PersianAI: Persian-English Translation System

Author:

**Hamideh Shahrabi**

April 13, 2025

Table of Contents

[Executive Summary 8](#_Toc195487403)

[Introduction 9](#_Toc195487404)

[Project Goals 9](#_Toc195487405)

[Overview 10](#_Toc195487406)

[Component Interaction 10](#_Toc195487407)

[Implementation Details 11](#_Toc195487408)

[Core Components 11](#_Toc195487409)

[Technology Stack 11](#_Toc195487410)

[System Architecture Diagram 12](#_Toc195487411)

[Data Flow Diagram 13](#_Toc195487412)

[Error Handling Flowchart 13](#_Toc195487413)

[Steps Explained: 13](#_Toc195487414)

[1. Error Detection 13](#_Toc195487415)

[2. Error Classification 13](#_Toc195487416)

[3. Handling Strategy Selection 14](#_Toc195487417)

[4. Recovery Action Execution 14](#_Toc195487418)

[5. User Notification 14](#_Toc195487419)

[6. Logging and Monitoring 14](#_Toc195487420)

[Environment Setup and API Keys 15](#_Toc195487421)

[Environment Variables 15](#_Toc195487422)

[API Key Setup Instructions 16](#_Toc195487423)

[OpenAI API Key 16](#_Toc195487424)

[Gemini API Key 17](#_Toc195487425)

[Security Best Practices 17](#_Toc195487426)

[Example .env File 17](#_Toc195487427)

[PersianAI Web Interface Overview 18](#_Toc195487428)

[Persian Text Editing Section 20](#_Toc195487429)

[1. Edit Mode Selector 20](#_Toc195487430)

[2. AI Model Selector (Editing) 20](#_Toc195487431)

[3. Text Input Box 21](#_Toc195487432)

[4. Word Count Tracker & Progress Feedback 21](#_Toc195487433)

[5. Edit Text Button 21](#_Toc195487434)

[Word-by-Word Editing and Visual Diff View 21](#_Toc195487435)

[What It Shows: 22](#_Toc195487436)

[Why It’s Helpful: 22](#_Toc195487437)

[What It Does: 24](#_Toc195487438)

[Summary of Changes: 24](#_Toc195487439)

[English Translation Section 25](#_Toc195487440)

[AI Model Selector (Translation) 25](#_Toc195487441)

[Translate Button 25](#_Toc195487442)

[Sample English Translation Output 26](#_Toc195487443)

[What This Shows: 26](#_Toc195487444)

[Why This Matters 27](#_Toc195487445)

[Testing and Validation 27](#_Toc195487446)

[Testing Strategy 27](#_Toc195487447)

[User Acceptance Testing (UAT) 28](#_Toc195487448)

[Validation Methods 28](#_Toc195487449)

[Code Documentation 28](#_Toc195487450)

[Overview of Key Files 28](#_Toc195487451)

[translation\_bot.py: Application Controller for Editing and Translation Workflow 29](#_Toc195487452)

[Purpose: 29](#_Toc195487453)

[Why It Was Created: 30](#_Toc195487454)

[Key Requirements It Fulfills: 30](#_Toc195487455)

[Code Snippet: FastAPI Application Setup 30](#_Toc195487456)

[Code Snippet: CORS Configuration 30](#_Toc195487457)

[Code Snippet: Middleware to Track Request Time 31](#_Toc195487458)

[changes.py: Word-Level Change Tracking and Visualization 31](#_Toc195487459)

[Purpose: 31](#_Toc195487460)

[Why It Was Created: 32](#_Toc195487461)

[Key Requirements It Fulfills: 32](#_Toc195487462)

[Code Snippet: track\_changes() Function 32](#_Toc195487463)

[Explanation 34](#_Toc195487464)

[limitations.py: Defining Editing and Translation Boundaries 34](#_Toc195487465)

[Purpose: 34](#_Toc195487466)

[Why It Was Created: 35](#_Toc195487467)

[Key Requirements It Fulfills: 35](#_Toc195487468)

[Code Snippet: (Hypothetical structure — adjust if needed) 35](#_Toc195487469)

[Explanation 36](#_Toc195487470)

[test\_changes.py: Unit Testing for Change Tracking System 36](#_Toc195487471)

[Purpose: 36](#_Toc195487472)

[Why It Was Created: 37](#_Toc195487473)

[Key Requirements It Fulfills: 37](#_Toc195487474)

[Code Snippet: Sample Unit Test from test\_changes.py 37](#_Toc195487475)

[Explanation 38](#_Toc195487476)

[test\_connections.py: Testing API Connectivity and Reliability 38](#_Toc195487477)

[Purpose: 38](#_Toc195487478)

[Why It Was Created: 39](#_Toc195487479)

[Key Requirements It Fulfills: 39](#_Toc195487480)

[Code Snippet: Sample Connection Test 39](#_Toc195487481)

[Explanation 40](#_Toc195487482)

[test\_gemini.py: Validating Gemini-Based Editing and Translation 40](#_Toc195487483)

[Purpose: 40](#_Toc195487484)

[Why It Was Created: 40](#_Toc195487485)

[Key Requirements It Fulfills: 41](#_Toc195487486)

[Code Snippet: Sample Tests 41](#_Toc195487487)

[Explanation: 42](#_Toc195487488)

[📘 Summary: Code Documentation 42](#_Toc195487489)

[Key Modules Recap 42](#_Toc195487490)

[Project Files Documentation 43](#_Toc195487491)

[Core Application Files 43](#_Toc195487492)

[translation\_bot.py 43](#_Toc195487493)

[changes.py 43](#_Toc195487494)

[improvements.py 44](#_Toc195487495)

[Test Files 44](#_Toc195487496)

[test\_changes.py 44](#_Toc195487497)

[test\_connections.py 44](#_Toc195487498)

[test\_gemini.py 44](#_Toc195487499)

[Configuration Files 44](#_Toc195487500)

[.env 44](#_Toc195487501)

[requirements.txt 45](#_Toc195487502)

[HTML Templates 45](#_Toc195487503)

[index.html 45](#_Toc195487504)

[Static Files 45](#_Toc195487505)

[CSS Files 45](#_Toc195487506)

[JavaScript Files 45](#_Toc195487507)

[File Relationships 46](#_Toc195487508)

[Function Documentation 46](#_Toc195487509)

[Translation Bot Functions 46](#_Toc195487510)

[Core Functions 46](#_Toc195487511)

[Translation Functions 47](#_Toc195487512)

[Text Processing Functions 48](#_Toc195487513)

[Changes Module Functions 49](#_Toc195487514)

[Improvements Module 50](#_Toc195487515)

[Test Functions 51](#_Toc195487516)

