All Code Files - Gemini RAG Project

This file contains ALL the code you need to run the project. Each section is a separate file.

File 1: `requirements.txt`

```
# Google Gemini
google-generativeai>=0.3.0

# Future AGI SDKs
traceai-google-genai>=0.1.0
fi-instrumentation>=0.1.0
fi-evaluation>=0.1.0

# Vector Database
chromadb>=0.4.22

# Document Processing
pypdf>=4.0.0
python-docx>=1.1.0

# Token Counting
tiktoken>=0.5.2

# Web Interface
gradio>=4.16.0

# Utilities
python-dotenv>=1.0.0
pydantic>=2.5.0
pydantic-settings>=2.1.0
```

File 2: `.env.example`

```
bash
```

```
# Future AGI Credentials
# Get these from: https://app.futureagi.com/settings/api-keys
FI_API_KEY=your_future_agi_api_key_here
FI_SECRET_KEY=your_future_agi_secret_key_here

# Google Gemini API Key
# Get from: https://aistudio.google.com/app/apikey
GOOGLE_API_KEY=your_google_api_key_here

# Project Configuration
PROJECT_NAME=gemini-document-qa
ENVIRONMENT=development
```

File 3: `config.py`

```
python
```

```python
"""
Configuration Management for Gemini RAG System
"""
import os
from pathlib import Path
from pydantic_settings import BaseSettings
from pydantic import Field
from dotenv import load_dotenv

# Load environment variables
load_dotenv()

class Settings(BaseSettings):
    """Application Settings with validation"""

    # API Keys
    fi_api_key: str = Field(..., env='FI_API_KEY')
    fi_secret_key: str = Field(..., env='FI_SECRET_KEY')
    google_api_key: str = Field(..., env='GOOGLE_API_KEY')

    # Project Configuration
    project_name: str = Field(default='gemini-document-qa', env='PROJECT_NAME')
    environment: str = Field(default='development', env='ENVIRONMENT')

    # Model Configuration
    llm_model: str = "gemini-2.0-flash-exp"  # Free tier model
    embedding_model: str = "models/text-embedding-004"  # Gemini embeddings
    eval_model: str = "turing_flash"  # Future AGI evaluation model

    # RAG Configuration
    chunk_size: int = 500  # Tokens per chunk
    chunk_overlap: int = 100  # Overlap between chunks
    top_k_retrieval: int = 3  # Number of chunks to retrieve

    # Generation settings
    temperature: float = 0.7  # Creativity (0=deterministic, 1=creative)
    max_output_tokens: int = 1000  # Max tokens in response

    # Evaluation Configuration
    enable_evaluation: bool = True
    eval_templates: list = [
        "hallucination",
        "relevance",
        "toxicity",
        "tone"
    ]

    # Paths
    base_dir: Path = Path(__file__).parent
    data_dir: Path = base_dir / "data"
    documents_dir: Path = data_dir / "documents"
    chroma_dir: Path = base_dir / "chroma_db"

    class Config:
        env_file = ".env"
        env_file_encoding = "utf-8"

# Initialize settings
try:
    settings = Settings()

    # Create necessary directories
    settings.documents_dir.mkdir(parents=True, exist_ok=True)
    settings.chroma_dir.mkdir(parents=True, exist_ok=True)

    # Set Google API key globally
    os.environ["GOOGLE_API_KEY"] = settings.google_api_key

    print("✅ Configuration loaded successfully")
    print(f"📁 Project: {settings.project_name}")
    print(f"🌍 Environment: {settings.environment}")
    print(f"🤖 LLM Model: {settings.llm_model}")

except Exception as e:
    print(f"❌ Configuration Error: {str(e)}")
    print("Please ensure .env file exists with required variables")
    print("Copy .env.example to .env and fill in your API keys")
    raise

# Export settings
__all__ = ['settings']
```

File 4: `src/__init__.py`

```
python
```

```
"""
Gemini RAG System - Source Package
"""

__version__ = "1.0.0"
__author__ = "Gemini RAG Team"

from .document_processor import DocumentProcessor
from .vector_store import VectorStore
from .rag_engine import RAGEngine
from .evaluator import QualityEvaluator
from .ui import create_interface

__all__ = [
    'DocumentProcessor',
    'VectorStore',
    'RAGEngine',
    'QualityEvaluator',
    'create_interface'
]
```

