SMART ENSEMBLE LANGUAGE LOOKUP SYSTEM
Complete Project Documentation and Code - Updated with Match Rationale Feature

PROJECT OVERVIEW
This is an advanced language lookup system that uses ensemble retrieval methods to match PDF document content against a

FEATURES
- Smart Ensemble Retrieval: LLM + BM25 + TF-IDF for candidate selection
- Multiple Similarity Methods: OpenAI, SentenceTransformers, TF-IDF
- LangGraph Multi-Agent Workflow for enhanced analysis
- **NEW: Intelligent Match Rationale** - Explains why each language match was found
- Comprehensive reasoning includes: exact matches, semantic similarity, technical context
- No vector database dependency (more reliable)
- Professional Streamlit interface with rationale display
- Export capabilities (JSON, Markdown) with match reasoning

PROJECT STRUCTURE
```
langchain-lookup-project/
■■■ src/
■   ■■■ ensemble_retriever.py       # Multi-method retrieval system with rationale
■   ■■■ smart_similarity_search.py  # Main search pipeline with reasoning
■   ■■■ pdf_processor.py           # LangChain PDF processing
■   ■■■ langgraph_workflow.py      # Multi-agent AI workflow
■■■ data/
■   ■■■ languages.csv             # AI/ML language database
■   ■■■ ai_topics.txt             # Sample document content
■■■ vectorstore/                  # Auto-created directory
■■■ app.py                        # Streamlit interface with rationale display
■■■ requirements.txt              # Dependencies
■■■ .env                         # Environment variables
■■■ README.md                    # Project documentation
```

REQUIREMENTS (requirements.txt)
```
streamlit
langchain
langchain-openai
langchain-community
langgraph
rank_bm25
scikit-learn
openai
pandas
numpy
pypdf
python-dotenv
tiktoken
sentence-transformers

pydantic
```

ENVIRONMENT VARIABLES (.env)
```
OPENAI_API_KEY=your_openai_api_key_here
CHROMA_INDEX_PATH=./vectorstore/language_index
LANGCHAIN_TRACING_V2=false
LANGCHAIN_API_KEY=your_langsmith_api_key_here
```

