Quantum Tutor - GRM Project

This document summarizes the GRM Project

in three parts

A)BeeAI agent

B)RAG agent

C)Quality Test

A)BeeAI agent

Cell 1: Clone repositories and setup

!git clone https://github.com/i-am-bee/beeai-framework.git
%cd beeai-framework

Clones the repository

```
bash
CopyEdit
git clone https://github.com/i-am-bee/beeai-framework.git
```

- Downloads the `beeai-framework` repository into your current working directory.

Changes the working directory

```
python
CopyEdit
%cd beeai-framework
```

- Moves your working directory into the newly cloned `beeai-framework` folder so subsequent commands (like installs or running scripts) happen in the right place.

Cell 2: Install dependencies (updated for Groq)

```
bash
CopyEdit
!pip install beeai qiskit python-dotenv groq requests xml
```

What it does:

Installs these Python packages into your environment:

- **beeai** – the BeeAI framework (core library).
- **qiskit** – IBM's quantum computing SDK.
- **python-dotenv** – load environment variables from `.env` files.
- **groq** – Groq API client.
- **requests** – for making HTTP requests.
- **xml** – *(Note: this does not exist as a PyPI package; it will error)*

Cell 3 : Environmental variables

```
# Set environment variables
os.environ["IBM_QUANTUM_API_TOKEN"] = "1y7ug1tjAEHZkLwp_vXX8S1bqXy
os.environ["ARXIV_API_BASE"] = "http://export.arxiv.org/api/query"
os.environ["GROQ_API_KEY"] = "gsk_b7nGx5Otfk36gpSLyXyoWGdyb3FYqQ0
```

Set environment variables

```
python
CopyEdit
os.environ["IBM_QUANTUM_API_TOKEN"] = "1y7ug1tjAEHZkLwp_vXX8S1b
qXyZrIVrR_g8Mx3qzghc"
```

Stores your **IBM Quantum API token** in the environment so your code (like Qiskit) can authenticate with IBM Quantum services.

```
python
CopyEdit
os.environ["ARXIV_API_BASE"] = "http://export.arxiv.org/api/query"
```

Sets the **base URL for the arXiv API**, which you'll use to query scientific papers.

```
python
CopyEdit
os.environ["GROQ_API_KEY"] = "gsk_b7nGx5Otfk36gpSLyXyoWGdyb3FYq
GxJGc7H63gdNob3X8h"
```

Stores your **Groq API key** so your code can connect to Groq's services.

Cell 4 : Writes to .env files

```
# Write to .env file
with open(".env", "w") as f:
    f.write(f"IBM_QUANTUM_API_TOKEN={os.environ['IBM_QUANTUM_API_TO
    f.write(f"ARXIV_API_BASE={os.environ['ARXIV_API_BASE']}\n")
    f.write(f"GROQ_API_KEY={os.environ['GROQ_API_KEY']}\n")

load_dotenv()

print("✅ Loaded tokens:")
print("IBM:", os.getenv("IBM_QUANTUM_API_TOKEN")[:10], "...")
print("arXiv API Base:", os.getenv("ARXIV_API_BASE"))
print("Groq API Key:", os.getenv("GROQ_API_KEY")[:10], "...")
```

Cell 5 : pip install qiskit

- `pip install` installs the **Qiskit IBM Runtime** package.
- `q` makes the output **quiet**, so you see minimal logs.

What is `qiskit-ibm-runtime`?

This is the **Qiskit module that lets you run quantum programs on IBM Quantum systems more efficiently**, including:

- Managed execution in IBM Cloud.
- Session management to reduce latency.
- Access to newer backends and runtime primitives (like `Sampler` and `Estimator`).

Cell 6 : Test IBM Quantum and arXiv connections

```
import os
from dotenv import load_dotenv
from qiskit_ibm_runtime import QiskitRuntimeService
import requests
import xml.etree.ElementTree as ET

load_dotenv()
```

```
python
CopyEdit
import os
```

- Lets you work with environment variables and file paths.

```
python
CopyEdit
from dotenv import load_dotenv
```

- Imports `load_dotenv`, which loads environment variables from a `.env` file into `os.environ`.

```
python
CopyEdit
```

```
from qiskit_ibm_runtime import QiskitRuntimeService
```

- Imports `QiskitRuntimeService`, which connects to IBM Quantum's runtime for running quantum circuits.

```
python
CopyEdit
import requests
```

- Imports the HTTP library `requests` for making API calls (e.g., to arXiv).

```
python
CopyEdit
import xml.etree.ElementTree as ET
```

- Imports `ElementTree`, a module for parsing XML (like responses from arXiv API).

Then:

```
python
CopyEdit
load_dotenv()
```

- Loads any environment variables defined in a `.env` file in your project directory, so you don't have to hardcode tokens in your script.

Cell 7 : Set arxiv and ibm quantum learning api

```
import os
import requests
import xml.etree.ElementTree as ET
from qiskit_ibm_runtime import QiskitRuntimeService

# === IBM QUANTUM BACKENDS (mimicking arXiv-style try-except) ===
api_token = os.getenv("IBM_QUANTUM_API_TOKEN")
```

```
try:
    service = QiskitRuntimeService(channel="ibm_quantum", token=api_token)
    print(" 🔹 Available IBM Quantum backends:")
    for backend in service.backends():
        print("-", backend.name)

except Exception as e:
    print(f"❌ Error loading IBM Quantum service: {e}")
    print("🔄 Switching to fallback resources...")
    print("🔗 IBM Quantum Login: https://cloud.ibm.com/quantum")
    print("📘 IBM Quantum Learning Portal: https://quantum-computing.ibm.co
    print("📚 Qiskit Textbook: https://qiskit.org/learn/")
    print("🧪 Quantum Lab: https://quantum-computing.ibm.com/")

# === arXiv QUERY (Quantum papers) ===
arxiv_base = os.getenv("ARXIV_API_BASE", "http://export.arxiv.org/api/query"
search_query = "quantum+computing"
max_results = 3

arxiv_url = f"{arxiv_base}?search_query=all:{search_query}&start=0&max_res

try:
    response = requests.get(arxiv_url)
    root = ET.fromstring(response.content)

    print("\n 🔹 Recent arXiv results (quantum computing):")
    for entry in root.findall('{http://www.w3.org/2005/Atom}entry'):
        title = entry.find('{http://www.w3.org/2005/Atom}title').text.strip()
        link = entry.find('{http://www.w3.org/2005/Atom}id').text.strip()
        print(f"- {title}\n  ↪ {link}")
except Exception as e:
    print(f"❌ Error querying arXiv: {e}")
```

