 chunks.append('.'.join(current\_chunk))  
 current\_chunk = [sentence]  
 current\_size = sentence\_size  
   
 if current\_chunk:  
 chunks.append('.'.join(current\_chunk))  
   
 return chunks

## Memory Management

Memory optimization strategies include:

# Resource cleanup  
 def cleanup\_resources():  
 gc.collect() # Garbage collection  
   
 # Memory-efficient processing  
 async def process\_large\_text(text: str):  
 chunks = await chunk\_text(text)  
 results = []  
   
 for chunk in chunks:  
 result = await process\_chunk(chunk)  
 results.append(result)  
 await cleanup\_resources() # Clean up after each chunk  
   
 return combine\_results(results)

## API Request Optimization

Strategies for optimizing API requests:

# Efficient API client management  
 @lru\_cache()  
 def get\_openai\_client():  
 """Cached OpenAI client to avoid repeated initialization"""  
 return OpenAI(  
 api\_key=OPENAI\_API\_KEY,  
 timeout=httpx.Timeout(60.0),  
 max\_retries=3  
 )  
   
 # Parallel processing for multiple requests  
 async def process\_multiple\_translations(texts: List[str]):  
 tasks = [translate\_text(text) for text in texts]  
 return await asyncio.gather(\*tasks)

# 6.15 Security Documentation

## API Key Management

Secure API key handling:

# Environment-based key management  
 OPENAI\_API\_KEY = os.getenv('OPENAI\_API\_KEY')  
 GEMINI\_API\_KEY = os.getenv('GEMINI\_API\_KEY')  
   
 def validate\_api\_keys():  
 if not OPENAI\_API\_KEY or not GEMINI\_API\_KEY:  
 raise ValueError("Missing required API keys")

## Rate Limiting

Rate limiting implementation:

# Rate limiting middleware  
 @app.middleware("http")  
 async def rate\_limit\_middleware(request: Request, call\_next):  
 client\_ip = request.client.host  
 if await is\_rate\_limited(client\_ip):  
 raise HTTPException(status\_code=429, detail="Too many requests")  
 return await call\_next(request)

## Input Validation

Input validation and sanitization:

# Input validation  
 def validate\_input(text: str, max\_length: int = 10000) -> bool:  
 if not text or len(text) > max\_length:  
 return False  
 return True  
   
 # Data sanitization  
 def sanitize\_input(text: str) -> str:  
 return text.strip().replace('<script>', '').replace('</script>', '')

# 6.16 Deployment Guide

## Environment Setup

Required environment setup:

# Environment variables  
 OPENAI\_API\_KEY=your\_openai\_key  
 GEMINI\_API\_KEY=your\_gemini\_key  
 DATABASE\_URL=postgresql://user:password@localhost:5432/dbname  
 REDIS\_URL=redis://localhost:6379

## Deployment Steps

Step-by-step deployment process:

# 1. Install dependencies  
 pip install -r requirements.txt  
   
 # 2. Set up environment  
 cp .env.example .env  
 # Edit .env with your configuration  
   
 # 3. Initialize database  
 python manage.py init\_db  
   
 # 4. Start the application  
 uvicorn translation\_bot:app --host 0.0.0.0 --port 8000

## Monitoring Setup

Monitoring and logging configuration:

# Logging configuration  
 logging.basicConfig(  
 level=logging.INFO,  
 format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',  
 handlers=[  
 logging.FileHandler('app.log'),  
 logging.StreamHandler()  
 ]  
 )  
   
 # Health check endpoint  
 @app.get("/health")  
 async def health\_check():  
 return {"status": "healthy"}

# 6.17 User Guide

## Getting Started

Quick start guide:

# Install dependencies  
 pip install -r requirements.txt  
   
 # Set up environment variables  
 cp .env.example .env  
 # Edit .env with your API keys  
   
 # Run the application  
 uvicorn translation\_bot:app --reload

## Common Use Cases

Example usage scenarios:

# 1. Basic Translation  
 curl -X POST "http://localhost:8000/" \  
 -H "Content-Type: application/json" \  
 -d '{"text": "Hello world", "source\_language": "en", "target\_language": "fa"}'  
   
 # 2. Text Editing  
 curl -X POST "http://localhost:8000/edit" \  
 -H "Content-Type: application/json" \  
 -d '{"text": "Hello world", "edit\_type": "improve"}'  
   
 # 3. WebSocket Connection  
 websocat ws://localhost:8000/ws

## Best Practices

Recommended usage patterns:

# 1. Batch Processing  
 async def process\_batch(texts: List[str]):  
 tasks = [translate\_text(text) for text in texts]  
 return await asyncio.gather(\*tasks)  
   