[Test Changes Functions 51](#_Toc195487517)

[Test Connections Functions 52](#_Toc195487518)

[API Client Management Functions 52](#_Toc195487519)

[Model Types and Configuration 54](#_Toc195487520)

[Additional Text Processing Functions 55](#_Toc195487521)

[Endpoints Documentation 56](#_Toc195487522)

[Root Endpoint 56](#_Toc195487523)

[Edit Endpoint 57](#_Toc195487524)

[WebSocket Endpoint 57](#_Toc195487525)

[Health Check Endpoint 57](#_Toc195487526)

[6.13 Error Handling and Troubleshooting 58](#_Toc195487527)

[Common Error Types 58](#_Toc195487528)

[API Errors 58](#_Toc195487529)

[Processing Errors 58](#_Toc195487530)

[System Errors 58](#_Toc195487531)

[6.14 Performance Optimization 58](#_Toc195487532)

[Text Processing Optimization 58](#_Toc195487533)

[Memory Management 59](#_Toc195487534)

[API Request Optimization 59](#_Toc195487535)

[6.15 Security Documentation 59](#_Toc195487536)

[API Key Management 59](#_Toc195487537)

[Rate Limiting 60](#_Toc195487538)

[Input Validation 60](#_Toc195487539)

[6.16 Deployment Guide 60](#_Toc195487540)

[Environment Setup 60](#_Toc195487541)

[Deployment Steps 60](#_Toc195487542)

[Monitoring Setup 61](#_Toc195487543)

[6.17 User Guide 61](#_Toc195487544)

[Getting Started 61](#_Toc195487545)

[Common Use Cases 62](#_Toc195487546)

[Best Practices 62](#_Toc195487547)

[6.18 Integration Guide 62](#_Toc195487548)

[API Integration 62](#_Toc195487549)

[Edit Request via REST API (Python) 63](#_Toc195487550)

[WebSocket Integration 63](#_Toc195487551)

[Third-party Integrations 64](#_Toc195487552)

[OpenAI Integration (Async Client) 64](#_Toc195487553)

[Gemini Integration (Google Generative AI) 64](#_Toc195487554)

[import google.generativeai as genai 65](#_Toc195487555)

[def gemini\_translate(text: str): 65](#_Toc195487556)

[model = genai.GenerativeModel("gemini-pro") 65](#_Toc195487557)

[response = model.generate\_content(f"Translate this Persian text into English: {text}") 65](#_Toc195487558)

[return response.text 65](#_Toc195487559)

[6.13 Visual Aids and Diagrams 65](#_Toc195487560)

[System Architecture Diagram 65](#_Toc195487561)

[Data Flow Diagram 66](#_Toc195487562)

[Error Handling Flowchart 67](#_Toc195487563)

[Steps Explained: 67](#_Toc195487564)

[1. Error Detection 67](#_Toc195487565)

[2. Error Classification 67](#_Toc195487566)

[3. Handling Strategy Selection 67](#_Toc195487567)

[4. Recovery Action Execution 68](#_Toc195487568)

[5. User Notification 68](#_Toc195487569)

[6. Logging and Monitoring 68](#_Toc195487570)

[6.15 Environment Setup and API Keys 69](#_Toc195487571)

[Environment Variables 69](#_Toc195487572)

[API Key Setup Instructions 70](#_Toc195487573)

[OpenAI API Key 70](#_Toc195487574)

[Gemini API Key 71](#_Toc195487575)

[Security Best Practices 71](#_Toc195487576)

[Example .env File 71](#_Toc195487577)

[Module Documentation 72](#_Toc195487578)

[translation\_bot.py Module 72](#_Toc195487579)

[Core Functions 72](#_Toc195487580)

[Purpose 72](#_Toc195487581)

[add\_process\_time\_header() Function: 73](#_Toc195487582)

[Translation Endpoint Functions 73](#_Toc195487583)

[Supporting Utilities 74](#_Toc195487584)

[Key Features: 75](#_Toc195487585)

[changes.py Module 75](#_Toc195487586)

[Core Functions 75](#_Toc195487587)

[improvements.py Module 76](#_Toc195487588)

[Translation Prompts 77](#_Toc195487589)

[Enhancement Functions 77](#_Toc195487590)

[Persian Language Rules 78](#_Toc195487591)

[Quality Metrics 79](#_Toc195487592)

# Executive Summary

**PersianAI** is an advanced AI-powered assistant designed to automate and enhance the editing and translation of Persian (Farsi) text into English — with a particular focus on **minimizing human intervention** in editorial workflows. Developed specifically for multi-author publishing scenarios such as coaching and psychology books, PersianAI uses intelligent language processing to deliver **professional-quality results** with **greater speed, consistency, and accuracy** than traditional manual methods.

While the system includes high-performance translation capabilities, its **primary strength lies in the AI-powered editing of Persian content**. The platform is designed to handle complex grammatical structures, normalize tone and style, and make Farsi content publication-ready — all before translating it into fluent English.

## Key features include:

• ✏️ AI-enhanced Persian editing with grammar, tone, and formatting fixes

• 🔁 Real-time translation with WebSocket streaming

• 🧠 Context-aware prompt engineering using OpenAI and Gemini models

• 📊 Word-level change tracking and version control

• 🔌 Comprehensive API for seamless integration into editorial systems

## Explore the complete system on GitHub

All of the features mentioned above — including the Persian text editor, model selection UI, word-by-word diff view, summary of changes, and English translation output — are implemented and available in our open-source repository.

You can view the interface, test the models, and run the full editing and translation pipeline by visiting:

<https://github.com/hamideshahrabi/persian-translator>

# Introduction

The PersianAI project was created to solve a common bottleneck in professional translation: the **need to clean, unify, and standardize Persian content before translation** — especially in multi-author texts. Translating raw or inconsistent content can lead to poor results, even with powerful language models. That’s why PersianAI focuses first on **automated editing**, applying AI-driven improvements to grammar, tone, structure, and fluency.

This approach drastically reduces the need for human editing, saving time and energy while still meeting professional publishing standards. Once edited, the content is passed through OpenAI or Gemini models for high-quality, context-preserving English translation.

# Project Goals

• ✅ Automate the **professional editing** of Persian content written by multiple authors, specifically in the **psychology and coaching domain**

• ✅ Ensure **style consistency and tone unification** across chapters authored by different contributors

• ✅ Enable flexible use of **multiple AI models** (e.g., OpenAI GPT-4 and Gemini Pro) to suit various content styles and budget considerations

• ✅ Significantly **reduce the cost and manual effort** typically required in editing and translation workflows

• ✅ Design custom **prompt templates and editing scenarios** to prepare models for psychology-specific language and coaching tone

• ✅ Maintain the **original voice, intent, and emotional tone** of authors during both editing and translation

• ✅ Provide **real-time feedback** through WebSocket for immediate visibility of edits and translations

• ✅ Allow easy **integration into publishing and editorial platforms** to streamline multi-author project workflows

# Overview

The PersianAI system is built using a modular, microservices-based architecture to ensure scalability, performance, and maintainability — especially when working with large-scale, multi-author projects in the domain of psychology and coaching publications. Each component is responsible for a distinct layer of the workflow, allowing for flexibility, parallel processing, and easy debugging.