SOURCE CODE FILES

1. ENSEMBLE RETRIEVER (src/ensemble_retriever.py) - Updated with Rationale
```python
import os
import pandas as pd
from typing import List, Dict, Any, Optional
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain.prompts import PromptTemplate
```

```
from langchain.schema import HumanMessage
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sentence_transformers import SentenceTransformer
import numpy as np
from rank_bm25 import BM25Okapi
import json


class EnsembleRetriever:
    def __init__(self, languages_csv_path: str):
        self.languages_df = pd.read_csv(languages_csv_path)
        self.languages = self.languages_df['Language'].tolist()

        # Initialize components (with error handling for API quota)
        try:
            self.llm = ChatOpenAI(
                model="gpt-4o-mini",  # Faster and cheaper for retrieval
                temperature=0.1,
                openai_api_key=os.getenv('OPENAI_API_KEY')
            )
            self.llm_available = True
        except Exception as e:
            print(f"■■ OpenAI LLM not available: {e}")
            self.llm = None
            self.llm_available = False

        # Initialize embedding models
        try:
            self.openai_embeddings = OpenAIEmbeddings(

                model="text-embedding-3-small",
                openai_api_key=os.getenv('OPENAI_API_KEY')
            )
            self.openai_available = True
        except Exception as e:
            print(f"■■ OpenAI Embeddings not available: {e}")
            self.openai_embeddings = None
            self.openai_available = False

        self.sentence_model = SentenceTransformer('all-MiniLM-L6-v2')

        # Initialize TF-IDF
        self.tfidf_vectorizer = TfidfVectorizer(
            stop_words='english',
            max_features=5000,
            lowercase=True
        )

        # Initialize BM25
        tokenized_languages = [lang.lower().split() for lang in self.languages]
        self.bm25 = BM25Okapi(tokenized_languages)

        # Fit TF-IDF on languages
        self.tfidf_vectorizer.fit(self.languages)

        print(f"■ Ensemble Retriever initialized with {len(self.languages)} languages")

    def _generate_match_rationale(self, chunk: str, term: str, similarity_score: float,
                                  sources: List[str], retrieval_method: str) -> str:
        """Generate human-readable rationale for why a language match was found"""

        # Extract key phrases from chunk for analysis
        chunk_lower = chunk.lower()
        term_lower = term.lower()

        # Check for exact matches
        exact_match = term_lower in chunk_lower

        # Check for partial word matches
        term_words = term_lower.split()
```

```python
        chunk_words = chunk_lower.split()
        matching_words = [word for word in term_words if any(word in chunk_word for chunk_word in chunk_words)]

        # Build rationale based on evidence
        rationale_parts = []

        if exact_match:
            rationale_parts.append(f"Direct mention of '{term}' found in the text")
        elif matching_words:
            rationale_parts.append(f"Related terms found: {', '.join(matching_words)}")

        # Add similarity context
        if similarity_score > 0.8:
            rationale_parts.append("Very high semantic similarity")
        elif similarity_score > 0.6:
            rationale_parts.append("High semantic similarity")
        elif similarity_score > 0.4:
            rationale_parts.append("Moderate semantic similarity")
        else:
            rationale_parts.append("Lower but significant semantic relationship")

        # Add retrieval method context
        source_context = {
            "llm": "LLM identified semantic relevance",
            "llm_fallback": "LLM analysis with fallback matching",
            "bm25": "Strong keyword matching using BM25",
            "tfidf": "Statistical text similarity detected"
        }

        method_explanations = []
        for source in sources:
            if source in source_context:
                method_explanations.append(source_context[source])

        if method_explanations:
            rationale_parts.append(f"Detection methods: {'; '.join(method_explanations)}")

        # Add embedding method context
        embedding_context = {
            "openai": "using OpenAI embeddings for deep semantic understanding",
            "sentence_transformer": "using SentenceTransformer for semantic analysis",
            "tfidf": "using TF-IDF statistical analysis"
        }

        if retrieval_method in embedding_context:
            rationale_parts.append(embedding_context[retrieval_method])

        # Look for technical context indicators
        tech_indicators = [
            "algorithm", "model", "training", "learning", "neural", "network",
            "data", "prediction", "classification", "optimization", "artificial",
            "intelligence", "machine", "deep", "analysis", "processing"
        ]

        found_indicators = [indicator for indicator in tech_indicators if indicator in chunk_lower]
        if found_indicators:
            rationale_parts.append(f"Technical context detected: {', '.join(found_indicators[:3])}")

        # Combine rationale parts
        if rationale_parts:

            return ". ".join(rationale_parts) + "."
        else:
            return f"Match found based on semantic similarity ({similarity_score:.3f}) using {retrieval_method} embeddi

    def llm_based_retrieval(self, query: str, top_k: int = 10) -> List[Dict[str, Any]]:
        """Use LLM to identify relevant language terms"""

        if not self.llm_available:
            print("■■ LLM not available, skipping LLM retrieval")
            return []
```

```
        # Prepare a sample of languages for the prompt (to avoid token limits)
        sample_languages = self.languages[:50]  # First 50 as examples

        retrieval_prompt = PromptTemplate(
            input_variables=["query", "languages"],
            template="""
            Given the following query about AI/ML/Data Science content:
            Query: {query}

            From this list of language terms:
            {languages}

            Identify the TOP {top_k} most relevant language terms that would likely appear in or relate to this query.
            Consider:
            1. Semantic similarity and conceptual relationships
            2. Technical context and domain relevance
            3. Common usage patterns in AI/ML literature

            Return ONLY a JSON list of the most relevant terms, like:
            ["Machine Learning", "Deep Learning", "Neural Networks"]

            Focus on quality over quantity - return only highly relevant matches.
            """.replace("{top_k}", str(top_k))
        )

        try:
            prompt = retrieval_prompt.format(
                query=query[:500],  # Limit query length
                languages=", ".join(sample_languages)
            )

            response = self.llm.invoke([HumanMessage(content=prompt)])

            # Parse JSON response
            try:
                relevant_terms = json.loads(response.content)
                if isinstance(relevant_terms, list):
                    # Filter to ensure terms exist in our database
                    filtered_terms = [term for term in relevant_terms if term in self.languages]

                    return [{"term": term, "source": "llm", "score": 1.0} for term in filtered_terms[:top_k]]
            except json.JSONDecodeError:
                # Fallback: extract terms from response text
                response_text = response.content.lower()
                found_terms = []
                for lang in self.languages:
                    if lang.lower() in response_text and len(found_terms) < top_k:
                        found_terms.append({"term": lang, "source": "llm_fallback", "score": 0.8})
                return found_terms

        except Exception as e:
            print(f"LLM retrieval error: {e}")
            return []

        return []

    def bm25_retrieval(self, query: str, top_k: int = 10) -> List[Dict[str, Any]]:
        """Use BM25 for keyword-based retrieval"""
        tokenized_query = query.lower().split()
        scores = self.bm25.get_scores(tokenized_query)

        # Get top k indices
        top_indices = np.argsort(scores)[::-1][:top_k]

        results = []
        for idx in top_indices:
            if scores[idx] > 0:  # Only include non-zero scores
                results.append({
                    "term": self.languages[idx],
                    "source": "bm25",
                    "score": float(scores[idx])
```

```python
            })

        return results

    def tfidf_retrieval(self, query: str, top_k: int = 10) -> List[Dict[str, Any]]:
        """Use TF-IDF for statistical text retrieval"""
        query_vector = self.tfidf_vectorizer.transform([query])
        language_vectors = self.tfidf_vectorizer.transform(self.languages)

        similarities = cosine_similarity(query_vector, language_vectors)[0]

        # Get top k indices
        top_indices = np.argsort(similarities)[::-1][:top_k]

        results = []
        for idx in top_indices:
            if similarities[idx] > 0.01:  # Minimum threshold
                results.append({
                    "term": self.languages[idx],

                    "source": "tfidf",
                    "score": float(similarities[idx])
                })

        return results

    def ensemble_retrieval(self, query: str, top_k: int = 15) -> List[Dict[str, Any]]:
        """Combine multiple retrieval methods using ensemble approach"""

        # Get results from different methods
        llm_results = self.llm_based_retrieval(query, top_k=8)
        bm25_results = self.bm25_retrieval(query, top_k=10)
        tfidf_results = self.tfidf_retrieval(query, top_k=10)

        # Combine and deduplicate results
        term_scores = {}

        # Weight different methods
        weights = {
            "llm": 0.5,        # Highest weight for semantic understanding
            "llm_fallback": 0.3,
            "bm25": 0.3,       # Good for keyword matching
            "tfidf": 0.2       # Statistical baseline
        }

        # Aggregate scores
        for results in [llm_results, bm25_results, tfidf_results]:
            for result in results:
                term = result["term"]
                source = result["source"]
                score = result["score"] * weights.get(source, 0.1)

                if term not in term_scores:
                    term_scores[term] = {"score": 0, "sources": []}

                term_scores[term]["score"] += score
                term_scores[term]["sources"].append(source)

        # Sort by combined score and return top k
        sorted_terms = sorted(
            [(term, data["score"], data["sources"]) for term, data in term_scores.items()],
            key=lambda x: x[1],
            reverse=True
        )[:top_k]

        ensemble_results = []
        for term, score, sources in sorted_terms:
            ensemble_results.append({
                "term": term,
                "ensemble_score": score,
```

```
                "sources": sources,
                "source_count": len(set(sources))
            })