 # 2. Error Handling  
 try:  
 result = await translate\_text(text)  
 except TranslationError as e:  
 logger.error(f"Translation failed: {e}")  
 # Handle error appropriately  
   
 # 3. Resource Management  
 async with AsyncClient() as client:  
 response = await client.post("/", json=data)

# 6.18 Integration Guide

## API Integration

Example API integration:

# Python client example  
 import requests  
   
 def translate\_text(text: str, source\_lang: str, target\_lang: str):  
 response = requests.post(  
 "http://localhost:8000/",  
 json={  
 "text": text,  
 "source\_language": source\_lang,  
 "target\_language": target\_lang  
 }  
 )  
 return response.json()

## WebSocket Integration

Real-time updates integration:

# WebSocket client example  
 import websockets  
   
 async def connect\_websocket():  
 async with websockets.connect('ws://localhost:8000/ws') as websocket:  
 while True:  
 message = await websocket.recv()  
 print(f"Received: {message}")

## Third-party Integrations

Integration with external services:

# OpenAI integration  
 from openai import AsyncOpenAI  
   
 async def openai\_translate(text: str):  
 client = AsyncOpenAI()  
 response = await client.chat.completions.create(  
 model="gpt-3.5-turbo",  
 messages=[  
 {"role": "system", "content": "You are a translator."},  
 {"role": "user", "content": f"Translate: {text}"}  
 ]  
 )  
 return response.choices[0].message.content  
   
 # Gemini integration  
 import google.generativeai as genai  
   
 def gemini\_translate(text: str):  
 model = genai.GenerativeModel('gemini-pro')  
 response = model.generate\_content(f"Translate: {text}")  
 return response.text

## 6.13 Visual Aids and Diagrams

### System Architecture Diagram

|  |  |  |
| --- | --- | --- |
|  | Frontend Web Interface |  |
| FastAPI Backend Server | Translation Service | WebSocket Server |
| OpenAI API | Gemini API | Database/Cache |

↑ API Requests

↑ Real-time Updates

↑ Data Storage

The system architecture consists of three main layers:  
1. Frontend Layer: Web interface for user interaction  
2. Backend Layer: Core translation and processing services  
3. External Services Layer: AI models and data storage

### Data Flow Diagram

|  |
| --- |
| 1. User Input → Frontend Validation |
| 2. Frontend → Backend API Request |
| 3. Backend → AI Model Processing |
| 4. AI Model → Response Formatting |
| 5. Response → User Interface |

↓

Data Flow

↓

Data Flow

↓

Data Flow

↓

Data Flow

The data flow diagram shows the sequence of data processing:  
• Input validation ensures data quality  
• API requests are authenticated and rate-limited  
• AI models process text in chunks  
• Responses are formatted and validated  
• Results are delivered to the user interface

### Error Handling Flowchart

|  |
| --- |
| 1. Error Detection |
| 2. Error Classification |
| 3. Error Handling Strategy |
| 4. Recovery Action |
| 5. User Notification |
| 6. Logging and Monitoring |

↓

Error Handling Flow

↓

Error Handling Flow

↓

Error Handling Flow

↓

Error Handling Flow

↓

Error Handling Flow

The error handling flowchart illustrates the system's error management:  
• Errors are detected through validation and monitoring  
• Errors are classified by type and severity  
• Appropriate handling strategies are selected  
• Recovery actions are executed when possible  
• Users are notified of errors and their status  
• All errors are logged for analysis and improvement

# 6.15 Environment Setup and API Keys

## Environment Variables

The system uses environment variables for configuration and API keys. Create a `.env` file in the root directory with the following variables:

|  |  |  |
| --- | --- | --- |
| Variable Name | Description | Example Value |
| OPENAI\_API\_KEY | Your OpenAI API key for GPT-4 access | sk-... |
| GEMINI\_API\_KEY | Your Google Gemini API key | AIza... |
| DATABASE\_URL | PostgreSQL database connection URL | postgresql://user:pass@localhost:5432/dbname |
| REDIS\_URL | Redis connection URL for caching | redis://localhost:6379/0 |
| LOG\_LEVEL | Logging level (DEBUG, INFO, WARNING, ERROR) | INFO |
| MAX\_TOKENS | Maximum tokens for API responses | 2000 |
| CACHE\_TTL | Cache time-to-live in seconds | 3600 |
| RATE\_LIMIT | API rate limit per minute | 60 |
| ENVIRONMENT | Deployment environment (development/production) | development |

## API Key Setup Instructions

### OpenAI API Key

To obtain an OpenAI API key:

* 1. Visit https://platform.openai.com/
* 2. Create an account or sign in
* 3. Navigate to API Keys section
* 4. Create a new API key
* 5. Copy the key and add it to your .env file
* 6. Never share or commit your API key

### Gemini API Key

To obtain a Gemini API key:

* 1. Visit https://makersuite.google.com/app/apikey
* 2. Sign in with your Google account
* 3. Create a new API key
* 4. Copy the key and add it to your .env file
* 5. Keep your API key secure

## Security Best Practices

* Never commit the .env file to version control
* Use different API keys for development and production
* Regularly rotate API keys
* Set up API key usage monitoring
* Implement rate limiting
* Use environment-specific .env files (.env.development, .env.production)

## Example .env File

Example .env file structure:

# API Keys  
OPENAI\_API\_KEY=sk-your-openai-key-here  
GEMINI\_API\_KEY=AIza-your-gemini-key-here  
  
# Database Configuration  
DATABASE\_URL=postgresql://user:password@localhost:5432/dbname  
REDIS\_URL=redis://localhost:6379/0  
  
# Application Settings  
LOG\_LEVEL=INFO  
MAX\_TOKENS=2000  
CACHE\_TTL=3600  
RATE\_LIMIT=60  
ENVIRONMENT=development

# 6.24 Module Documentation

## changes.py Module

The changes.py module handles version control, change tracking, and text comparison functionality.

### Core Functions

track\_changes() Function:

Purpose: Tracks and records changes between original and modified text.

def track\_changes(original: str, modified: str) -> Dict[str, Any]:  
 """Track changes between original and modified text.  
   
 Args:  
 original (str): Original text  
 modified (str): Modified text  
   
 Returns:  
 Dict containing:  
 - replacements: Number of replaced segments  
 - deletions: Number of deleted segments  
 - insertions: Number of inserted segments  
 - details: List of specific changes  
 """  
 changes = {  
 'replacements': 0,  
 'deletions': 0,  
 'insertions': 0,  
 'details': []  
 }  
 # Implementation details...

generate\_html\_diff() Function:

Purpose: Generates HTML visualization of text differences.

def generate\_html\_diff(original: str, modified: str) -> str:  
 """Generate HTML visualization of differences.  
   
 Args:  
 original (str): Original text  
 modified (str): Modified text  
   
 Returns:  
 str: HTML string showing differences with color coding  
 """  
 # Implementation details...

Version Control Functions:

The module includes comprehensive version control:

class VersionControl:  
 def save\_version(self, text: str, version: str) -> None:  
 """Save a new version of the text."""  
   
 def get\_version(self, version: str) -> str:  
 """Retrieve a specific version."""  
   
 def list\_versions(self) -> List[str]:  
 """List all available versions."""  
   
 def compare\_versions(self, v1: str, v2: str) -> Dict[str, Any]:  
 """Compare two versions and return differences."""

## improvements.py Module

The improvements.py module contains logic for enhancing translation quality.

### Translation Prompts

Specialized prompts for different translation scenarios:

# Base translation prompt  
 TRANSLATION\_PROMPT = """  
 Translate the following Persian text to English:  
 - Maintain formal/informal tone  
 - Preserve cultural context  
 - Keep technical terms accurate  
 - Maintain formatting and structure  
 {text}  
 """  
   
 # Technical translation prompt  
 TECHNICAL\_TRANSLATION\_PROMPT = """  
 Translate the following technical Persian text:  
 - Preserve technical terminology  
 - Maintain academic/technical tone  
 - Keep formatting and citations  
 - Include glossary for key terms  
 {text}  
 """  
   
 # Literary translation prompt  
 LITERARY\_TRANSLATION\_PROMPT = """  
 Translate the following literary Persian text:  
 - Preserve literary style and tone  
 - Maintain metaphors and cultural references  
 - Keep poetic elements where applicable  
 - Preserve author's voice  
 {text}  
 """

### Enhancement Functions

Functions for improving translation quality:

analyze\_context() Function:

Purpose: Analyzes text context for better translation.

def analyze\_context(text: str) -> Dict[str, Any]:  
 """Analyze text context for better translation.  
   
 Args:  
 text (str): Input text  
   
 Returns:  
 Dict containing:  
 - domain: Technical domain (e.g., medical, legal)  
 - formality: Text formality level  
 - style: Writing style  
 - key\_terms: Important terminology  
 """  
 # Implementation details...

enhance\_translation() Function:

Purpose: Applies quality improvements to translations.

async def enhance\_translation(  
 text: str,  
 context: Dict[str, Any],  
 model\_type: str = "gpt-4"  
 ) -> str:  
 """Enhance translation quality.  
   