Imports

```
python
CopyEdit
import os
import requests
import xml.etree.ElementTree as ET
from qiskit_ibm_runtime import QiskitRuntimeService
```

Purpose:

- `os`: get environment variables (`os.getenv`)
- `requests`: make HTTP requests to APIs (like arXiv)
- `ElementTree`: parse XML responses
- `QiskitRuntimeService`: connect to IBM Quantum runtime

IBM Quantum Backends

```
python
CopyEdit
api_token = os.getenv("IBM_QUANTUM_API_TOKEN")
```

Loads your IBM Quantum API token from environment variables.

```
python
CopyEdit
try:
    service = QiskitRuntimeService(channel="ibm_quantum", token=api_toke
n)
    print("🔹 Available IBM Quantum backends:")
    for backend in service.backends():
        print("-", backend.name)
```

Tries to connect to IBM Quantum Runtime and **lists all available backends** (quantum computers and simulators).

```
python
CopyEdit
except Exception as e:
    print(f"❌ Error loading IBM Quantum service: {e}")
    print("🔄 Switching to fallback resources...")
    print("🔗 IBM Quantum Login: https://cloud.ibm.com/quantum")
    print("📘 IBM Quantum Learning Portal: https://quantum-computing.ibm.
com/lab/docs/iql/")
    print("📚 Qiskit Textbook: https://qiskit.org/learn/")
    print("🧪 Quantum Lab: https://quantum-computing.ibm.com/")
```

If there's any error (wrong token, network issue), it:

- Prints the error.
- Shows helpful fallback links for IBM Quantum resources.

---

arXiv Query (Quantum papers)

```
python
CopyEdit
arxiv_base = os.getenv("ARXIV_API_BASE", "http://export.arxiv.org/api/quer
y")
search_query = "quantum+computing"
max_results = 3
```

Sets:

- The arXiv API base URL.
- The search term (`quantum computing`).
- How many papers to return (3).

```
python
CopyEdit
```

```
arxiv_url = f"{arxiv_base}?search_query=all:{search_query}&start=0&max_r
esults={max_results}"
```

Builds the **full API URL** for the query.

```
python
CopyEdit
try:
    response = requests.get(arxiv_url)
    root = ET.fromstring(response.content)

    print("\n 🔹 Recent arXiv results (quantum computing):")
    for entry in root.findall('{http://www.w3.org/2005/Atom}entry'):
        title = entry.find('{http://www.w3.org/2005/Atom}title').text.strip()
        link = entry.find('{http://www.w3.org/2005/Atom}id').text.strip()
        print(f"- {title}\n ↪ {link}")
```

Queries arXiv API, parses the XML, and prints each paper's:

- Title
- Link

```
python
CopyEdit
except Exception as e:
    print(f"❌ Error querying arXiv: {e}")
```

If any error occurs (e.g., network issue), it prints an error message.

Cell 10)Clone additional repos and setup project structure

!git clone https://github.com/i-am-bee/beeai-platform-agent-starter.git
!git clone
https://github.com/i-am-bee/beeai-framework-py-starter.git

Create project folder in Drive

!mkdir -p /content/drive/MyDrive/quantum_tutor_project

Move repos into it

!mv beeai-platform-agent-starter /content/drive/MyDrive/quantum_tutor_project/
!mv beeai-framework-py-starter /content/drive/MyDrive/quantum_tutor_project/

Cell 11: Install framework

%cd /content/drive/MyDrive/quantum_tutor_project/beeai-framework-py-starter
!pip install -e .

Explanation of Cell 11:

```
python
CopyEdit
%cd /content/drive/MyDrive/quantum_tutor_project/beeai-framework-py-starter
```

Changes your working directory to the cloned `beeai-framework-py-starter` folder inside your Google Drive project directory.

```
bash
CopyEdit
!pip install -e .
```

Installs the project in "editable" mode (`-e`):

- This means any changes you make to the code inside the folder will immediately reflect without reinstalling.
- Useful for development.