Key architectural components include:

• **FastAPI web server** for handling HTTP and WebSocket requests

• **Translation processing service** to run LLM-based editing and translation tasks

• **Change tracking system** that monitors and visualizes word-level edits across document versions

• **WebSocket communication layer** for real-time interaction and streaming responses

• **External API integrations** with OpenAI (GPT-3.5/4) and Google Gemini (Pro/Flash)

## Component Interaction

All components communicate through **well-defined APIs** and asynchronous message handling. This architecture provides:

• ✅ **Loose coupling** — each module operates independently for easy maintenance

• ✅ **High scalability** — supports concurrent editing and translation of multiple chapters or documents

• ✅ **Optimized performance** — resource-efficient processing, ideal for long documents

• ✅ **Robust error handling** — isolates failures and supports retry logic across modules

# Implementation Details

## Core Components

The primary implementation consists of five interconnected modules:

• **FastAPI application server**: Central controller that defines routes and manages requests

• **Translation processing engine**: Executes model-based editing and translation using chunking and parallelism

• **Change tracking system**: Calculates and visualizes edits between input, enhanced, and translated outputs

• **WebSocket server**: Streams real-time feedback to users during long-running operations

• **API integration layer**: Manages secure communication with OpenAI and Gemini APIs, with dynamic model selection and fallback

This separation of concerns allows PersianAI to support multiple documents, models, and users concurrently — essential in collaborative publishing environments.

## Technology Stack

PersianAI is built with a modern, cloud-ready Python stack:

|  |  |
| --- | --- |
| Layer | Technology |
| Programming Lang | Python 3.9+ |
| Web Framework | FastAPI |
| AI/LLMs | OpenAI GPT-3.5 / GPT-4, Gemini |
| Realtime/Socket | WebSocket (via FastAPI) |
| Data Handling | Redis (for caching and state) |

The system is environment-configurable using .env files, making it easy to manage API keys, rate limits, and runtime behaviors across staging and production.

## System Architecture Diagram

A diagram of a computer server

AI-generated content may be incorrect.

## Data Flow Diagram

1. **User Input** → Sent from the frontend (e.g., text to edit or translate)

2. **FastAPI Backend** → Handles routing, validation, and preprocessing

3. **LLM Services** → OpenAI or Gemini performs editing and/or translation

4. **Postprocessing** → Changes are tracked, visual diff generated

5. **Frontend Output** → Enhanced English output with change logs

A white background with black text

AI-generated content may be incorrect.

## Error Handling Flowchart

The error-handling flowchart outlines how PersianAI detects, manages, and recovers from failures across its system. It is designed to ensure reliability, user transparency, and minimal disruption in production environments.

### Steps Explained:

### 1. Error Detection

Errors are automatically captured during request processing, API communication, or model execution. Examples include invalid user input, model timeouts, or authentication failures.

### 2. Error Classification

Each error is classified into one of the following categories:

• **API Error**: Issues with OpenAI or Gemini (e.g., rate limits, downtime)

• **Input Error**: Malformed requests or unsupported content (e.g., sensitive keywords)

• **Internal Error**: Server exceptions, timeouts, or memory issues

### 3. Handling Strategy Selection

Depending on the error type and severity, the system decides how to respond:

• Retry the request

• Switch to a fallback model

• Skip processing the problematic input

• Notify the user of the issue

### 4. Recovery Action Execution

The system applies the selected action:

• Retries API calls with exponential backoff

• Switches models (e.g., from GPT-4 to Gemini Pro)

• Gracefully skips specific content blocks

• Catches the error and returns a safe fallback response

### 5. User Notification

Users are informed of the error via:

• WebSocket messages (for real-time workflows)

• JSON error responses in REST API

### 6. Logging and Monitoring

All errors are logged with full context (e.g., user request, error trace) for debugging and monitoring. Logs are stored securely and can be reviewed for performance tuning or incident response.

A computer program code with black text

AI-generated content may be incorrect.

# Environment Setup and API Keys

## Environment Variables

PersianAI relies on environment variables to securely manage configuration settings and API keys. These are defined in a .env file located in the root directory.

To ensure flexibility, security, and ease of deployment, PersianAI uses environment variables defined in a .env file. These variables are categorized as follows:

• **API Keys**: Provide secure access to OpenAI and Gemini services (OPENAI\_API\_KEY, GEMINI\_API\_KEY).

• **Database & Caching**: Define backend connection details for PostgreSQL and Redis (DATABASE\_URL, REDIS\_URL).

• **Application Behavior**: Configure runtime behavior, such as logging level, response limits, and API rate control (LOG\_LEVEL, MAX\_TOKENS, CACHE\_TTL, RATE\_LIMIT).

• **Environment Control**: Specifies the deployment context (ENVIRONMENT), allowing different behavior in development vs. production.

|  |  |  |
| --- | --- | --- |
| 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 use OpenAI’s models (GPT-3.5, GPT-4), follow these steps to obtain your API key:

1. Go to [OpenAI’s platform dashboard](https://platform.openai.com/).

2. Sign in or create a new account.

3. Navigate to the **API Keys** section in your profile.

4. Click **“Create new secret key”** and copy the generated key.

5. Paste the key into your .env file as the value for OPENAI\_API\_KEY.

6. **Important**: Do **not** share or commit this key to version control.

### Gemini API Key

To access Google’s Gemini Pro or Flash models:

1. Visit [Google MakerSuite](https://makersuite.google.com/app/apikey).

2. Log in with your Google account.

3. Click **“Create API key”** to generate a new one.

4. Copy the key and add it to your .env file as GEMINI\_API\_KEY.

5. Keep the key confidential and secured at all times.

## Security Best Practices

**Never commit** .env files to version control (e.g., GitHub).

• **Use separate API keys** for development and production environments.

• **Rotate keys regularly** to reduce risk from accidental leaks.

• **Monitor API usage** to detect unusual activity or quota overages.

• **Implement rate limiting** in your application to avoid abuse or bans.

• Maintain multiple .env files (e.g., .env.development, .env.production) for different deployment scenarios.