File 5: `src/document_processor.py`

python

```python
"""
Document Processor Module
Handles loading and chunking of documents (PDF, TXT)
"""

import os
from pathlib import Path
from typing import List, Dict
import tiktoken
from pypdf import PdfReader

class DocumentChunk:
    """Represents a chunk of text from a document"""

    def __init__(self, text: str, metadata: Dict):
        self.text = text
        self.metadata = metadata

    def __repr__(self):
        return f"DocumentChunk(text_length={len(self.text)}, source={self.metadata.get('source')})"


class DocumentProcessor:
    """
    Process documents and split them into semantic chunks
    """

    def __init__(self, chunk_size: int = 500, chunk_overlap: int = 100):
        """
        Initialize document processor

        Args:
            chunk_size: Target size for each chunk (in tokens)
            chunk_overlap: Number of tokens to overlap between chunks
        """
        self.chunk_size = chunk_size
        self.chunk_overlap = chunk_overlap
        self.encoding = tiktoken.get_encoding("cl100k_base")  # GPT-4 tokenizer

    def load_documents(self, directory: Path) -> List[DocumentChunk]:
        """
        Load all documents from a directory

        Args:
            directory: Path to directory containing documents

        Returns:
            List of DocumentChunk objects
        """
        all_chunks = []

        if not directory.exists():
            print(f"⚠️  Directory not found: {directory}")
            return all_chunks

        files = list(directory.glob("*.*"))
        print(f"📁 Found {len(files)} files in {directory}")

        for file_path in files:
            print(f"📄 Processing: {file_path.name}...")

            try:
                if file_path.suffix.lower() == '.pdf':
                    chunks = self._process_pdf(file_path)
                elif file_path.suffix.lower() == '.txt':
                    chunks = self._process_txt(file_path)
                else:
                    print(f"   ⚠️  Skipping unsupported format: {file_path.suffix}")
                    continue

                all_chunks.extend(chunks)
                print(f"   ✅ Created {len(chunks)} chunks")

            except Exception as e:
                print(f"   ❌ Error processing {file_path.name}: {str(e)}")

        print(f"\n✅ Total chunks created: {len(all_chunks)}")
        return all_chunks

    def _process_pdf(self, file_path: Path) -> List[DocumentChunk]:
        """Process a PDF file"""
        chunks = []

        reader = PdfReader(str(file_path))

        for page_num, page in enumerate(reader.pages, 1):
            text = page.extract_text()

            if not text.strip():
                continue

            # Split page into chunks
            page_chunks = self._split_text(
                text,
                metadata={
                    "source": file_path.name,
                    "page": page_num,
                    "total_pages": len(reader.pages)
                }
            )
            chunks.extend(page_chunks)

        return chunks

    def _process_txt(self, file_path: Path) -> List[DocumentChunk]:
        """Process a text file"""
        with open(file_path, 'r', encoding='utf-8') as f:
            text = f.read()

        return self._split_text(
            text,
            metadata={
                "source": file_path.name,
                "type": "text"
            }
        )

    def _split_text(self, text: str, metadata: Dict) -> List[DocumentChunk]:
        """
        Split text into overlapping chunks based on token count
        """
        chunks = []

        # Tokenize the text
        tokens = self.encoding.encode(text)

        # If text is smaller than chunk size, return as single chunk
        if len(tokens) <= self.chunk_size:
            return [DocumentChunk(text, metadata)]

        # Split into overlapping chunks
        start = 0
        chunk_id = 0
        while start < len(tokens):
            # Get chunk tokens
            end = start + self.chunk_size
            chunk_tokens = tokens[start:end]

            # Decode back to text
            chunk_text = self.encoding.decode(chunk_tokens)

            # Create chunk with metadata
            chunk_metadata = metadata.copy()
            chunk_metadata.update({
                "chunk_id": chunk_id,
                "start_token": start,
                "end_token": end,
                "total_tokens": len(tokens)
            })

            chunks.append(DocumentChunk(chunk_text, chunk_metadata))

            # Move start position (with overlap)
            start += (self.chunk_size - self.chunk_overlap)
            chunk_id += 1

        return chunks

    def count_tokens(self, text: str) -> int:
        """Count tokens in text"""
        return len(self.encoding.encode(text))
```