Cell 12) Multi-turn Quantum Tutor Pipeline

```
# Cell 12: Create Enhanced Multi-Turn QuantumTutorAgent with Conversation
code = """import re
import os
import json
import time
from datetime import datetime
from groq import Groq
from typing import List, Dict, Any

# --------- Base Agent Definition ---------
class Agent:
    def run(self, message: str, **kwargs):
        raise NotImplementedError

# --------- Conversation Memory Manager ---------
class ConversationMemory:
    def __init__(self, max_history: int = 10):
        self.history: List[Dict[str, Any]] = []
        self.max_history = max_history
        self.session_start = datetime.now()
        self.user_preferences = {}

    def add_interaction(self, user_message: str, bot_response: str, category: str
        interaction = {
            'timestamp': datetime.now().isoformat(),
            'user_message': user_message,
            'bot_response': bot_response,
            'category': category,
            'metadata': metadata or {}
        }

        self.history.append(interaction)

        # Keep only recent history
        if len(self.history) > self.max_history:
            self.history = self.history[-self.max_history:]
```

```
    def get_context_summary(self) → str:
        if not self.history:
            return "This is the start of our conversation."

        recent_topics = []
        for interaction in self.history[-3:]:  # Last 3 interactions
            category = interaction['category']
            snippet = interaction['user_message'][:50] + "..." if len(interaction['use
            recent_topics.append(f"({category}): {snippet}")

        return f"Recent conversation context: {'; '.join(recent_topics)}"

    def get_learning_progress(self) → Dict[str, int]:
        categories = {}
        for interaction in self.history:
            cat = interaction['category']
            categories[cat] = categories.get(cat, 0) + 1
        return categories

    def is_follow_up_question(self, current_message: str) → bool:
        follow_up_indicators = [
            'can you explain more', 'tell me more', 'what about', 'how about',
            'also', 'and', 'but', 'however', 'what if', 'why', 'how'
        ]
        return any(indicator in current_message.lower() for indicator in follow_up

# --------- Enhanced Multi-Turn QuantumTutorAgent ---------
class QuantumTutorAgent(Agent):
    def __init__(self, groq_client):
        self.groq_client = groq_client
        self.memory = ConversationMemory()
        self.session_stats = {
            'total_queries': 0,
            'session_start': datetime.now(),
            'favorite_topics': {}
        }
```

```
def format_response(self, response_text):
    # Add emojis to key quantum terms for fun and engagement
    response_text = re.sub(r'\\bquantum\\b', 'Quantum ', response_text, flags
    response_text = re.sub(r'\\bentanglement\\b', 'entanglement 🔗', respons
    response_text = re.sub(r'\\bsuperposition\\b', 'superposition ⚡', respons
    response_text = re.sub(r'\\bqubit\\b', 'qubit 🎯', response_text, flags=re.I
    response_text = re.sub(r'\\bcircuit\\b', 'circuit 🔌', response_text, flags=re
    return response_text

def classify_query(self, message: str) → str:
    lowered = message.lower()

    # Check for follow-up/continuation patterns
    if self.memory.is_follow_up_question(message):
        if self.memory.history:
            last_category = self.memory.history[-1]['category']
            return f"followup_{last_category}"

    # Regular classification
    if any(k in lowered for k in ['code', 'python', 'program', 'implementation', '
        return 'code'
    elif any(k in lowered for k in ['arxiv', 'paper', 'research', 'journal', 'citation'
        return 'research'
    elif any(k in lowered for k in ['difference', 'vs', 'compare', 'better']):
        return 'comparison'
    elif any(k in lowered for k in ['formula', 'derive', 'equation', 'proof', 'math']
        return 'math'
    elif any(k in lowered for k in ['application', 'real world', 'industry', 'use cas
        return 'application'
    elif any(k in lowered for k in ['history', 'who discovered', 'origin', 'timeline'
        return 'history'
    elif any(k in lowered for k in ['fun fact', 'joke', 'trivia', 'interesting']):
        return 'fun'
    elif any(k in lowered for k in ['mcq', 'quiz', 'questionnaire', 'test', 'practice
        return 'quiz'
    elif any(k in lowered for k in ['translate', 'in hindi', 'in tamil', 'meaning in']):
        return 'translation'
    elif any(k in lowered for k in ['help', 'what can you do', 'commands', 'featu
```

```
        return 'help'
    elif any(k in lowered for k in ['progress', 'summary', 'what have we covere
        return 'progress'
    else:
        return 'general'

def build_contextual_prompt(self, message: str, category: str) → str:
    context = self.memory.get_context_summary()
    learning_progress = self.memory.get_learning_progress()

    base_prompt = f'''You are QuantumTutor 🤖, a friendly and enthusiastic c

CONVERSATION CONTEXT: {context}

CURRENT QUERY: "{message}"
QUERY CATEGORY: {category}

LEARNING PROGRESS: {', '.join([f"{k}({v})" for k, v in learning_progress.items(

Instructions:
- Reference previous topics we discussed when relevant
- Build upon earlier explanations if this is a follow-up question
- Use simple language, analogies, and real-world examples
- Structure with: Hook → Key Points (•) → Encouraging Conclusion
- Keep responses engaging and conversational'''

    # Category-specific additions
    category_prompts = {
        'code': "\\n\\nInclude Python/Qiskit code snippets with explanations.",
        'research': "\\n\\nSuggest relevant arXiv papers and research direction
        'comparison': "\\n\\nProvide clear comparisons with pros/cons tables."
        'math': "\\n\\nInclude mathematical formulations when helpful.",
        'application': "\\n\\nEmphasize real-world applications and industry use
        'history': "\\n\\nAdd historical context and discovery timeline.",
        'fun': "\\n\\nInclude fun facts, analogies, or quantum jokes!",
        'quiz': "\\n\\nCreate 2-3 MCQs with detailed explanations.",
        'translation': "\\n\\nProvide explanations in multiple languages if reque
        'help': "\\n\\nList my capabilities and suggest interesting quantum topic
```

```
        'progress': "\\n\\nSummarize what we've covered and suggest next lea
    }

    # Handle follow-up questions
    if category.startswith('followup_'):
        base_prompt += "\\n\\nThis seems like a follow-up question. Build dire
        original_category = category.replace('followup_', '')
        if original_category in category_prompts:
            base_prompt += category_prompts[original_category]
    elif category in category_prompts:
        base_prompt += category_prompts[category]

    return base_prompt

def generate_session_summary(self) → str:
    progress = self.memory.get_learning_progress()
    total_interactions = len(self.memory.history)

    if total_interactions == 0:
        return "🌟 Welcome to your quantum learning journey!"

    summary = f'''
📊 **Session Summary**
• Total questions asked: {total_interactions}
• Topics explored: {', '.join(progress.keys())}
• Most discussed: {max(progress.keys(), key=progress.get) if progress else 'N
• Session duration: {datetime.now() - self.memory.session_start}
    '''
    return summary

def run(self, message: str, **kwargs):
    start_time = time.time()

    try:
        # Update session stats
        self.session_stats['total_queries'] += 1

        # Classify and build contextual prompt
```

```
        category = self.classify_query(message)
        prompt = self.build_contextual_prompt(message, category)

        # Handle special commands
        if category == 'help':
            response = self.get_help_response()
        elif category == 'progress':
            response = self.generate_session_summary()
        else:
            # Generate response using Groq
            chat_completion = self.groq_client.chat.completions.create(
                messages=[{"role": "user", "content": prompt}],
                model="llama3-8b-8192",
                temperature=0.7,
                max_tokens=2000,
            )
            response = chat_completion.choices[0].message.content

        # Format and store in memory
        formatted_response = self.format_response(response)

        # Add to conversation memory
        self.memory.add_interaction(
            user_message=message,
            bot_response=formatted_response,
            category=category,
            metadata={'response_time': time.time() - start_time}
        )

        return {
            'response': formatted_response,
            'category': category,
            'session_stats': self.session_stats.copy(),
            'conversation_length': len(self.memory.history),
            'response_time': time.time() - start_time
        }

    except Exception as e:
```

```
            error_response = f'🔧 Oops! I encountered an error: {str(e)}\\nLet\\'s tr

            self.memory.add_interaction(
                user_message=message,
                bot_response=error_response,
                category='error',
                metadata={'error': str(e)}
            )

            return {
                'response': error_response,
                'category': 'error',
                'session_stats': self.session_stats.copy(),
                'conversation_length': len(self.memory.history),
                'response_time': time.time() - start_time
            }

    def get_help_response(self) → str:
        return '''
🤖 **QuantumTutor Capabilities**

**What I can help you with:**
• 💻 **Code**: Python/Qiskit quantum programming
• 📚 **Research**: Latest papers and arXiv suggestions
• ⚖️ **Comparisons**: Classical vs Quantum concepts
• 🌍 **Applications**: Real-world quantum use cases
• 📅 **History**: Quantum computing timeline
• 🎯 **Quizzes**: Test your quantum knowledge
• 🌐 **Translation**: Concepts in multiple languages

**Try asking:**
- "Explain quantum entanglement"
- "Show me a simple Qiskit circuit"
- "What are the latest quantum research papers?"
- "Give me a quantum quiz"
- "What's my learning progress?"

Let's explore the quantum world together! 🚀
```

```
        '''

    def reset_conversation(self):
        '''Reset conversation memory - useful for starting fresh'''
        self.memory = ConversationMemory()
        self.session_stats = {
            'total_queries': 0,
            'session_start': datetime.now(),
            'favorite_topics': {}
        }
        return "🔄 Conversation reset! Ready for a fresh quantum learning sessio
"""

file_path = '/content/drive/MyDrive/quantum_tutor_project/beeai-platform-age
with open(file_path, 'w') as f:
    f.write(code)

print(f"✅ Enhanced Multi-Turn QuantumTutorAgent saved to: {file_path}")
```