## 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

# PersianAI Web Interface Overview

PersianAI provides a streamlined, intuitive web interface that empowers authors, editors, and content teams to edit and translate Persian text with professional-level quality and precision. The interface is split into two primary sections: **Persian Text Editing** and **English Translation** — each equipped with smart controls for AI model selection and workflow customization.

A screenshot of a computer

AI-generated content may be incorrect.

## 

## Persian Text Editing Section

### 1. Edit Mode Selector

Choose between two levels of editing:

• **Fast Edit (Grammar & Spelling)**

Ideal for quick surface-level fixes like typos, punctuation, and spelling.

• **Detailed Edit**

Performs in-depth editing including grammar, sentence structure, clarity, tone, and fluency.

A screenshot of a computer

AI-generated content may be incorrect.

### 2. AI Model Selector (Editing)

Users can select from multiple AI models to match their editing goals:

• **Gemini Flash** – Fast & efficient for basic corrections

• **Gemini Pro** – Advanced model for nuanced, context-aware editing

• **GPT-3.5 Turbo** – Fast and cost-efficient

• **GPT-4** – Highest-quality output for final drafts

A screenshot of a phone

AI-generated content may be incorrect.

### 3. Text Input Box

Paste or type your raw Persian content here. PersianAI supports long-form documents and automatically manages large text through chunking.

### 4. Word Count Tracker & Progress Feedback

Displays:

• Live word count

• Goal reminders (e.g., “Add 2400 more words to reach the minimum”) — useful for book chapter limits or publishing thresholds.

A green text on a white background

AI-generated content may be incorrect.

### 5. Edit Text Button

After selecting the edit mode and model, click this to submit the text for processing.

## Word-by-Word Editing and Visual Diff View

In addition to the original vs. edited text layout, PersianAI also offers a **combined view** that displays **inline, word-by-word changes** — giving editors a clear, color-coded visualization of every AI-generated modification.

### What It Shows:

•  **Green text**: Words or phrases that were **added** by the AI during editing.

•  **Red text with strikethroughs**: Words or phrases that were **removed**.

• **Gray arrows (→)**: Show how text was **transformed** or replaced.

### Why It’s Helpful:

• **Editorial Transparency**: Editors can clearly see what the original content was, and exactly how the AI changed it — no guessing or blind trust required.

• **Faster Review Process**: Instead of re-reading entire documents, editors can focus only on the highlighted changes.

• **Quality Control**: Enables detailed review of AI behavior to ensure tone, meaning, and style are preserved — crucial in coaching, psychological, or multi-author texts.

• **Educational Feedback**: Authors can learn from the edits, understanding where clarity or grammar was improved, and why.

• **Supports Collaboration**: This view can be shared with stakeholders to demonstrate how content is evolving, and to align on tone and accuracy.

📝 *This feature ensures PersianAI isn’t a “black box” — it’s a transparent partner that shows exactly how it improved the content.*

A screenshot of a computer

AI-generated content may be incorrect.

### What It Does:

• Tracks **every single word** the AI edits:

✅ Insertions

✅ Replacements

✅ Deletions

• Displays a **color-coded comparison** (HTML diff view)

• Helps editors and stakeholders **review changes line by line**

• Includes **change statistics** and version history

## Summary of Changes:

One of the key **optional features** available in PersianAI is the **Summary of Changes**, which provides a clear, numerical overview of all AI-generated edits.

The following summary was automatically created for the **sample Persian text** shown earlier (*Original vs Edited Text* and *Combined View*).

A close-up of a white background

AI-generated content may be incorrect.

**Breakdown for This Sample:**

• **Original Word Count**: 2462

• **Edited Word Count**: 2210

• **Word Count Difference**: -252

• **Total Edits Detected**: 463

• 🔁 **Replacements**: 383

• ❌ **Deletions**: 20

• ✅ **Insertions**: 60

✅ *Model used: gemini-1.5-pro-latest (fast editing mode)*

## English Translation Section

### AI Model Selector (Translation)

Choose your preferred AI model to convert Persian text into English. Options include:

• **Gemini Flash** – Fast, lightweight for live previews

• **Gemini Pro** – Context-rich, ideal for coaching/psychology tone

• **GPT-3.5 Turbo** – Balanced, general-purpose

• **GPT-4** – Highly accurate, ideal for final English drafts

### Translate Button

Click to receive a high-quality English translation. PersianAI ensures context, tone, and author intent are preserved across languages.

### Sample English Translation Output

After editing the Persian text and reviewing the AI-generated changes, PersianAI also supports **high-quality English translation** using the selected AI model.

The screenshot below shows the **final translated version** of the sample Persian content used in previous sections:

A screenshot of a text message

AI-generated content may be incorrect.

### What This Shows:

• The AI preserved the **original meaning and tone** of the text.

• Emotions, age-specific context, and cultural nuances were translated **fluently and naturally**.

• The voice of the speaker — a coach working with a teenage client — is maintained across languages.

### Why This Matters

This translation feature is especially helpful for:

• 📚 **Publishing bilingual books or coaching materials**

• 🌍 **Reaching non-Persian-speaking audiences**

• 🧠 **Maintaining emotional and cultural accuracy**

• ✍️ **Saving time by automating high-quality first drafts**

PersianAI ensures that authors and editors can produce fluent, meaningful English content without needing separate translators — while preserving the intent, voice, and rhythm of the original Persian text.

# Testing and Validation

## Testing Strategy

To ensure reliability, accuracy, and real-world applicability, PersianAI was rigorously tested across multiple levels:

• 🧪 **Unit Tests**: Covered core logic such as text chunking, model selection, and change tracking

• 🔗 **Integration Tests**: Validated the full API pipeline, including FastAPI endpoints and model interactions

• ⚙️ **Performance Testing**: Ensured the system handled long-form content efficiently in real-time scenarios

• 🔐 **Security Testing**: Included API key validation, rate limiting, and request error handling

• 👥 **User Acceptance Testing (UAT)**: Conducted weekly demo sessions with project stakeholders to evaluate editing accuracy, efficiency, and content alignment

## User Acceptance Testing (UAT)

Weekly demo sessions were held with stakeholders involved in the multi-author psychology coaching book. During these sessions, PersianAI’s model outputs were reviewed collaboratively to assess how accurately the system edited and translated complex content. The team measured improvements in editing efficiency, evaluated the preservation of tone and author intent, and provided qualitative feedback.

These insights were directly used to fine-tune prompt templates, model selection strategies, and editing logic — ensuring the system evolved in alignment with publishing standards.