        return ensemble_results

    def compute_final_similarity(self, chunk: str, candidate_terms: List[Dict],
                                method: str = "openai") -> List[Dict[str, Any]]:
        """Compute final similarity scores using specified embedding method and generate rationale"""

        if not candidate_terms:
            return []

        terms = [item["term"] for item in candidate_terms]

        if method == "openai" and self.openai_available:
            try:
                # Get OpenAI embeddings
                chunk_embedding = self.openai_embeddings.embed_query(chunk)
                term_embeddings = self.openai_embeddings.embed_documents(terms)

                # Compute cosine similarities
                similarities = []
                chunk_emb = np.array(chunk_embedding)
                for term_emb in term_embeddings:
                    term_emb = np.array(term_emb)
                    similarity = np.dot(chunk_emb, term_emb) / (
                        np.linalg.norm(chunk_emb) * np.linalg.norm(term_emb)
                    )
                    similarities.append(float(similarity))
            except Exception as e:
                print(f"OpenAI embeddings error: {e}, falling back to sentence transformer")
                method = "sentence_transformer"

        if method == "sentence_transformer" or (method == "openai" and not self.openai_available):
            # Get sentence transformer embeddings
            all_texts = [chunk] + terms
            embeddings = self.sentence_model.encode(all_texts)
            chunk_emb = embeddings[0]
            term_embs = embeddings[1:]

            similarities = cosine_similarity([chunk_emb], term_embs)[0]
            similarities = [float(sim) for sim in similarities]

        elif method == "tfidf":
            # Use TF-IDF similarity
            all_texts = [chunk] + terms
            tfidf_matrix = self.tfidf_vectorizer.transform(all_texts)
            chunk_vector = tfidf_matrix[0]
            term_vectors = tfidf_matrix[1:]

            similarities = cosine_similarity(chunk_vector, term_vectors)[0]
            similarities = [float(sim) for sim in similarities]

        else:
            # Default to simple string similarity
            similarities = [0.5 for _ in terms]

        # Combine ensemble scores with similarity scores and generate rationale
        final_results = []
        for i, term_data in enumerate(candidate_terms):
            final_score = similarities[i]
            ensemble_score = term_data.get("ensemble_score", 0)

            # Weighted combination of ensemble retrieval and similarity
            combined_score = 0.6 * final_score + 0.4 * (ensemble_score / max(1, max(item.get("ensemble_score", 1) for i

            # Generate rationale for this match
            rationale = self._generate_match_rationale(
                chunk=chunk,
                term=term_data["term"],
```

```
            similarity_score=final_score,
            sources=term_data.get("sources", []),
            retrieval_method=method
        )

        final_results.append({
            "term": term_data["term"],
            "similarity_score": final_score,
            "ensemble_score": ensemble_score,
            "combined_score": combined_score,
            "sources": term_data.get("sources", []),
            "retrieval_method": method,
            "rationale": rationale
        })

    # Sort by combined score
    final_results.sort(key=lambda x: x["combined_score"], reverse=True)

    return final_results

def search_chunk(self, chunk: str, similarity_method: str = "openai",
                top_k_retrieval: int = 15, top_k_final: int = 5,
                threshold: float = 0.7) -> Dict[str, Any]:
    """Complete search pipeline for a single chunk"""

    # Step 1: Ensemble retrieval to get candidate terms
    candidates = self.ensemble_retrieval(chunk, top_k=top_k_retrieval)

    if not candidates:

        return {
            "chunk_preview": chunk[:200] + "..." if len(chunk) > 200 else chunk,
            "candidates_found": 0,
            "final_matches": [],
            "status": "No candidates found"
        }

    # Step 2: Compute final similarities
    final_matches = self.compute_final_similarity(
        chunk, candidates, method=similarity_method
    )[:top_k_final]

    # Step 3: Apply threshold
    threshold_matches = [
        match for match in final_matches
        if match["combined_score"] >= threshold
    ]

    best_match = threshold_matches[0] if threshold_matches else None

    return {
        "chunk_preview": chunk[:200] + "..." if len(chunk) > 200 else chunk,
        "candidates_found": len(candidates),
        "final_matches": final_matches,
        "threshold_matches": threshold_matches,
        "best_match": best_match,
        "status": "Language found" if best_match else "Not found"
    }

def search_multiple_chunks(self, chunks: List[str], similarity_method: str = "openai",
                        top_k_retrieval: int = 15, top_k_final: int = 5,
                        threshold: float = 0.7) -> List[Dict[str, Any]]:
    """Search multiple chunks efficiently"""

    results = []

    for i, chunk in enumerate(chunks):
        print(f"Processing chunk {i+1}/{len(chunks)}")

        result = self.search_chunk(
            chunk=chunk,
```

```
                similarity_method=similarity_method,
                top_k_retrieval=top_k_retrieval,
                top_k_final=top_k_final,
                threshold=threshold
            )

            result["chunk_index"] = i
            results.append(result)

        return results
```