Structure Overview

Let's look at the **main parts** in order:

`Agent` base class

```
python
CopyEdit
class Agent:
    def run(self, message: str, **kwargs):
        raise NotImplementedError
```

- Defines a generic **interface** for agents.
- Any child class must implement `run()`.

`ConversationMemory`

This manages **memory** of what the user and bot have said.

- Keeps a history of up to 10 interactions.
- Tracks session start.
- Stores user preferences.
- Provides summaries of recent context.
- Detects if the current message is a **follow-up question**.

Important Methods:

`add_interaction()`

Stores a single turn (user message + bot response).

`get_context_summary()`

Summarizes the last 3 topics so the bot can reference them.

`get_learning_progress()`

Counts how many queries per category.

`is_follow_up_question()`

Uses keywords to decide if the message is a follow-up.

`QuantumTutorAgent`

This is the **smart quantum tutor**.

Constructor

```
python
CopyEdit
def __init__(self, groq_client):
```

- Takes a `groq_client` (Groq is the LLM backend).
- Creates:
  - A `ConversationMemory` instance.
  - A `session_stats` dictionary.

Key Methods

`classify_query(message)`

Decides what kind of question this is:

- `code`: Python, Qiskit
- `research`: papers
- `comparison`: differences
- `math`: formulas
- `application`: real-world use
- `history`: timeline
- `fun`: trivia
- `quiz`: MCQs
- `translation`: different languages
- `help`: show capabilities
- `progress`: show learning progress
- `general`: fallback
- `followup_...`: if it's a continuation of a previous question

`build_contextual_prompt()`

Constructs a prompt string to send to Groq.

- Includes:
  - A summary of the last interactions.
  - The learning progress.
  - The current query and category.
  - Special instructions (e.g., add code snippets, create quizzes).

This helps the model generate a **custom response** tailored to the conversation.

`format_response()`

Adds fun formatting:

- Replaces words like “qubit” with “qubit 🎯” to make the output engaging.

`generate_session_summary()`

Builds a report of:

- Total questions
- Topics covered
- Most discussed topic
- Session duration

`get_help_response()`

Lists everything the bot can do.

`run(message)`

Main function called every time the user says something:

1. Increments query count.
2. Classifies the question.
3. Builds prompt.
4. Checks for special commands:
    - If `help`: returns help text.
    - If `progress`: returns progress summary.
    - Otherwise, calls Groq to get the response.
5. Formats the response.
6. Saves it in memory.
7. Returns a dictionary containing:
    - The response
    - The category
    - Session stats
    - Number of interactions
    - Response time

If there's an error, it logs and returns a friendly error message.

`reset_conversation()`

Clears the memory and resets stats.

What does the final part do?

```
python
CopyEdit
file_path = '/content/drive/MyDrive/.../__init__.py'
with open(file_path, 'w') as f:
    f.write(code)
```

- This **writes all this code to a file** in your Google Drive, so you can import it as a module (`quantum_tutor_agent`) in your project.

Quick Example

Suppose you ask:

> "Can you explain quantum entanglement?"