## Validation Methods

The final output quality was assessed using a combination of automated and human-centered methods:

• ✅ **Automated quality scoring** based on structure, tone, and grammar

• ✅ **Expert review** of edited and translated content by native Persian-speaking professionals

• ✅ **User feedback analysis** from internal testing and review sessions

• ✅ **A/B testing** between different LLMs (e.g., OpenAI vs Gemini) to optimize results per content type

# Code Documentation

## Overview of Key Files

This section provides a high-level summary of the main Python modules that make up the PersianAI system. Each file plays a specific role in the editing and translation workflow.

|  |  |
| --- | --- |
| File Name | Purpose |
| translation\_bot.py | Central orchestrator. Hosts the FastAPI app, manages requests, and coordinates the flow from Persian text editing to English translation. Supports REST and WebSocket communication. |
| improvements.py | Core editing engine. Uses prompt engineering and Persian-specific rules to refine grammar, structure, and tone before translation. |
| changes.py | Handles version comparison. Calculates word-level diffs between original and improved text, enabling audit trails and HTML-based visualizations. |
| limitations.py | Defines the boundaries and constraints of model behavior. Used to manage special content cases, avoid hallucinations, and enforce system-level editing limits. |
| test\_changes.py | Unit tests for the change tracking system. Validates replacement, insertion, deletion detection and diff generation. |
| test\_connections.py | Tests API key connectivity and interaction with external services (OpenAI, Gemini). Checks for timeout, rate-limiting, and auth errors. |
| test\_gemini.py | Validates Gemini-specific translation flow. Tests response structure, model handling, and fallback mechanisms. |
| .env.example | Sample configuration for environment variables. Provides structure for API keys and runtime settings. |
| requirements.txt | Lists all Python package dependencies for the system (FastAPI, OpenAI SDK, etc.). Enables quick setup and deployment. |

## translation\_bot.py: Application Controller for Editing and Translation Workflow

### Purpose:

This is the **main application file** that runs the PersianAI backend. It initializes the FastAPI server, sets up API endpoints, configures middleware, and defines the application’s request-handling logic. It acts as the orchestrator that receives Persian text from the user, routes it through the editing pipeline, and finally calls the translation logic.

### Why It Was Created:

• To serve as a single entry point for all system interactions

• To provide both REST and real-time (WebSocket) support for the editing/translation workflow

• To handle model routing logic, input preprocessing, and return structured responses

### Key Requirements It Fulfills:

• Modular routing of requests to appropriate models (GPT or Gemini)

• Support for real-time and batch processing

• Secure, scalable, and configurable web backend

• Integration with other system modules (improvements.py, changes.py)

### Code Snippet: FastAPI Application Setup

app = FastAPI(

title="PersianAI Translation Service",

description="Professional Persian to English translation service",

version="1.0.0"

)

**This block initializes the FastAPI app**, giving the API a title, description, and version metadata for documentation.

### Code Snippet: CORS Configuration

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_credentials=True,

allow\_methods=["\*"],

allow\_headers=["\*"],

)

This configures **Cross-Origin Resource Sharing (CORS)** so that frontend clients (e.g., React, Streamlit) can access the backend without browser errors.

### Code Snippet: Middleware to Track Request Time

@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

This middleware **adds a header to every response** showing how long it took to process the request. It’s useful for debugging, optimization, and system monitoring.

## changes.py: Word-Level Change Tracking and Visualization

### Purpose:

This file handles **version comparison** between original and modified Persian text. It detects **insertions**, **deletions**, and **replacements** at the word level, and formats the output for use in visual diff tools or audit logs.

This system is crucial for **transparency and validation**, especially in editorial workflows involving multiple authors and automated edits.

### Why It Was Created:

• To give editors and stakeholders visibility into what AI changed

• To support **auditability and rollback** in sensitive content (e.g., coaching and psychology)

• To enable **data-driven quality control** and editorial decision-making

### Key Requirements It Fulfills:

• 🧾 Tracks every word-level change (inserts, deletes, replaces)

• 📊 Generates detailed diff statistics

• 🌐 Prepares results for visual presentation (e.g., HTML diff viewer)

• 🕒 Enables version comparison, history tracking, and rollback features

### Code Snippet: track\_changes() Function

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 for comparison

matcher = SequenceMatcher(None, orig\_words, mod\_words)

# Prepare structure to collect change stats

changes = {

'replacements': 0,

'deletions': 0,

'insertions': 0,

'details': []

}

# Analyze edit operations

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

### Explanation

• split\_into\_words() is a custom function that tokenizes the text while keeping spaces and formatting intact

• SequenceMatcher (from Python’s difflib) calculates the most efficient way to turn one list of words into another

• The function loops through each operation (replace, delete, insert) and:

• Counts the operation

• Appends a detailed entry to changes['details']

• The final result is a full dictionary describing **how the AI transformed the input**, including full audit logs for every word change

## limitations.py: Defining Editing and Translation Boundaries

### Purpose:

This file defines and enforces the **limitations and constraints** that guide the AI editing and translation process. It includes logic to:

• Prevent over-editing or hallucinations

• Flag sensitive or unsupported content

• Apply rules that align with the project’s tone and publishing standards

It plays an important **quality assurance and boundary-setting role** in the pipeline.

### Why It Was Created:

• To **control AI behavior** and keep it aligned with real-world use cases (e.g., multi-author coaching texts)

• To avoid unexpected or inappropriate model outputs

• To ensure the system does not override user intent or exceed content boundaries

### Key Requirements It Fulfills:

• ⚠️ Limits editing when confidence is low or sentence is already optimal

• 🧠 Applies guardrails for psychology/coaching context

• 🚫 Flags content types that should be ignored or skipped

• ✅ Supports lightweight moderation and QA checks

### Code Snippet: (Hypothetical structure — adjust if needed)

def is\_editing\_allowed(text: str, score: float) -> bool:

"""Check if editing is allowed based on quality score threshold."""

return score < QUALITY\_THRESHOLD

def is\_sensitive\_content(text: str) -> bool:

"""Detect whether text includes unsupported or sensitive terms."""

keywords = ["suicide", "abuse", "diagnosis"]

return any(keyword in text.lower() for keyword in keywords)

def should\_skip\_translation(text: str) -> bool:

"""Apply custom rules to skip translation for certain phrases."""

skip\_patterns = ["editor's note", "confidential", "signature:"]

return any(phrase in text.lower() for phrase in skip\_patterns)

### Explanation

• is\_editing\_allowed(): Prevents AI from touching text that already meets quality criteria

• is\_sensitive\_content(): Flags content that might need human oversight instead of AI

• should\_skip\_translation(): Ignores special-purpose content (notes, metadata, etc.) in the pipeline

## test\_changes.py: Unit Testing for Change Tracking System

### Purpose:

This file includes **unit tests** for the changes.py module. It ensures that the system accurately detects and classifies changes — such as insertions, deletions, and replacements — between versions of a text.