 Args:  
 text (str): Translated text  
 context (dict): Context information  
 model\_type (str): AI model to use  
   
 Returns:  
 str: Enhanced translation  
 """  
 # Apply Persian-specific rules  
 text = apply\_persian\_rules(text)  
   
 # Generate context-aware prompt  
 prompt = generate\_context\_prompt(text, context)  
   
 # Get model suggestions  
 suggestions = await get\_model\_suggestions(prompt)  
   
 # Apply improvements  
 improved\_text = apply\_improvements(text, suggestions)  
   
 # Validate quality  
 quality\_score = calculate\_quality\_score(improved\_text)  
   
 if quality\_score < QUALITY\_THRESHOLD:  
 improved\_text = await apply\_additional\_improvements(improved\_text)  
   
 return improved\_text

### Persian Language Rules

Specialized rules for Persian language processing:

class PersianRules:  
 @staticmethod  
 def fix\_spacing(text: str) -> str:  
 """Fix Persian text spacing issues."""  
   
 @staticmethod  
 def normalize\_characters(text: str) -> str:  
 """Normalize Persian characters."""  
   
 @staticmethod  
 def fix\_punctuation(text: str) -> str:  
 """Fix Persian punctuation."""  
   
 @staticmethod  
 def handle\_numbers(text: str) -> str:  
 """Handle Persian numbers and dates."""

### Quality Metrics

Functions for measuring translation quality:

def calculate\_quality\_score(text: str) -> float:  
 """Calculate translation quality score.  
   
 Metrics include:  
 - Grammar correctness  
 - Terminology accuracy  
 - Style consistency  
 - Cultural appropriateness  
 - Technical accuracy  
 """  
   
 def validate\_technical\_terms(text: str, domain: str) -> bool:  
 """Validate technical terminology."""  
   
 def check\_style\_consistency(text: str, style: str) -> bool:  
 """Check for consistent writing style."""

# 6.25 Project Files Documentation

## Core Application Files

### translation\_bot.py

Main application file that handles the core translation functionality.

# Key components:  
 - FastAPI application setup  
 - WebSocket connections  
 - Translation endpoints  
 - AI model integration  
 - Error handling  
 - Rate limiting  
 - Caching system

### changes.py

Handles version control and change tracking for translations.

# Key features:  
 - Version history management  
 - Change tracking  
 - HTML diff generation  
 - Version comparison  
 - Change statistics

### improvements.py

Enhances translation quality and handles Persian-specific rules.

# Key features:  
 - Context analysis  
 - Quality enhancement  
 - Persian language rules  
 - Technical term handling  
 - Style consistency

## Test Files

### test\_changes.py

Tests for the changes module functionality.

# Test cases:  
 - Version tracking  
 - Change detection  
 - HTML diff generation  
 - Version comparison  
 - Change statistics

### test\_connections.py

Tests API connections and external service integration.

# Test cases:  
 - OpenAI API connection  
 - Gemini API connection  
 - Database connection  
 - Redis connection  
 - WebSocket connection

### test\_gemini.py

Specific tests for Gemini model integration.

# Test cases:  
 - API key validation  
 - Model response  
 - Error handling  
 - Rate limiting  
 - Response formatting

## Configuration Files

### .env

Environment configuration file for API keys and settings.

# Configuration items:  
 - API keys  
 - Database URLs  
 - Cache settings  
 - Log levels  
 - Rate limits  
 - Environment type

### requirements.txt

Python package dependencies for the project.

# Key dependencies:  
 - fastapi  
 - openai  
 - google-generativeai  
 - python-dotenv  
 - asyncpg  
 - redis  
 - python-docx

## HTML Templates

### index.html

Main web interface for the translation system.

# Features:  
 - Translation interface  
 - Real-time updates  
 - Error display  
 - Version history  
 - Change visualization

## Static Files

### CSS Files

Styling for the web interface.

# Styles:  
 - Main layout  
 - Translation interface  
 - Error messages  
 - Version history  
 - Change highlights

### JavaScript Files

Client-side functionality for the web interface.

# Features:  
 - WebSocket connection  
 - Real-time updates  
 - Form handling  
 - Error handling  
 - Version management

## File Relationships

How the files work together:

* **translation\_bot.py:** Main application that coordinates all components
* **changes.py:** Tracks and manages translation versions
* **improvements.py:** Enhances translation quality
* **test\_\*.py:** Ensures functionality of each component
* **.env:** Provides configuration for all components
* **requirements.txt:** Lists dependencies for all components
* **HTML/CSS/JS:** Provides user interface for the system