2. SMART SIMILARITY SEARCH (src/smart_similarity_search.py) - Updated
```python
from typing import List, Dict, Any
from src.ensemble_retriever import EnsembleRetriever
from src.pdf_processor import LangChainPDFProcessor
from src.langgraph_workflow import LanguageLookupWorkflow
import numpy as np
import json


class SmartSimilaritySearch:
    def __init__(self, languages_csv_path: str = "./data/languages.csv"):
        self.pdf_processor = LangChainPDFProcessor()
        self.ensemble_retriever = EnsembleRetriever(languages_csv_path)
        self.workflow = LanguageLookupWorkflow()

    def process_pdf_document(self, uploaded_file) -> Dict[str, Any]:
        """Process PDF document using LangChain"""
        try:
            # Process PDF into documents
            documents = self.pdf_processor.process_uploaded_pdf(uploaded_file)

            # Extract text chunks
            chunks = self.pdf_processor.process_text_chunks(documents)

            # Get enhanced chunk data
            enhanced_chunks = self.pdf_processor.create_enhanced_chunks(documents)

            return {
                "success": True,
                "documents": documents,
                "chunks": chunks,
                "enhanced_chunks": enhanced_chunks,
                "total_chunks": len(chunks),
                "metadata": {
                    "source_file": uploaded_file.name,
                    "total_pages": len(set(doc.metadata.get('page', 0) for doc in documents)),
                    "avg_chunk_size": np.mean([len(chunk) for chunk in chunks]) if chunks else 0
                }
            }
        except Exception as e:
            return {
                "success": False,
                "error": str(e),
                "documents": [],
                "chunks": [],
                "enhanced_chunks": [],

                "total_chunks": 0
            }

    def search_with_ensemble(self, chunks: List[str],
                           similarity_method: str = "openai",
                           top_k_retrieval: int = 15,
                           top_k_final: int = 5,
                           threshold: float = 0.7) -> List[Dict[str, Any]]:
        """Search using ensemble retrieval approach"""

        return self.ensemble_retriever.search_multiple_chunks(
```

```python
            chunks=chunks,
            similarity_method=similarity_method,
            top_k_retrieval=top_k_retrieval,
            top_k_final=top_k_final,
            threshold=threshold
        )

    def run_complete_analysis(self, uploaded_file,
                            similarity_method: str = "openai",
                            top_k_retrieval: int = 15,
                            top_k_final: int = 5,
                            threshold: float = 0.7,
                            enable_workflow: bool = True) -> Dict[str, Any]:
        """Run complete document analysis pipeline"""

        results = {
            "pdf_processing": {},
            "ensemble_search": {},
            "workflow_analysis": {},
            "summary": {}
        }

        # Step 1: Process PDF
        print("■ Processing PDF document...")
        pdf_results = self.process_pdf_document(uploaded_file)
        results["pdf_processing"] = pdf_results

        if not pdf_results["success"]:
            return results

        chunks = pdf_results["chunks"]

        # Step 2: Ensemble Search
        print("■ Running ensemble retrieval and similarity matching...")
        search_results = self.search_with_ensemble(
            chunks=chunks,
            similarity_method=similarity_method,
            top_k_retrieval=top_k_retrieval,
            top_k_final=top_k_final,

            threshold=threshold
        )

        # Calculate statistics
        found_matches = sum(1 for r in search_results if r["status"] == "Language found")
        success_rate = (found_matches / len(search_results) * 100) if search_results else 0

        results["ensemble_search"] = {
            "results": search_results,
            "total_chunks": len(chunks),
            "found_matches": found_matches,
            "success_rate": success_rate,
            "method_used": similarity_method
        }

        # Step 3: LangGraph Workflow (if enabled)
        if enable_workflow:
            print("■ Running LangGraph analysis workflow...")
            # Convert search results to format expected by workflow
            workflow_results = self.workflow.run_workflow(chunks, search_results)
            results["workflow_analysis"] = workflow_results

        # Step 4: Generate Summary
        results["summary"] = self._generate_summary(results)

        return results

    def _generate_summary(self, results: Dict[str, Any]) -> Dict[str, Any]:
        """Generate summary statistics"""

        pdf_results = results.get("pdf_processing", {})
```

```
        search_results = results.get("ensemble_search", {})
        workflow_results = results.get("workflow_analysis", {})

        summary = {
            "total_chunks": pdf_results.get("total_chunks", 0),
            "found_matches": search_results.get("found_matches", 0),
            "success_rate": search_results.get("success_rate", 0),
            "method_used": search_results.get("method_used", "unknown"),
            "workflow_enabled": bool(workflow_results),
            "processing_success": pdf_results.get("success", False),
            "search_success": bool(search_results.get("results")),
            "workflow_success": workflow_results.get("success", False) if workflow_results else None
        }

        # Add workflow insights if available
        if workflow_results and workflow_results.get("success"):
            analysis = workflow_results.get("analysis", {})
            if "match_statistics" in analysis:
                summary.update(analysis["match_statistics"])

        return summary

    def export_results(self, results: Dict[str, Any], format: str = "json") -> str:
        """Export results in specified format with rationale included"""

        if format == "json":
            return json.dumps(results, indent=2, default=str)

        elif format == "markdown":
            summary = results.get("summary", {})
            search_results = results.get("ensemble_search", {}).get("results", [])
            workflow_report = results.get("workflow_analysis", {}).get("report", "")

            markdown_report = f"""
# Smart Ensemble Language Lookup Report

## Summary Statistics
- **Total Chunks**: {summary.get('total_chunks', 0)}
- **Found Matches**: {summary.get('found_matches', 0)}
- **Success Rate**: {summary.get('success_rate', 0):.1f}%
- **Method Used**: {summary.get('method_used', 'unknown')}

## Ensemble Retrieval Results
{"### Found Matches" if summary.get('found_matches', 0) > 0 else "### No Matches Found"}

"""

            for result in search_results[:10]:  # Show first 10 results
                status_icon = "■" if result["status"] == "Language found" else "■"
                best_match = result.get("best_match", {})

                markdown_report += f"""
{status_icon} **Chunk {result.get('chunk_index', 0) + 1}**: {result['status']}
- **Text Preview**: {result['chunk_preview']}
- **Candidates Found**: {result.get('candidates_found', 0)}
"""

                if best_match:
                    markdown_report += f"""- **Best Match**: {best_match.get('term', 'None')}
- **Combined Score**: {best_match.get('combined_score', 0):.3f}
- **Similarity Score**: {best_match.get('similarity_score', 0):.3f}
- **Ensemble Score**: {best_match.get('ensemble_score', 0):.3f}
- **Sources**: {', '.join(best_match.get('sources', []))}
- **Rationale**: {best_match.get('rationale', 'No rationale provided')}
"""

                markdown_report += "\n---\n"

            if workflow_report:

                markdown_report += f"\n## AI Analysis Report\n\n{workflow_report}"
```

```
            return markdown_report

        else:
            return str(results)
```