File 6: src/vector_store.py

```
python
```

```python
"""
Vector Store Module
Handles vector database operations using ChromaDB and Gemini embeddings
"""

from typing import List, Dict
from pathlib import Path
import chromadb
from chromadb.config import Settings
import google.generativeai as genai
from .document_processor import DocumentChunk


class VectorStore:
    """
    Manages vector database for semantic search using Gemini embeddings
    """

    def __init__(
        self,
        persist_directory: Path,
        collection_name: str = "documents",
        embedding_model: str = "models/text-embedding-004"
    ):
        """
        Initialize vector store

        Args:
            persist_directory: Where to store the database
            collection_name: Name for the collection
            embedding_model: Gemini embedding model to use
        """
        self.embedding_model = embedding_model

        # Initialize ChromaDB with persistence
        self.client = chromadb.PersistentClient(
            path=str(persist_directory),
            settings=Settings(
                anonymized_telemetry=False,
                allow_reset=True
            )
        )

        # Get or create collection
        self.collection = self.client.get_or_create_collection(
            name=collection_name,
            metadata={"hnsw:space": "cosine"}  # Use cosine similarity
        )

        print(f"📊 Vector Store initialized")
        print(f"   Collection: {collection_name}")
        print(f"   Documents: {self.collection.count()}")

    def add_documents(self, chunks: List[DocumentChunk]) -> None:
        """
        Add document chunks to vector store

        Args:
            chunks: List of DocumentChunk objects
        """
        if not chunks:
            print("⚠️  No chunks to add")
            return

        print(f"\n🔄 Adding {len(chunks)} chunks to vector store...")

        # Prepare data for ChromaDB
        documents = []
        embeddings = []
        metadatas = []
        ids = []

        for i, chunk in enumerate(chunks):
            # Generate embedding using Gemini
            embedding = self._get_embedding(chunk.text)

            # Prepare data
            documents.append(chunk.text)
            embeddings.append(embedding)
            metadatas.append(chunk.metadata)
            ids.append(f"chunk_{i}")

            # Progress indicator
            if (i + 1) % 10 == 0:
                print(f"   Processed {i + 1}/{len(chunks)} chunks...")

        # Add to ChromaDB
        self.collection.add(
            documents=documents,
```

```python
            embeddings=embeddings,
            metadatas=metadatas,
            ids=ids
        )

        print(f"✅ Added {len(chunks)} chunks successfully")
        print(f"   Total in database: {self.collection.count()}")

    def search(self, query: str, top_k: int = 3) -> List[Dict]:
        """
        Search for relevant document chunks using Gemini embeddings

        Args:
            query: User's question
            top_k: Number of results to return

        Returns:
            List of dicts with text, metadata, and similarity score
        """
        if self.collection.count() == 0:
            print("⚠️  No documents in vector store")
            return []

        # Generate query embedding using Gemini
        query_embedding = self._get_embedding(query)

        # Search in ChromaDB
        results = self.collection.query(
            query_embeddings=[query_embedding],
            n_results=top_k,
            include=["documents", "metadatas", "distances"]
        )

        # Format results
        formatted_results = []
        for i in range(len(results['documents'][0])):
            formatted_results.append({
                'text': results['documents'][0][i],
                'metadata': results['metadatas'][0][i],
                'similarity': 1 - results['distances'][0][i]  # Convert distance to similarity
            })

        return formatted_results

    def _get_embedding(self, text: str) -> List[float]:
        """
        Generate embedding for text using Gemini API

        Args:
            text: Text to embed

        Returns:
            Embedding vector (list of floats)
        """
        result = genai.embed_content(
            model=self.embedding_model,
            content=text,
            task_type="retrieval_document"
        )
        return result['embedding']

    def clear(self) -> None:
        """Clear all documents from collection"""
        ids = self.collection.get()['ids']
        if ids:
            self.collection.delete(ids=ids)
            print(f"🗑️ Cleared {len(ids)} documents from vector store")

    def get_stats(self) -> Dict:
        """Get statistics about the vector store"""
        return {
            'total_documents': self.collection.count(),
            'collection_name': self.collection.name,
            'embedding_model': self.embedding_model
        }
```

File 7: `src/rag_engine.py`

python

```python
"""
RAG Engine Module
Orchestrates retrieval and generation with Future AGI TraceAI instrumentation
Uses Google Gemini 2.0 Flash for generation
"""

import os
from typing import Dict, List
import google.generativeai as genai
from traceai_google_genai import GoogleGenAIInstrumentor
from fi_instrumentation import register
from .vector_store import VectorStore
from config import settings


class RAGEngine:
    """
    Retrieval-Augmented Generation Engine with Google Gemini and TraceAI
    """

    def __init__(self, vector_store: VectorStore):
        """
        Initialize RAG engine with tracing

        Args:
            vector_store: Vector store for document retrieval
        """
        self.vector_store = vector_store

        # Configure Gemini
        genai.configure(api_key=settings.google_api_key)
        self.model = genai.GenerativeModel(settings.llm_model)

        # ================================================
        # FUTURE AGI TRACING SETUP
        # ================================================

        print("\n🔍 Setting up Future AGI TraceAI...")

        # Step 1: Register project with Future AGI
        trace_provider = register(
            project_name=settings.project_name,
            fi_api_key=settings.fi_api_key,
            fi_secret_key=settings.fi_secret_key,
        environment=settings.environment
    )

    # Step 2: Instrument Google GenAI SDK
    GoogleGenAIInstrumentor().instrument(tracer_provider=trace_provider)

    print("✅ TraceAI instrumentation complete!")
    print(f"   Project: {settings.project_name}")
    print(f"   View traces at: https://app.futureagi.com")

    # ============================================
    # All Gemini calls now automatically traced!
    # ============================================

  def query(self, user_question: str, top_k: int = 3) -> Dict:
    """
    Process a user question and generate an answer

    Args:
        user_question: The user's question
        top_k: Number of document chunks to retrieve

    Returns:
        Dict with answer, sources, and metadata
    """

    # Step 1: Retrieve relevant documents
    print(f"\n🔍 Retrieving relevant documents...")
    relevant_chunks = self.vector_store.search(user_question, top_k=top_k)

    if not relevant_chunks:
        return {
            'answer': "I don't have enough information to answer that question.",
            'sources': [],
            'metadata': {
                'chunks_retrieved': 0,
                'model': settings.llm_model
            }
        }

    print(f"   Found {len(relevant_chunks)} relevant chunks")

    # Step 2: Format context
    context = self._format_context(relevant_chunks)
```

```
        # Step 3: Generate answer with Gemini
        # This Gemini call is AUTOMATICALLY TRACED by TraceAI!
        print(f"🤖 Generating answer with {settings.llm_model}...")

        prompt = f"""{self._get_system_prompt()}

Context from documents:
{context}

Question: {user_question}

Please provide a clear and accurate answer based on the context above.
If the context doesn't contain enough information, say so.
Always cite which document/page your information comes from."""

        response = self.model.generate_content(
            prompt,
            generation_config=genai.types.GenerationConfig(
                temperature=settings.temperature,
                max_output_tokens=settings.max_output_tokens
            )
        )

        answer = response.text

        # Extract usage metadata (if available)
        usage_metadata = getattr(response, 'usage_metadata', None)
        prompt_tokens = getattr(usage_metadata, 'prompt_token_count', 0) if usage_metadata else 0
        completion_tokens = getattr(usage_metadata, 'candidates_token_count', 0) if usage_metadata else 0
        total_tokens = prompt_tokens + completion_tokens

        # Step 4: Format response
        result = {
            'answer': answer,
            'sources': self._format_sources(relevant_chunks),
            'metadata': {
                'chunks_retrieved': len(relevant_chunks),
                'model': settings.llm_model,
                'temperature': settings.temperature,
                'prompt_tokens': prompt_tokens,
                'completion_tokens': completion_tokens,
                'total_tokens': total_tokens,
                'cost_estimate': self._calculate_cost(total_tokens)
            }
        }
        print(f"✅ Answer generated!")
        print(f"   Tokens used: {total_tokens}")
        print(f"   Estimated cost: ${result['metadata']['cost_estimate']:.6f}")

        return result

    def _format_context(self, chunks: List[Dict]) -> str:
        """Format retrieved chunks into context string"""
        context_parts = []

        for i, chunk in enumerate(chunks, 1):
            source = chunk['metadata'].get('source', 'Unknown')
            page = chunk['metadata'].get('page', 'N/A')

            context_parts.append(
                f"[Source {i}: {source}, Page {page}]\n{chunk['text']}\n"
            )

        return "\n".join(context_parts)

    def _format_sources(self, chunks: List[Dict]) -> List[Dict]:
        """Format source information for response"""
        sources = []

        for i, chunk in enumerate(chunks, 1):
            sources.append({
                'source_id': i,
                'filename': chunk['metadata'].get('source', 'Unknown'),
                'page': chunk['metadata'].get('page', 'N/A'),
                'similarity': f"{chunk.get('similarity', 0):.3f}",
                'text_preview': chunk['text'][:200] + "..."
            })

        return sources

    def _get_system_prompt(self) -> str:
        """Get system prompt for LLM"""
        return """You are a helpful AI assistant that answers questions based on provided documents.

Guidelines:
- Provide accurate, concise answers based on the context
- Always cite your sources (mention the document and page)
- If information is not in the context, say "I don't have enough information"
- Be objective and factual
- Use clear, professional language"""

    def _calculate_cost(self, total_tokens: int) -> float:
        """
        Calculate estimated cost for Gemini API call

        Gemini 2.0 Flash pricing (as of 2024):
        - Free tier: 15 RPM, 1M TPM, 1500 RPD
        - Paid: $0.075 per 1M input tokens, $0.30 per 1M output tokens

        For simplicity, we'll use average rate
        """
        # Average cost per 1M tokens
        avg_cost_per_million = 0.1875  # ($0.075 + $0.30) / 2

        return (total_tokens / 1_000_000) * avg_cost_per_million
```