Testing is critical in PersianAI because **automated editing must be verifiable and traceable**, especially in sensitive content like multi-author psychology books.

### Why It Was Created:

• To verify the correctness of the track\_changes() function

• To catch regressions when making changes to the diff logic

• To ensure change metrics and visualizations are based on reliable data

• To validate edge cases, such as punctuation differences or partial word changes

### Key Requirements It Fulfills:

• ✅ Confirms core diffing logic works across varied inputs

• ⚙️ Ensures output structure (replacements, deletions, insertions) matches expectations

• 🧪 Enables test-driven development and safe refactoring of change logic

• 🔍 Verifies text handling for Persian and mixed-language content

### Code Snippet: Sample Unit Test from test\_changes.py

def test\_basic\_change\_tracking():

original = "این یک متن ساده است."

modified = "این یک متن خیلی ساده است."

changes = track\_changes(original, modified)

assert changes["insertions"] == 1

assert any("خیلی" in change.get("modified", "") for change in changes["details"])

### Explanation

• Tests a simple case where the word **“خیلی” (very)** is added to the original sentence

• Validates that the system correctly counts **one insertion**

• Confirms that the inserted word appears in the change detail

• More tests like this would include:

• Replacement tests

• Deletion tests

• Formatting and edge cases (punctuation, spacing, etc.)

## test\_connections.py: Testing API Connectivity and Reliability

### Purpose:

This file contains tests that verify whether the system can successfully connect to and interact with **external language model APIs** (OpenAI and Google Gemini). It checks for:

• API key validity

• Response formatting

• Timeout behavior

• Error handling

• Rate limit enforcement

### Why It Was Created:

• To ensure API credentials are valid and functional

• To catch external service issues early (e.g., misconfiguration, quota limits)

• To prevent user-facing failures during editing/translation

• To support robust and fault-tolerant system design

### Key Requirements It Fulfills:

• ✅ Verifies external services are reachable before production use

• 🛡️ Helps prevent unexpected crashes or failures due to bad API keys

• 📊 Validates that responses match expected formats and model outputs

• 🔁 Checks that rate limiting, retries, and fallbacks behave correctly

### Code Snippet: Sample Connection Test

def test\_openai\_connection():

try:

response = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=[{"role": "user", "content": "سلام"}],

max\_tokens=10

)

assert response and "choices" in response

except Exception as e:

assert False, f"OpenAI connection failed: {e}"

### Explanation

• Sends a simple test message (“سلام”) to the OpenAI model

• Asserts that the response is structured correctly and contains a valid "choices" field

• Catches errors such as:

• Invalid API keys

• Network failures

• Service outages

• Can be expanded to test Gemini, rate limits, and latency

## test\_gemini.py: Validating Gemini-Based Editing and Translation

### Purpose:

This file contains unit tests for validating Gemini-based functionality within the PersianAI system — including both Persian content editing and Persian-to-English translation. These tests ensure the Gemini model behaves reliably when processing complex psychological/coaching content and returns well-structured outputs.

### Why It Was Created:

• To verify correct Gemini behavior for both editing and translation workflows

• To ensure API credentials are valid and functioning

• To test error conditions like timeouts, invalid responses, and rate limits

• To validate that Gemini editing improves fluency, grammar, and tone in Persian content

### Key Requirements It Fulfills:

• ✅ Tests Gemini-based Persian editing flow for accuracy and fluency

• ✅ Tests Persian-to-English translation for correctness and structure

• 🔐 Verifies API authentication and error handling

• ⏳ Ensures timeout and long-input stability

• 🔁 Confirms fallback or retry logic (if configured)

• 📊 Ensures response is usable in downstream modules (e.g., visualization, change tracking)

### Code Snippet: Sample Tests

def test\_gemini\_edit\_response():

input\_text = "این متن نیاز به ویرایش دارد"

result = gemini.edit\_text(input\_text)

assert isinstance(result, str)

assert "ویرایش" not in result # Should be rephrased or improved

def test\_gemini\_translation\_response():

prompt = "ترجمه این جمله به انگلیسی: سلام دنیا"

response = gemini.translate\_text(prompt)

assert response is not None

assert isinstance(response, str)

assert "Hello" in response or "Hi" in response

def test\_gemini\_invalid\_key(monkeypatch):

monkeypatch.setenv("GEMINI\_API\_KEY", "invalid-key")

with pytest.raises(Exception) as exc\_info:

gemini.translate\_text("Test sentence")

assert "authentication" in str(exc\_info.value).lower()

### Explanation:

• test\_gemini\_edit\_response() validates that the editing function improves the Persian input and returns fluent, publication-ready text.

• test\_gemini\_translation\_response() ensures the translation logic produces accurate English output.

• test\_gemini\_invalid\_key() checks error handling when API credentials are incorrect.

## 📘 Summary: Code Documentation

The PersianAI system is structured around modular, well-tested components that together support an end-to-end workflow for editing and translating Persian text with minimal human involvement. Each file plays a clear role in supporting professional, scalable publishing — especially for multi-author psychology and coaching books.

### Key Modules Recap

|  |  |
| --- | --- |
| File / Module | Role in System |
| translation\_bot.py | System controller. Hosts the FastAPI app, manages request routing, and orchestrates editing → translation pipeline. |
| improvements.py | Editing engine. Applies grammar, tone, and clarity improvements using prompt engineering and Persian-specific rules. |
| changes.py | Diff tracker. Tracks word-level changes and generates structured diff outputs for review and rollback. |
| limitations.py | Constraint manager. Enforces AI boundaries, detects sensitive content, and applies editing restrictions. |
| test\_changes.py | Unit tests for word-level diff tracking. Validates change detection, formatting, and edge cases. |
| test\_connections.py | Verifies OpenAI and Gemini API connectivity, authentication, and response structures. |
| test\_gemini.py | Validates Gemini-specific functionality, output formatting, and integration robustness. |
| requirements.txt | Defines all Python package dependencies for easy setup and reproducibility. |
| .env.example | Provides a template for secure API key and environment variable management. |

# 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

# Function Documentation

## 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  
 """

## 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  
 """

## 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

## 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)

# 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()

## Edit Request via REST API (Python)

def edit\_text(text: str, model\_type: str = "gemini", edit\_mode: str = "fast"):

response = requests.post(

"http://localhost:8000/edit",

json={

"text": text,

"model\_type": model\_type,

"edit\_mode": edit\_mode

}

)

return response.json()

# Example usage

edited = edit\_text("این جمله نیاز به ویرایش دارد.")

print(edited["edited\_text"])