Flow:

1. `classify_query()` → `general` (or `math`)
2. `build_contextual_prompt()` → includes past conversation
3. Sends prompt to Groq
4. Gets response
5. `format_response()` adds emojis
6. `add_interaction()` saves the turn
7. Returns the response to you

```
# Enhanced Quantum Tutor Web Dashboard - Google Colab Compatible
import sys
import os
import time
import json
import re
from datetime import datetime
from flask import Flask, render_template_string, request, jsonify
import threading
from groq import Groq

# Install required packages
try:
    from google.colab import drive
    import subprocess
    subprocess.run(['pip', 'install', 'flask', 'pyngrok'], check=True)
    from pyngrok import ngrok
    IN_COLAB = True
except ImportError:
    IN_COLAB = False
    print("Not running in Google Colab")

class QuantumTutorWebApp:
    def __init__(self):
        self.app = Flask(__name__)
        self.groq_client = None
        self.agent = None
        self.chat_history = []
        self.session_stats = {
            'start_time': datetime.now(),
            'total_queries': 0,
            'topics_covered': set(),
            'avg_response_time': 0
        }

        # Modern Color Scheme
        self.colors = {
            'primary': '#2563eb',
```

```
        'secondary': '#7c3aed',
        'accent': '#06b6d4',
        'success': '#10b981',
        'warning': '#f59e0b',
        'error': '#ef4444',
        'bg_light': '#f8fafc',
        'bg_dark': '#1e293b',
        'text_primary': '#1f2937',
        'text_secondary': '#6b7280'
    }

    self.setup_routes()
    self.initialize_system()

def initialize_system(self):
    """Initialize the quantum tutor system"""
    try:
        if IN_COLAB:
            # Mount drive if needed
            if not os.path.exists('/content/drive'):
                drive.mount('/content/drive', force_remount=True)

            # Add project path
            sys.path.append('/content/drive/MyDrive/quantum_tutor_project/be

        # Initialize Groq client
        self.groq_client = Groq(api_key="gsk_b7nGx5Otfk36gpSLyXyoWGdyb3

        # Load quantum tutor agent
        from agents.quantum_tutor_agent import QuantumTutorAgent
        self.agent = QuantumTutorAgent(self.groq_client)

        print("✅ QuantumTutor initialized successfully!")
        return True

    except Exception as e:
        print(f"❌ Initialization Error: {str(e)}")
        return False
```

```
def setup_routes(self):
    """Setup Flask routes"""

    @self.app.route('/')
    def index():
        return render_template_string(self.get_html_template())

    @self.app.route('/chat', methods=['POST'])
    def chat():
        if not self.agent:
            return jsonify({
                'success': False,
                'error': 'System not initialized'
            })

        try:
            data = request.json
            user_query = data.get('message', '').strip()

            if not user_query:
                return jsonify({
                    'success': False,
                    'error': 'Empty message'
                })

            # Process with agent
            start_time = time.time()
            result = self.agent.run(user_query)
            response_time = time.time() - start_time

            # Update stats
            self.session_stats['total_queries'] += 1
            self.session_stats['topics_covered'].add(result.get('category', 'gener
            self.session_stats['avg_response_time'] = (
                (self.session_stats['avg_response_time'] * (self.session_stats['tota
                / self.session_stats['total_queries']
            )
```

```
        # Add to chat history
        timestamp = datetime.now().strftime("%H:%M:%S")
        self.chat_history.extend([
            {
                'sender': 'user',
                'message': user_query,
                'timestamp': timestamp
            },
            {
                'sender': 'bot',
                'message': result['response'],
                'timestamp': timestamp,
                'metadata': {
                    'response_time': response_time,
                    'category': result.get('category', 'general'),
                    'conversation_length': result.get('conversation_length', 0)
                }
            }
        ])

        return jsonify({
            'success': True,
            'response': result['response'],
            'metadata': {
                'response_time': response_time,
                'category': result.get('category', 'general'),
                'conversation_length': result.get('conversation_length', 0)
            },
            'stats': self.get_stats()
        })

    except Exception as e:
        return jsonify({
            'success': False,
            'error': str(e)
        })
```

```
        @self.app.route('/clear', methods=['POST'])
        def clear_chat():
            self.chat_history = []
            if self.agent:
                try:
                    self.agent.reset_conversation()
                except:
                    pass
            return jsonify({'success': True})

        @self.app.route('/progress', methods=['GET'])
        def get_progress():
            if self.agent:
                try:
                    summary = self.agent.generate_session_summary()
                    return jsonify({
                        'success': True,
                        'summary': summary
                    })
                except Exception as e:
                    return jsonify({
                        'success': False,
                        'error': str(e),
                        'stats': self.get_stats()
                    })
            return jsonify({
                'success': False,
                'error': 'Agent not initialized'
            })

        @self.app.route('/stats', methods=['GET'])
        def get_stats_endpoint():
            return jsonify(self.get_stats())

    def get_stats(self):
        """Get current session statistics"""
        duration = datetime.now() - self.session_stats['start_time']
        minutes = int(duration.total_seconds() // 60)
```

```
        seconds = int(duration.total_seconds() % 60)

        return {
            'total_queries': self.session_stats['total_queries'],
            'topics_covered': len(self.session_stats['topics_covered']),
            'avg_response_time': round(self.session_stats['avg_response_time'], 2)
            'duration': f"{minutes}m {seconds}s"
        }



def run_server():
    """Run the Flask server with ngrok tunnel"""
    app_instance = QuantumTutorWebApp()

    if IN_COLAB:
        # Set up ngrok tunnel
        public_url = ngrok.connect(5000)
        print(f"🌐 Public URL: {public_url}")
        print(f"🚀 Quantum Tutor Web Dashboard is now running!")
        print(f"📱 Access your dashboard at: {public_url}")
        print(f"🔄 The dashboard will remain active as long as this cell is running
```


```
def run_server():
    """Run the Flask server with ngrok tunnel"""
    app_instance = QuantumTutorWebApp()

    if IN_COLAB:
        # Set up ngrok tunnel
        public_url = ngrok.connect(5000)
        print(f"🌐 Public URL: {public_url}")
        print(f"🚀 Quantum Tutor Web Dashboard is now running!")
        print(f"📱 Access your dashboard at: {public_url}")
        print(f"🔄 The dashboard will remain active as long as this cell is running
```

```
        # Display clickable link in Colab output
        from IPython.display import display, HTML
        display(HTML(f"""
        <div style="background: linear-gradient(135deg, #2563eb 0%, #7c3aed
                    padding: 25px; border-radius: 16px; color: white; text-align: center
                    margin: 25px 0; box-shadow: 0 8px 32px rgba(37, 99, 235, 0.3);">
            <h2 style="margin: 0 0 10px 0;">🌐 Dashboard Now Available Online<
            <p style="margin: 0 0 15px 0; font-size: 1.2em;">
                <a href="{public_url}" target="_blank" style="color: white; text-deco
                    Click to Open Dashboard
                </a>
            </p>
            <div style="background: rgba(255,255,255,0.2); padding: 10px; border
                <p style="margin: 0; font-family: monospace;">Keep this Colab runn
            </div>
        </div>
        """))
    else:
        print("Running in local mode (not Colab)")

    # Run Flask app
    app_instance.app.run(host='0.0.0.0', port=5000)

# Run the server
if __name__ == '__main__':
    if IN_COLAB:
        # In Colab, we need to run in a thread
        import threading
        flask_thread = threading.Thread(target=run_server)
        flask_thread.daemon = True
        flask_thread.start()

        # Keep the cell running
        try:
            while True:
                time.sleep(1)
        except KeyboardInterrupt:
            print("\nShutting down server...")
```

```
  else:
    # In local mode, just run normally
    run_server()
```