File 8: src/evaluator.py

```
python
```

```python
"""
Quality Evaluator Module
Automated evaluation of RAG outputs using Future AGI's AI Evaluation SDK
"""

from typing import Dict, List
from fi.evals import Evaluator
from config import settings


class QualityEvaluator:
    """
    Automated Quality Evaluation for RAG Outputs
    """

    def __init__(self):
        """Initialize evaluator with Future AGI credentials"""
        self.evaluator = Evaluator(
            fi_api_key=settings.fi_api_key,
            fi_secret_key=settings.fi_secret_key
        )

        self.eval_templates = settings.eval_templates

        print("\n🔍 Quality Evaluator initialized")
        print(f"   Evaluation templates: {', '.join(self.eval_templates)}")

    def evaluate_response(
        self,
        question: str,
        answer: str,
        context: str,
        sources: List[Dict] = None
    ) -> Dict:
        """
        Evaluate a RAG response for quality

        Args:
            question: User's original question
            answer: Generated answer
            context: Retrieved document context
            sources: List of source documents (optional)

        Returns:
```

```
        Dict with evaluation results and explanations
    """
    if not settings.enable_evaluation:
        return self._mock_evaluation()

    print(f"\n📊 Running quality evaluation...")

    try:
        # Run evaluation with Future AGI
        result = self.evaluator.evaluate(
            eval_templates=self.eval_templates,
            inputs={
                "input": question,
                "output": answer,
                "context": context
            },
            model_name=settings.eval_model
        )

        # Format results
        eval_results = {}

        for eval_result in result.eval_results:
            template_name = eval_result.template

            eval_results[template_name] = {
                'score': eval_result.output,
                'passed': eval_result.passed,
                'reason': eval_result.reason,
                'severity': self._get_severity(template_name, eval_result.passed)
            }

            # Print results
            status = "✅ PASS" if eval_result.passed else "❌ FAIL"
            print(f"   {template_name}: {status}")
            if not eval_result.passed:
                print(f"      Reason: {eval_result.reason}")

        return {
            'overall_passed': all(r['passed'] for r in eval_results.values()),
            'evaluations': eval_results,
            'summary': self._create_summary(eval_results)
        }

    except Exception as e:
            print(f"⚠️  Evaluation error: {str(e)}")
            return self._mock_evaluation()

    def _get_severity(self, template_name: str, passed: bool) -> str:
        """Get severity level for failed evaluations"""
        if passed:
            return "none"

        # Critical failures
        if template_name in ["hallucination", "toxicity", "pii_detection"]:
            return "critical"

        # Important failures
        if template_name in ["relevance", "factual_accuracy"]:
            return "high"

        # Minor issues
        return "medium"

    def _create_summary(self, eval_results: Dict) -> str:
        """Create human-readable summary"""
        total = len(eval_results)
        passed = sum(1 for r in eval_results.values() if r['passed'])

        if passed == total:
            return f"✅ All {total} quality checks passed!"

        failed_checks = [
            name for name, result in eval_results.items()
            if not result['passed']
        ]

        return f"⚠️  {len(failed_checks)} of {total} checks failed: {', '.join(failed_checks)}"

    def _mock_evaluation(self) -> Dict:
        """Mock evaluation for testing when evaluation is disabled"""
        return {
            'overall_passed': True,
            'evaluations': {
                template: {
                    'score': 'PASS',
                    'passed': True,
                    'reason': 'Evaluation disabled',
                    'severity': 'none'
                }
            for template in self.eval_templates
        },
        'summary': '⚠️  Evaluation disabled'
    }

def format_evaluation_for_ui(self, evaluation: Dict) -> str:
    """
    Format evaluation results for display in UI

    Args:
        evaluation: Evaluation results dict

    Returns:
        Formatted string for UI display
    """
    lines = []
    lines.append("\n📊 **Quality Evaluation:**\n")
    lines.append(f"{evaluation['summary']}\n")

    for name, result in evaluation['evaluations'].items():
        status_icon = "✅" if result['passed'] else "❌"
        lines.append(f"{status_icon} **{name.upper()}**: {result['score']}")

        if not result['passed']:
            lines.append(f"   *Reason*: {result['reason']}")
            lines.append(f"   *Severity*: {result['severity']}")

    return "\n".join(lines)
```