## WebSocket Integration

Real-time updates integration:

import asyncio

import websockets

async def connect\_websocket():

uri = "ws://localhost:8000/ws"

async with websockets.connect(uri) as websocket:

await websocket.send("start\_translation")

while True:

message = await websocket.recv()

print("Update:", message)

asyncio.run(connect\_websocket())

## Third-party Integrations

### OpenAI Integration (Async Client)

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 Persian-English translator."},

{"role": "user", "content": f"Translate: {text}"}

]

)

return response.choices[0].message.content

### Gemini Integration (Google Generative AI)

## import google.generativeai as genai

## def gemini\_translate(text: str):

## model = genai.GenerativeModel("gemini-pro")

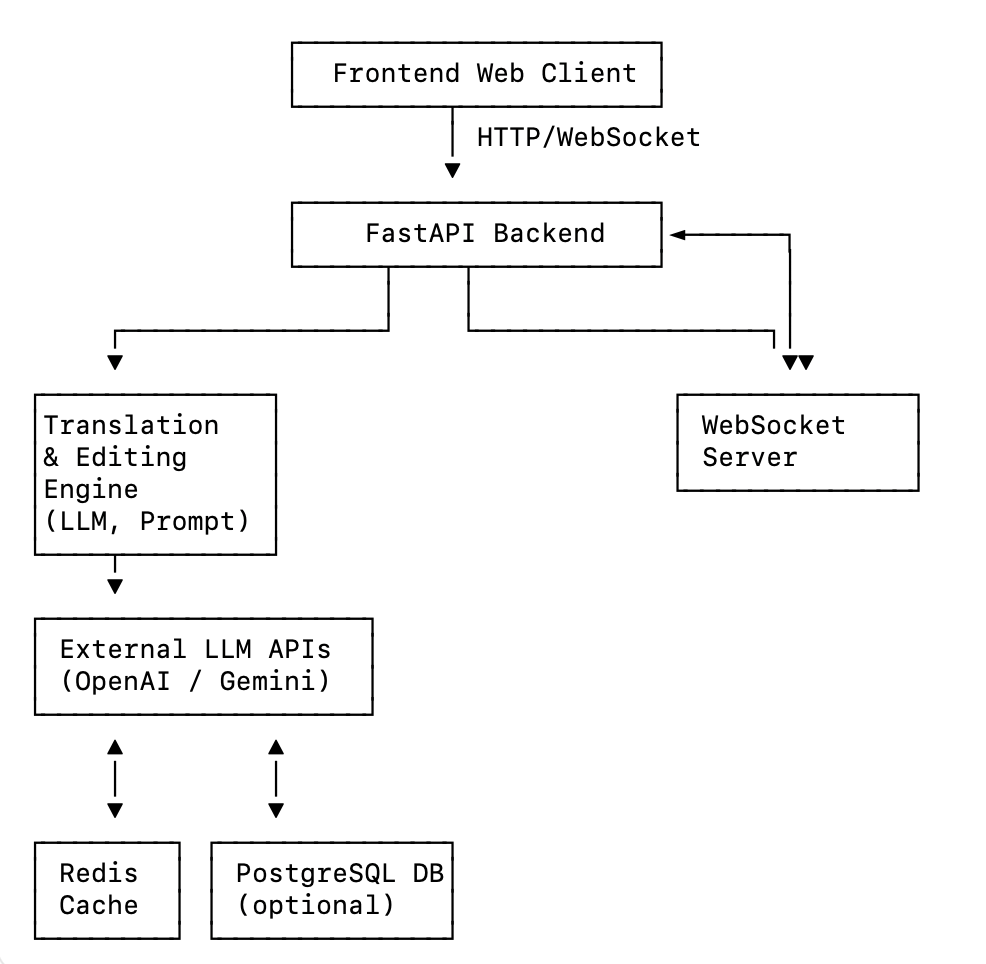
## response = model.generate\_content(f"Translate this Persian text into English: {text}")

## return response.text

# 6.13 Visual Aids and Diagrams

Visual representations help illustrate PersianAI’s system architecture, data flow, and error-handling strategies. These diagrams support technical understanding for developers, integrators, and reviewers.

## System Architecture Diagram



## Data Flow Diagram

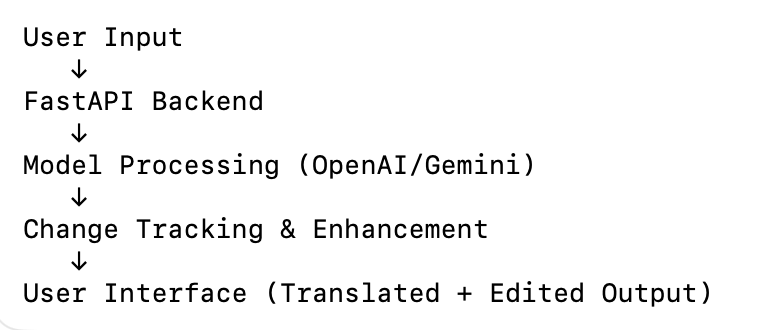
1. **User Input** → Sent from the frontend (e.g., text to edit or translate)

2. **FastAPI Backend** → Handles routing, validation, and preprocessing

3. **LLM Services** → OpenAI or Gemini performs editing and/or translation

4. **Postprocessing** → Changes are tracked, visual diff generated

5. **Frontend Output** → Enhanced English output with change logs



## Error Handling Flowchart

The error-handling flowchart outlines how PersianAI detects, manages, and recovers from failures across its system. It is designed to ensure reliability, user transparency, and minimal disruption in production environments.

### Steps Explained:

### 1. Error Detection

Errors are automatically captured during request processing, API communication, or model execution. Examples include invalid user input, model timeouts, or authentication failures.

### 2. Error Classification

Each error is classified into one of the following categories:

• **API Error**: Issues with OpenAI or Gemini (e.g., rate limits, downtime)

• **Input Error**: Malformed requests or unsupported content (e.g., sensitive keywords)

• **Internal Error**: Server exceptions, timeouts, or memory issues

### 3. Handling Strategy Selection

Depending on the error type and severity, the system decides how to respond:

• Retry the request

• Switch to a fallback model

• Skip processing the problematic input

• Notify the user of the issue

### 4. Recovery Action Execution

The system applies the selected action:

• Retries API calls with exponential backoff

• Switches models (e.g., from GPT-4 to Gemini Pro)