Package Install

```
python
CopyEdit
!pip install flask pyngrok groq
```

- Installs dependencies:
  - `flask`: lightweight web server.
  - `pyngrok`: for creating a public tunnel (especially in Colab).
  - `groq`: your LLM API client.

```
python
CopyEdit
!ngrok authtoken ...
```

- Sets up your **ngrok authentication token** (so ngrok can give you a stable URL).

Environment Detection

```
python
CopyEdit
try:
  from google.colab import drive
  ...
  IN_COLAB = True
except ImportError:
```

```
IN_COLAB = False
```

- Checks if running inside **Google Colab**.
- Sets `IN_COLAB = True` if so.

Class: `QuantumTutorWebApp`

This is your **main Flask application**.

Constructor `__init__()`

- Initializes:
  - Flask app.
  - Groq client placeholder.
  - Agent placeholder.
  - Chat history list (holds all messages).
  - Session stats dict (total queries, avg response time, etc.).
  - Color palette for styling.
- Calls:
  - `setup_routes()` to define web endpoints.
  - `initialize_system()` to load Groq + your agent.

Method: `initialize_system()`

This sets up the **Groq client and the Quantum Tutor Agent**:

- If running in Colab:
  - Mounts Google Drive if needed.
  - Adds your project path to `sys.path`.
- Initializes:
  - `Groq` client (`Groq(api_key=...)`).
  - `QuantumTutorAgent` instance.

- Prints success or error.

Method: `setup_routes()`

Defines all the Flask **HTTP endpoints**.

Here's each endpoint in plain English:

`/` — Home Page

Returns **HTML template** to show the UI.

> You see render_template_string(self.get_html_template()).
>
> (Note: In your snippet, you didn't paste `get_html_template()`, but presumably it returns the web UI.)

`/chat` — Chat Endpoint

POST endpoint you call when you submit a message.

1. Reads JSON body:

    ```
    python
    CopyEdit
    data = request.json
    user_query = data.get('message', '').strip()
    ```

2. Validates input.
3. Sends the query to your `QuantumTutorAgent.run()`.
4. Measures response time.
5. Updates:
    - Total queries.
    - Topics covered.
    - Average response time.
6. Appends the user & bot messages to `self.chat_history`.

7. Returns JSON with:
    - The bot response.
    - Metadata (category, response time, etc.).
    - Updated session stats.

`/clear` — Clear Chat

POST endpoint to:

- Clear the chat history.
- Reset the agent's memory.

> Useful if you want a fresh session.

`/progress` — Get Learning Progress

GET endpoint:

- Calls `agent.generate_session_summary()` to summarize what you've learned so far.
- Returns that summary.

`/stats` — Session Stats

GET endpoint:

- Returns `total_queries`, `topics_covered`, `avg_response_time`, and session duration.

Method: `get_stats()`

Helper to produce a nice stats dictionary:

- Calculates duration.
- Returns counts and averages.

Method: `run_server()`

Responsible for **launching the server**:

- Creates an `QuantumTutorWebApp` instance.

- If in Colab:
  - Starts ngrok tunnel on port `5000`.
  - Prints public URL to access your dashboard.
  - Shows a nice HTML card with a clickable link.
- If local:
  - Just runs Flask directly.

Entrypoint `__main__`

```
python
CopyEdit
if __name__ == '__main__':
```

What happens when you run this script?

- If Colab:
  - Launches the server in a separate thread (`flask_thread`).
  - Keeps the cell running forever with `while True: time.sleep(1)`.
- If not Colab:
  - Runs server normally.

Example Workflow

1. Start in **Colab**.
2. This script:
   - Mounts Drive.
   - Loads your `QuantumTutorAgent`.
   - Starts Flask.
   - Creates an ngrok tunnel.
3. You get a **public URL** (e.g., `https://...ngrok.io`).

4. Open it in your browser.
5. You can:
    - Send messages (`/chat`).
    - Clear chat (`/clear`).
    - See progress (`/progress`).
    - Get stats (`/stats`).