File 9: `src/ui.py`

python

```python
"""
User Interface Module
Gradio-based web interface for the RAG system
"""

import gradio as gr
from typing import Tuple
from .rag_engine import RAGEngine
from .evaluator import QualityEvaluator


class RAGUI:
    """Gradio UI for Document Q&A System"""

    def __init__(self, rag_engine: RAGEngine, evaluator: QualityEvaluator):
        """
        Initialize UI with RAG engine and evaluator

        Args:
            rag_engine: RAG engine for question answering
            evaluator: Quality evaluator
        """
        self.rag_engine = rag_engine
        self.evaluator = evaluator

    def process_question(self, question: str, top_k: int) -> Tuple[str, str, str, str]:
        """
        Process a user question and return formatted results

        Args:
            question: User's question
            top_k: Number of chunks to retrieve

        Returns:
            Tuple of (answer, sources, evaluation, metadata)
        """
        if not question.strip():
            return (
                "Please enter a question.",
                "",
                "",
                ""
            )
        try:
            # Get answer from RAG engine
            result = self.rag_engine.query(question, top_k=top_k)

            # Format answer
            answer = result['answer']

            # Format sources
            sources = self._format_sources(result['sources'])

            # Run evaluation
            context = self._extract_context(result['sources'])
            evaluation_result = self.evaluator.evaluate_response(
                question=question,
                answer=answer,
                context=context,
                sources=result['sources']
            )
            evaluation = self.evaluator.format_evaluation_for_ui(evaluation_result)

            # Format metadata
            metadata = self._format_metadata(result['metadata'])

            return answer, sources, evaluation, metadata

        except Exception as e:
            error_msg = f"❌ Error: {str(e)}"
            return error_msg, "", "", ""

    def _format_sources(self, sources: list) -> str:
        """Format sources for display"""
        if not sources:
            return "No sources found."

        formatted = ["## 📚 Sources\n"]

        for source in sources:
            formatted.append(f"""
### Source {source['source_id']}: {source['filename']}
- **Page**: {source['page']}
- **Similarity**: {source['similarity']}

**Text Preview:**
> {source['text_preview']}
---
""")

        return "\n".join(formatted)

    def _format_metadata(self, metadata: dict) -> str:
        """Format metadata for display"""
        return f"""
## 📊 Request Metadata

- **Model**: {metadata['model']}
- **Temperature**: {metadata['temperature']}
- **Chunks Retrieved**: {metadata['chunks_retrieved']}

### Token Usage:
- **Prompt Tokens**: {metadata['prompt_tokens']}
- **Completion Tokens**: {metadata['completion_tokens']}
- **Total Tokens**: {metadata['total_tokens']}

### Cost:
- **Estimated Cost**: ${metadata['cost_estimate']:.6f}
"""

    def _extract_context(self, sources: list) -> str:
        """Extract context text from sources"""
        return "\n\n".join([s['text_preview'] for s in sources])

    def create_interface(self) -> gr.Blocks:
        """Create Gradio interface"""
        with gr.Blocks(
            title="Gemini Document Q&A with Future AGI",
            theme=gr.themes.Soft()
        ) as interface:

            gr.Markdown("""
            # 📚 Gemini Document Q&A System
            ### Powered by Google Gemini 2.0 Flash & Future AGI

            Ask questions about your documents and get AI-powered answers with:
            - ✅ Automatic quality evaluation
            - 🔍 Complete tracing and observability
            - 📊 Source citations
            - 💰 Cost tracking (FREE with Gemini!)
            """)
```

```
with gr.Row():
    with gr.Column(scale=2):
        # Input section
        question_input = gr.Textbox(
            label="Your Question",
            placeholder="What would you like to know about your documents?",
            lines=3
        )

        with gr.Row():
            top_k_slider = gr.Slider(
                minimum=1,
                maximum=10,
                value=3,
                step=1,
                label="Number of chunks to retrieve",
                info="Higher = more context but slower"
            )

        submit_btn = gr.Button(
            "🔍 Ask Question",
            variant="primary",
            size="lg"
        )

        gr.Markdown("""
        ---
        **💡 Tips:**
        - Be specific in your questions
        - Ask about content in your uploaded documents
        - Check the evaluation results for answer quality
        - Using FREE Google Gemini 2.0 Flash!
        """)

# Output section
with gr.Column(scale=3):
    answer_output = gr.Markdown(
        label="Answer",
        value=""
    )

    with gr.Tabs():
        with gr.Tab("📚 Sources"):
            sources_output = gr.Markdown(value="")
            with gr.Tab("📊 Quality Evaluation"):
                evaluation_output = gr.Markdown(value="")

            with gr.Tab("📈 Metadata"):
                metadata_output = gr.Markdown(value="")

        # Connect components
        submit_btn.click(
            fn=self.process_question,
            inputs=[question_input, top_k_slider],
            outputs=[
                answer_output,
                sources_output,
                evaluation_output,
                metadata_output
            ]
        )

        # Allow Enter key to submit
        question_input.submit(
            fn=self.process_question,
            inputs=[question_input, top_k_slider],
            outputs=[
                answer_output,
                sources_output,
                evaluation_output,
                metadata_output
            ]
        )

        # Footer
        gr.Markdown("""
        ---
        ### 🔍 Observability
        View detailed traces and metrics at: [Future AGI Dashboard](https://app.futureagi.com)

        ### 📖 Features
        - **Google Gemini 2.0 Flash**: FREE, fast, high-quality AI
        - **TraceAI**: Automatic tracing for all operations
        - **AI Evaluation**: Automated quality checks (hallucination, relevance, toxicity)
        - **Vector Search**: Semantic search with ChromaDB
        - **Cost Tracking**: Know exactly what you're spending (virtually nothing!)
        """)

    return interface
```

```python
def create_interface(rag_engine: RAGEngine, evaluator: QualityEvaluator) -> gr.Blocks:
    """
    Convenience function to create UI

    Args:
        rag_engine: RAG engine instance
        evaluator: Quality evaluator instance

    Returns:
        Gradio interface
    """
    ui = RAGUI(rag_engine, evaluator)
    return ui.create_interface()
```