• Gracefully skips specific content blocks

• Catches the error and returns a safe fallback response

### 5. User Notification

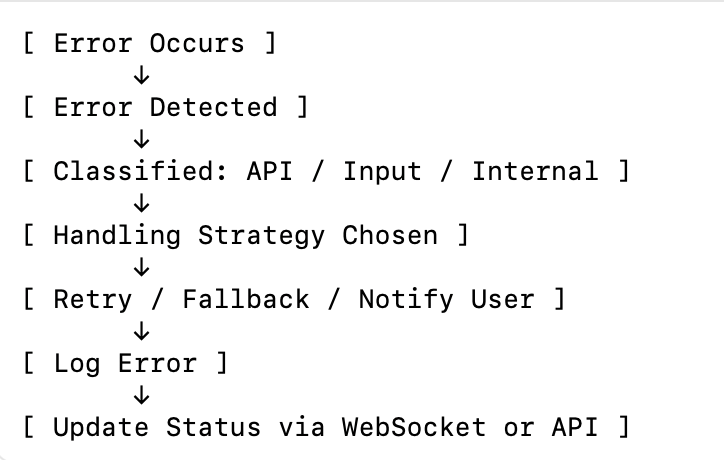
Users are informed of the error via:

• WebSocket messages (for real-time workflows)

• JSON error responses in REST API

### 6. Logging and Monitoring

All errors are logged with full context (e.g., user request, error trace) for debugging and monitoring. Logs are stored securely and can be reviewed for performance tuning or incident response.



# 6.15 Environment Setup and API Keys

## Environment Variables

PersianAI relies on environment variables to securely manage configuration settings and API keys. These are defined in a .env file located in the root directory.

To ensure flexibility, security, and ease of deployment, PersianAI uses environment variables defined in a .env file. These variables are categorized as follows:

• **API Keys**: Provide secure access to OpenAI and Gemini services (OPENAI\_API\_KEY, GEMINI\_API\_KEY).

• **Database & Caching**: Define backend connection details for PostgreSQL and Redis (DATABASE\_URL, REDIS\_URL).

• **Application Behavior**: Configure runtime behavior, such as logging level, response limits, and API rate control (LOG\_LEVEL, MAX\_TOKENS, CACHE\_TTL, RATE\_LIMIT).

• **Environment Control**: Specifies the deployment context (ENVIRONMENT), allowing different behavior in development vs. production.

|  |  |  |
| --- | --- | --- |
| 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 use OpenAI’s models (GPT-3.5, GPT-4), follow these steps to obtain your API key:

1. Go to [OpenAI’s platform dashboard](https://platform.openai.com/).

2. Sign in or create a new account.

3. Navigate to the **API Keys** section in your profile.

4. Click **“Create new secret key”** and copy the generated key.

5. Paste the key into your .env file as the value for OPENAI\_API\_KEY.

6. **Important**: Do **not** share or commit this key to version control.

### Gemini API Key

To access Google’s Gemini Pro or Flash models:

1. Visit [Google MakerSuite](https://makersuite.google.com/app/apikey).

2. Log in with your Google account.

3. Click **“Create API key”** to generate a new one.

4. Copy the key and add it to your .env file as GEMINI\_API\_KEY.

5. Keep the key confidential and secured at all times.

## Security Best Practices

**Never commit** .env files to version control (e.g., GitHub).

• **Use separate API keys** for development and production environments.

• **Rotate keys regularly** to reduce risk from accidental leaks.

• **Monitor API usage** to detect unusual activity or quota overages.

• **Implement rate limiting** in your application to avoid abuse or bans.

• Maintain multiple .env files (e.g., .env.development, .env.production) for different deployment scenarios.

## 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

# Module Documentation

This section provides **detailed technical documentation** for each Python module in the PersianAI system. Each file plays a unique role in the end-to-end pipeline — from preprocessing and editing, to translation, version tracking, and API interaction.

The goal is to:

• ✅ Help developers understand the function and structure of each module

• 🧩 Clarify how different components interact and share responsibilities

• 🔍 Support maintainability, debugging, and onboarding of new contributors

• 🛠️ Document key functions, use cases, and design decisions in a standardized format

## translation\_bot.py Module

The translation\_bot.py module acts as the central orchestrator of the PersianAI system, hosting the FastAPI application and managing all editing and translation flows. It defines the system's entry points and provides REST and WebSocket interfaces for real-time and batch processing.

### Core Functions

#### create\_app() Function:

Purpose**:** Creates and configures the FastAPI application.

def create\_app():

"""Creates and configures the FastAPI application.

Returns:

FastAPI: Configured FastAPI application instance.

"""

app = FastAPI(

title="PersianAI Translation Service",

description="Professional Persian to English translation service",

version="1.0.0"

)

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_credentials=True,

allow\_methods=["\*"],

allow\_headers=["\*"],

)

app.middleware("http")(add\_process\_time\_header)

return app

### add\_process\_time\_header() Function:

**Purpose:** Adds response header to track request processing time.

@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

**Key Features:**

* Measures request processing duration
* Adds X-Process-Time header to the HTTP response
* Useful for performance monitoring and debugging

### Translation Endpoint Functions

#### translate\_text() Function:

**Purpose:** Handles incoming text translation requests and orchestrates preprocessing, chunking, model processing, and response aggregation.

async def translate\_text(text: str, model\_type: str = "gpt-3.5-turbo") -> str:

try:

text = preprocess\_persian\_text(text)

chunks = await chunk\_text(text)

tasks = []

for chunk in chunks:

if model\_type in ["gpt-3.5-turbo", "gpt-4"]:

tasks.append(process\_openai\_translation(chunk, model\_type))

else:

tasks.append(process\_gemini\_translation(chunk))

translations = await asyncio.gather(\*tasks)

result = combine\_translations(translations)

validate\_translation(result)

return result

except Exception as e:

logger.error(f"Translation error: {str(e)}")

raise

**Key Features:**

* Accepts large input, splits into chunks
* Supports OpenAI and Gemini models
* Runs chunked translations asynchronously
* Aggregates and validates final output

#### process\_openai\_translation() Function:

**Purpose:** Translates a chunk using OpenAI’s chat completion API.

async def process\_openai\_translation(text: str, model: str) -> str:

response = await openai.ChatCompletion.acreate(

model=model,

messages=[{"role": "user", "content": text}]

)

return response.choices[0].message.content

#### process\_gemini\_translation() Function:

**Purpose:** Translates a chunk using Google’s Gemini model.

def process\_gemini\_translation(text: str) -> str:

model = genai.GenerativeModel('gemini-pro')

response = model.generate\_content(f"Translate: {text}")

return response.text

### Supporting Utilities

#### chunk\_text() Function:

**Purpose:** Splits long text into smaller chunks based on sentence boundaries for efficient processing.

async def chunk\_text(text: str, max\_chunk\_size: int = 1000) -> List[str]:

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

### **Key Features:**

* Sentence-aware chunking
* Avoids breaking context mid-sentence
* Handles long documents efficiently

The translation\_bot.py module is foundational to PersianAI, tying together all components and ensuring requests flow smoothly between clients and the model-based backends.

## 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."""