B)RAG Agent

```
# Install required packages
!pip install langchain langchain-community chromadb transformers torch sent
!pip install huggingface_hub

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

1. Install required packages

Your first block installs all dependencies needed for building retrieval-augmented generation (RAG) pipelines, embedding models, and working with PDFs:

```
python
CopyEdit
!pip install langchain langchain-community chromadb transformers torch s
entence-transformers pypdf
!pip install huggingface_hub
```

Here's a quick explanation of each package:

- **langchain** and **langchain-community** – Framework for building LLM-powered apps and retrieval pipelines.
- **chromadb** – Vector store backend (to persist and search embeddings).
- **transformers** – Models from Hugging Face (e.g., BERT, GPT-2).
- **torch** – PyTorch backend, required by many transformer models.
- **sentence-transformers** – For generating dense embeddings from text (e.g., with `all-MiniLM`).
- **pypdf** – For parsing PDFs.
- **huggingface_hub** – To download models from Hugging Face easily.

```
# Import necessary libraries
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline, Au
import torch
from langchain.embeddings import HuggingFaceEmbeddings
import os
```

```
python
CopyEdit
# Import necessary libraries
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline,
AutoModel
import torch
from langchain.embeddings import HuggingFaceEmbeddings
```

```
import os
```

Here's a **quick explanation of what each import does**:

LangChain-related imports

- `PyPDFLoader`:
    - From `langchain_community`, used to load PDF documents for processing.
- `RecursiveCharacterTextSplitter`:
    - Splits text into chunks recursively (often used to create manageable embeddings).
- `Chroma`:
    - A vector store implementation that allows you to store and query embeddings (for retrieval-augmented generation).
- `HuggingFaceEmbeddings`:
    - Wrapper to generate embeddings using any Hugging Face model.

Transformers / Hugging Face

- `AutoTokenizer`:
    - Automatically loads the tokenizer for a given model.
- `AutoModelForCausalLM`:
    - Loads a language model suitable for text generation (e.g., GPT, Falcon, etc.).
- `AutoModel`:
    - Loads a generic model (e.g., for embeddings).
- `pipeline`:
    - High-level API for common NLP tasks.

Torch

- `torch`:
    - PyTorch library, typically used for tensor operations and GPU acceleration.

OS

- `os`:
    - Standard Python library to handle environment variables, paths, etc.

```
# Data Ingestion from Google Drive
def load_pdfs_from_drive(directory_path):
    """Load and split PDF documents from a specified Google Drive directory."'
    documents = []
    for filename in os.listdir(directory_path):
        if filename.endswith('.pdf'):
            file_path = os.path.join(directory_path, filename)
            loader = PyPDFLoader(file_path)
            docs = loader.load()
            documents.extend(docs)
    return documents

# Specify the Google Drive directory containing PDFs
drive_pdf_directory = '/content/drive/MyDrive/pdfs'
documents = load_pdfs_from_drive(drive_pdf_directory)

# Split documents into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overla
split_documents = text_splitter.split_documents(documents)

print(f"Loaded and split {len(split_documents)} document chunks.")
```

Define a function to load PDFs from a directory

```
python
CopyEdit
def load_pdfs_from_drive(directory_path):
    """Load and split PDF documents from a specified Google Drive director
y."""
    documents = []
    for filename in os.listdir(directory_path):
        if filename.endswith('.pdf'):
```

```
        file_path = os.path.join(directory_path, filename)
        loader = PyPDFLoader(file_path)
        docs = loader.load()
        documents.extend(docs)
    return documents
```

- **Loops through all files** in `directory_path`.
- If a file ends with `.pdf`, it:
  - Builds the full path.
  - Creates a `PyPDFLoader` instance.
  - Loads the document (`loader.load()` returns a list of `Document` objects).
  - Adds all `docs` to the `documents` list.
- Returns a **list of all loaded Document objects.**

Specify the Google Drive directory

```
python
CopyEdit
drive_pdf_directory = '/content/drive/MyDrive/pdfs'
```

This is the path in your Google Drive (assuming you mounted it with something like:

```
python
CopyEdit
from google.colab import drive
drive.mount('/content/drive')
```

).

Load documents

```python
python
CopyEdit
documents = load_pdfs_from_drive(drive_pdf_directory)

 This now contains all the loaded PDFs as Document objects.
```

Split documents into chunks

```python
python
CopyEdit
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_ove
rlap=200)
split_documents = text_splitter.split_documents(documents)
```

Why do this?

- **Chunking** helps you:
  - Fit text within embedding model limits.
  - Preserve context across overlaps.
- **Settings:**
  - Each chunk: up to **1000 characters**.
  - Overlap between chunks: **200 characters** (so the chunks share context).

Print confirmation

```python
python
CopyEdit
print(f"Loaded and split {len(split_documents)} document chunks.")
```

This tells you how many total chunks you now have ready for:

- Embedding,

- Storing in a vector DB (e.g., Chroma),
- Retrieval for question-answering.

```
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
import torch

# Model ID
model_id = "deepseek-ai/deepseek-llm-7b-chat"

# Initialize tokenizer and model
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    device_map="cuda",  # Explicitly use T4 GPU
    offload_folder="/content/drive/MyDrive/quantum_tutor_offload",
    torch_dtype=torch.float16,  # Half-precision for T4 GPU
    low_cpu_mem_usage=True  # Minimize CPU memory usage
)

# Create generation pipeline
deepseek_llm = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    max_new_tokens=500,
    temperature=0.7,
    top_p=0.9
)
```

Code Walkthrough

Imports

```
python
CopyEdit
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
import torch
```

- Loads all the necessary Hugging Face and PyTorch components.

Model ID

```
python
CopyEdit
model_id = "deepseek-ai/deepseek-llm-7b-chat"
```

- You're loading **DeepSeek LLM 7B**, a popular 7-billion parameter chat model.
- Make sure you have the right permissions—some models require authentication.

Initialize Tokenizer and Model

```
python
CopyEdit
tokenizer = AutoTokenizer.from_pretrained(model_id)
```

- Downloads and loads the tokenizer.