File 10: main.py

python

```python
"""
Main Application Entry Point
Initializes and runs the Gemini Document Q&A System with Future AGI
"""

import os
import sys
from pathlib import Path

# Add project root to path
sys.path.insert(0, str(Path(__file__).parent))

from config import settings
from src import (
    DocumentProcessor,
    VectorStore,
    RAGEngine,
    QualityEvaluator,
    create_interface
)


def initialize_system():
    """Initialize all components of the RAG system"""
    print("\n" + "=" * 60)
    print("🚀 Initializing Gemini Document Q&A System")
    print("=" * 60)

    # Step 1: Initialize Document Processor
    print("\n📄 Step 1: Initializing Document Processor...")
    doc_processor = DocumentProcessor(
        chunk_size=settings.chunk_size,
        chunk_overlap=settings.chunk_overlap
    )

    # Step 2: Load and Process Documents
    print("\n📁 Step 2: Loading Documents...")
    chunks = doc_processor.load_documents(settings.documents_dir)

    if not chunks:
        print("\n⚠️  WARNING: No documents found!")
        print(f"   Please add PDF or TXT files to: {settings.documents_dir}")
        print(f"   Or run: python add_sample_docs.py")
        print("\n   The system will start, but you won't be able to ask questions.")
        input("\nPress Enter to continue anyway...")

    # Step 3: Initialize Vector Store
    print("\n🗄️  Step 3: Initializing Vector Store...")
    vector_store = VectorStore(
        persist_directory=settings.chroma_dir,
        collection_name=settings.project_name,
        embedding_model=settings.embedding_model
    )

    # Step 4: Add Documents to Vector Store (if new)
    if chunks:
        existing_count = vector_store.collection.count()
        if existing_count == 0:
            print("\n📥 Step 4: Adding documents to vector store...")
            vector_store.add_documents(chunks)
        else:
            print(f"\n✅ Step 4: Using existing vector store ({existing_count} documents)")
            print("   To reindex: Delete the 'chroma_db' folder and restart")

    # Step 5: Initialize RAG Engine with TraceAI
    print("\n🤖 Step 5: Initializing RAG Engine...")
    rag_engine = RAGEngine(vector_store)

    # Step 6: Initialize Quality Evaluator
    print("\n📊 Step 6: Initializing Quality Evaluator...")
    evaluator = QualityEvaluator()

    print("\n" + "=" * 60)
    print("✅ System Initialization Complete!")
    print("=" * 60)

    return rag_engine, evaluator


def print_system_info():
    """Print system information and helpful links"""
    print("\n" + "=" * 60)
    print("📊 SYSTEM INFORMATION")
    print("=" * 60)
    print(f"\n📁 Project: {settings.project_name}")
    print(f"🌍 Environment: {settings.environment}")
    print(f"🤖 LLM Model: {settings.llm_model}")
    print(f"📏 Chunk Size: {settings.chunk_size} tokens")
    print(f"🔍 Top-K Retrieval: {settings.top_k_retrieval}")
    print(f"📊 Evaluation: {'Enabled' if settings.enable_evaluation else 'Disabled'}")

    print("\n" + "=" * 60)
    print("🔗 USEFUL LINKS")
    print("=" * 60)
    print("\n🔍 View Traces:")
    print(f"   https://app.futureagi.com/projects/{settings.project_name}/traces")
    print("\n📊 View Evaluations:")
    print(f"   https://app.futureagi.com/projects/{settings.project_name}/evaluations")
    print("\n🔑 Get Gemini API Key:")
    print("   https://aistudio.google.com/app/apikey")
    print("\n📚 Documentation:")
    print("   https://docs.futureagi.com")

    print("\n" + "=" * 60)
    print("💡 TIPS")
    print("=" * 60)
    print("""
1. Add documents to: data/documents/
2. Ask specific questions about your documents
3. Check evaluation results for answer quality
4. View traces in Future AGI dashboard
5. Monitor costs (virtually FREE with Gemini!)
6. Adjust chunk_size in config.py for better results
""")


def main():
    """Main application entry point"""
    try:
        # Initialize system
        rag_engine, evaluator = initialize_system()

        # Print system info
        print_system_info()

        # Create and launch Gradio interface
        print("\n" + "=" * 60)
        print("🌐 Starting Web Interface...")
        print("=" * 60)

        interface = create_interface(rag_engine, evaluator)

        print("\n✅ Application is running!")
        print("   Open in browser: http://localhost:7860")
        print("\n⏸️  Press Ctrl+C to stop\n")

        # Launch interface
        interface.launch(
            server_name="0.0.0.0",
            server_port=7860,
            share=False,
            show_error=True
        )

    except KeyboardInterrupt:
        print("\n\n👋 Shutting down gracefully...")
        print("✅ Application stopped")

    except Exception as e:
        print(f"\n❌ Fatal Error: {str(e)}")
        print("\nTroubleshooting:")
        print("1. Check .env file has correct API keys")
        print("2. Ensure all dependencies are installed: pip install -r requirements.txt")
        print("3. Check documents exist in: data/documents/")
        print("4. Get FREE Gemini API key: https://aistudio.google.com/app/apikey")
        print("\nFor help, see: https://docs.futureagi.com")
        sys.exit(1)


if __name__ == "__main__":
    main()
```