MATCH RATIONALE FEATURE EXPLANATION

How the Rationale System Works:

1. **Direct Text Analysis**: Checks for exact mentions of language terms in the document chunk
2. **Partial Word Matching**: Identifies related terms and word components
3. **Semantic Similarity Assessment**: Categorizes the strength of semantic relationships
4. **Retrieval Method Context**: Explains which methods detected the match (LLM, BM25, TF-IDF)
5. **Embedding Analysis**: Describes the embedding method used for final similarity computation
6. **Technical Context Detection**: Identifies AI/ML related terms that provide supporting evidence

Example Rationales:

- "Direct mention of 'Machine Learning' found in the text. Very high semantic similarity. Detection methods: LLM identi

- "Related terms found: neural, network. High semantic similarity. Detection methods: Statistical text similarity detec

Benefits:

1. **Transparency**: Users understand why matches were made
2. **Confidence**: Higher confidence in system decisions
3. **Debugging**: Easier to identify false positives/negatives
4. **Education**: Learn about AI/ML relationships in documents
5. **Quality Assurance**: Validate matching logic

INSTALLATION INSTRUCTIONS

1. Create project directory and virtual environment:
```bash
mkdir langchain-lookup-project
cd langchain-lookup-project
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

2. Install dependencies:
```bash
pip install -r requirements.txt
```

3. Set up environment variables:
```bash

# Create .env file with your OpenAI API key
echo "OPENAI_API_KEY=your_openai_api_key_here" > .env
```

4. Create data directory and add language database:
```bash
mkdir data
# Add languages.csv file to data/ directory
```

5. Run the Streamlit application:
```bash
streamlit run app.py
```

USAGE

1. **Upload PDF**: Select a PDF document containing AI/ML content
2. **Configure Settings**: Choose similarity method, thresholds, and retrieval parameters
3. **Run Analysis**: Click "Analyze Document" to process the file
4. **Review Results**: Examine matches with detailed rationale explanations
5. **Export Data**: Download results as JSON or Markdown with reasoning included

The system now provides comprehensive explanations for every language match, making it transparent and trustworthy for

---

System Architecture: Ensemble Retrieval → Similarity Computation → Rationale Generation → Results Display

Key Innovation: Intelligent match explanation combining multiple evidence sources for transparent, explainable AI-p