File 11: `add_sample_docs.py`

python

```python
"""
Helper Script to Add Sample Documents
Creates sample TXT files for testing the system
"""

from pathlib import Path


def create_sample_documents():
    """Create sample documents in data/documents/"""

    documents_dir = Path("data/documents")
    documents_dir.mkdir(parents=True, exist_ok=True)

    print("📄 Creating sample documents...")

    # Sample 1: AI Basics (TXT)
    ai_content = """
Artificial Intelligence (AI) Overview

What is Artificial Intelligence?
Artificial Intelligence (AI) refers to the simulation of human intelligence in machines
that are programmed to think and learn like humans. AI systems can perform tasks that
typically require human intelligence, such as visual perception, speech recognition,
decision-making, and language translation.

Types of AI:
1. Narrow AI (Weak AI): AI systems designed for specific tasks, such as voice assistants,
   recommendation systems, or image recognition.

2. General AI (Strong AI): Hypothetical AI systems that possess the ability to understand,
   learn, and apply intelligence across a wide range of tasks, similar to human intelligence.

3. Superintelligent AI: A theoretical form of AI that surpasses human intelligence in all aspects.

Machine Learning:
Machine Learning (ML) is a subset of AI that enables systems to learn and improve from
experience without being explicitly programmed. ML algorithms use statistical techniques
to identify patterns in data and make predictions or decisions.

Common ML algorithms include:
- Linear Regression
- Decision Trees
- Neural Networks
- Support Vector Machines
- Random Forests

Applications of AI:
- Healthcare: Disease diagnosis, drug discovery, personalized medicine
- Finance: Fraud detection, algorithmic trading, risk assessment
- Transportation: Autonomous vehicles, traffic optimization
- Customer Service: Chatbots, virtual assistants
- Manufacturing: Predictive maintenance, quality control
- Education: Personalized learning, automated grading

Future of AI:
The future of AI holds immense potential. Advancements in deep learning, natural language
processing, and computer vision are enabling more sophisticated AI applications.
"""

    with open(documents_dir / "ai_basics.txt", "w") as f:
        f.write(ai_content)
    print("   ✅ Created: ai_basics.txt")

    # Sample 2: Python Programming (TXT)
    python_content = """
Python Programming Language Guide

Introduction to Python:
Python is a high-level, interpreted programming language known for its simplicity,
readability, and versatility. Created by Guido van Rossum and first released in 1991,
Python has become one of the most popular programming languages in the world.

Key Features of Python:
1. Easy to Learn: Python's syntax is clean and easy to understand.
2. Interpreted Language: Python code is executed line by line.
3. Dynamic Typing: Variable types are determined at runtime.
4. Extensive Libraries: Vast ecosystem of libraries and frameworks.
5. Cross-Platform: Runs on Windows, macOS, Linux.

Popular Python Libraries:
- NumPy: Numerical computing and array operations
- Pandas: Data manipulation and analysis
- Matplotlib: Data visualization and plotting
- Scikit-learn: Machine learning algorithms
- TensorFlow/PyTorch: Deep learning frameworks
- Django/Flask: Web development frameworks

Common Use Cases:
```

```
1. Data Science and Analytics
2. Web Development
3. Machine Learning and AI
4. Automation and Scripting
5. Scientific Computing

Python continues to evolve with regular updates, making it a future-proof choice.
"""

    with open(documents_dir / "python_guide.txt", "w") as f:
        f.write(python_content)
    print("  ✅ Created: python_guide.txt")

    # Sample 3: RAG Systems (TXT)
    rag_content = """
Retrieval-Augmented Generation (RAG) Systems

What is RAG?
Retrieval-Augmented Generation (RAG) is a technique that combines information retrieval
with large language model generation to produce more accurate, grounded, and contextually
relevant responses.

How RAG Works:
1. Document Ingestion: Load and process documents
2. Query Processing: Convert user query to embedding
3. Retrieval: Search for semantically similar documents
4. Context Formation: Combine retrieved documents
5. Generation: LLM generates answer based on context
6. Response: Return answer with source citations

Benefits of RAG:
- Reduces hallucinations by grounding responses in source documents
- Enables LLMs to access up-to-date information
- Provides source attribution and citations
- More cost-effective than fine-tuning

Best Practices:
1. Optimize chunk size for your use case
2. Use chunk overlap to preserve context
3. Implement evaluation and monitoring
4. Add metadata for better filtering
5. Monitor for hallucinations
"""

    with open(documents_dir / "rag_systems.txt", "w") as f:
        f.write(rag_content)
    print("  ✅ Created: rag_systems.txt")

    print("\n✅ Sample documents created successfully!")
    print(f"  Location: {documents_dir.absolute()}")
    print("\nYou can now run: python main.py")


if __name__ == "__main__":
    create_sample_documents()
```

File 12: `.gitignore`

```
# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
env/
venv/
ENV/
build/

# Environment Variables
.env
.env.local

# IDEs
.vscode/
.idea/
*.swp
.DS_Store

# ChromaDB
chroma_db/
*.db
*.sqlite3

# Logs
*.log
logs/
```

```
# Testing
.pytest_cache/
.coverage

# Gradio
flagged/
```

File 13: README.md

```
markdown
```

~~~markdown
# Gemini Document Q&A System with Future AGI

AI-Powered document question-answering using Google Gemini 2.0 Flash with automatic observability and evaluation.

## Features

✅ **FREE Google Gemini 2.0 Flash** - Fast, high-quality, generous free tier
✅ **Automatic Observability** - TraceAI for complete tracing
✅ **Quality Evaluation** - Automated checks for hallucination, relevance, toxicity
✅ **Document Processing** - PDF/TXT ingestion with chunking
✅ **Vector Search** - ChromaDB for semantic search
✅ **Web Interface** - Gradio UI for easy interaction
✅ **Cost Tracking** - Know exactly what you're spending (virtually nothing!)

## Quick Start

### 1. Install Dependencies
```bash
pip install -r requirements.txt
```


### 2. Get API Keys

**Google Gemini (FREE):**
- Go to: https://aistudio.google.com/app/apikey
- Click "Create API key"
- Copy your key

**Future AGI:**
- Go to: https://app.futureagi.com/settings/api-keys
- Copy your API key and Secret key

### 3. Configure Environment
```bash
cp .env.example .env
# Edit .env and add your API keys
```


### 4. Add Documents
```bash
# Add sample documents
python add_sample_docs.py

# Or add your own PDF/TXT files to data/documents/
~~~

~~~
```

### 5. Run!
```bash
python main.py
```

Open http://localhost:7860 in your browser!

## View Traces

Go to https://app.futureagi.com to see:
- Complete request traces
- Token usage
- Cost tracking
- Evaluation results
- Performance metrics

## Cost

**Google Gemini 2.0 Flash FREE Tier:**
- 15 requests per minute
- 1 million tokens per minute
- 1,500 requests per day

For most personal use, you'll stay within the free tier!

## Support

- Google Gemini Docs: https://ai.google.dev/docs
- Future AGI Docs: https://docs.futureagi.com
- Issues: Open a GitHub issue
~~~

Installation Instructions

Step 1: Create Project Directory

```bash
mkdir gemini-rag-project
cd gemini-rag-project
```

Step 2: Create Files

Copy each file section above into its respective file:

- Create `requirements.txt`
- Create `.env.example`
- Create `config.py`
- Create `src/` directory
- Create all files in `src/`
- Create `main.py`
- Create `add_sample_docs.py`
- Create `.gitignore`
- Create `README.md`

Step 3: Setup Virtual Environment

```bash
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

Step 4: Install Dependencies

```bash
pip install -r requirements.txt
```

Step 5: Configure

```bash
cp .env.example .env
# Edit .env with your API keys
```

Step 6: Add Sample Documents

```bash
python add_sample_docs.py
```

Step 7: Run!

```bash
python main.py
```

That's It!

You now have all the code files needed to run the Gemini RAG project with Future AGI! 🚀