```
python
CopyEdit
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    device_map="cuda",  # Puts model on GPU
    offload_folder="/content/drive/MyDrive/quantum_tutor_offload",
    torch_dtype=torch.float16,  # Use half-precision (faster and saves memo
ry)
```

```
    low_cpu_mem_usage=True
)
```

Notes:

- `device_map="cuda"`: loads the model directly to your GPU (e.g., T4 in Colab).
- `offload_folder`: where layers get offloaded if GPU runs out of VRAM (smart if you only have ~16GB VRAM).
- `torch_dtype=torch.float16`: uses half precision for:
  - Faster inference.
  - Lower VRAM usage.
- `low_cpu_mem_usage=True`: minimizes RAM footprint while loading.

Create a text generation pipeline

```
python
CopyEdit
deepseek_llm = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    max_new_tokens=500,
    temperature=0.7,
    top_p=0.9
)
```

Explanation of parameters:

- **Task:** `"text-generation"`
- **max_new_tokens=500:** Generates up to 500 tokens.
- **temperature=0.7:** Controls randomness (0.0 = deterministic, >1.0 = more creative).
- **top_p=0.9:** Top-p nucleus sampling (keeps the most probable tokens whose cumulative probability adds up to 0.9).

```
from langchain_community.vectorstores import Chroma
from langchain.embeddings import HuggingFaceEmbeddings
import os

# IBM Granite Embedding
granite_embedding = HuggingFaceEmbeddings(
    model_name="ibm-granite/granite-embedding-125m-english",
    model_kwargs={"device": "cuda"},
    encode_kwargs={"batch_size": 16}
)

# persist ChromaDB vector store
db = Chroma.from_documents(
    documents=split_documents,
    embedding=granite_embedding,
    persist_directory="/content/drive/MyDrive/chroma_db"
)
db.persist()
print("Vector store created and persisted with Granite Embedding.")
```

Code Walkthrough

Imports

```
python
CopyEdit
from langchain_community.vectorstores import Chroma
from langchain.embeddings import HuggingFaceEmbeddings
import os
```

- `Chroma`: vector DB for storing/retrieving embeddings.
- `HuggingFaceEmbeddings`: wrapper for Hugging Face embedding models.

Initialize IBM Granite Embedding

```
python
CopyEdit
granite_embedding = HuggingFaceEmbeddings(
    model_name="ibm-granite/granite-embedding-125m-english",
    model_kwargs={"device": "cuda"},
    encode_kwargs={"batch_size": 16}
)
```

Explanation:

- **Model:** `ibm-granite/granite-embedding-125m-english`
  - Lightweight embedding model by IBM.
  - Good tradeoff between speed and semantic quality.
- **model_kwargs={"device": "cuda"}**
  - Runs on GPU.
- **encode_kwargs={"batch_size": 16}**
  - Controls how many chunks are encoded at once (reduce if you get OOM errors).

Create Chroma Vector Store

```
python
CopyEdit
db = Chroma.from_documents(
    documents=split_documents,
    embedding=granite_embedding,
    persist_directory="/content/drive/MyDrive/chroma_db"
)
```

What this does:

- For each chunk in `split_documents`:

- Encodes the chunk into a vector embedding.
- Stores it inside the Chroma DB.

- `persist_directory` is where your database files are saved (so you can reload later).

Persist to disk

```
python
CopyEdit
db.persist()
```

Saves all data to:

```
bash
CopyEdit
/content/drive/MyDrive/chroma_db
```

so you can **reload it later without recomputing embeddings**.

Confirmation

```
python
CopyEdit
print("Vector store created and persisted with Granite Embedding.")
```

Confirms everything worked.

```
import time

def rag_pipeline(query, db, llm, top_k=3, max_tokens=1000):
    import torch, gc

    # Start total timing
```

```
    total_start = time.time()

    # Clear GPU memory before generation
    gc.collect()
    torch.cuda.empty_cache()

    # Retrieve relevant documents - with timing
    retrieval_start = time.time()
    docs = db.similarity_search(query, k=top_k)
    context = "\n".join([doc.page_content[:500] for doc in docs])  # Truncate ea
    retrieval_time = time.time() - retrieval_start

    # Friendly, educational prompt
    prompt = f"""You are an expert quantum computing tutor with a friendly anc

Context: {context}

Query: {query}

Answer:"""

    # Generate model output - with timing
    generation_start = time.time()
    response = llm(
        prompt,
        return_full_text=False,
        max_new_tokens=max_tokens,
        temperature=0.8,
        top_p=0.8
    )[0]['generated_text']
    generation_time = time.time() - generation_start

    # Calculate total time
    total_time = time.time() - total_start

    # Clear GPU memory after generation
    gc.collect()
    torch.cuda.empty_cache()
```

```
# Return response and timing info
return response, {
    'retrieval_time': retrieval_time,
    'generation_time': generation_time,
    'total_time': total_time
}
```

What This Function Does

Purpose

Designed a pipeline that:

1. **Retrieves top-k relevant chunks** from your Chroma vector store.
2. **Builds a friendly prompt** including the retrieved context.
3. **Generates a detailed answer** using your LLM (e.g., DeepSeek).
4. **Measures performance timing** for retrieval and generation.
5. **Manages GPU memory** before and after generation.

Start timing

```
python
CopyEdit
total_start = time.time()
```

You begin recording total runtime.

Clean up GPU memory

```
python
CopyEdit
gc.collect()
```

```
torch.cuda.empty_cache()
```

Why do this?

- Frees up VRAM and RAM to avoid OOM errors when generating.

Retrieve documents (timed)

```
python
CopyEdit
retrieval_start = time.time()
docs = db.similarity_search(query, k=top_k)
context = "\n".join([doc.page_content[:500] for doc in docs])  # Truncate e
ach doc
retrieval_time = time.time() - retrieval_start
```

- Retrieves `top_k` chunks.
- Truncates each chunk to **500 characters** for a concise context window.
- Records retrieval time.

Build prompt

```
python
CopyEdit
prompt = f"""You are an expert quantum computing tutor ...
Context: {context}

Query: {query}

Answer:"""
```

Features:

- Friendly, educational style.
- Encourages analogies, examples, clear structure.

- If context is insufficient, the LLM is instructed to explain generally.

Generate answer (timed)

```
python
CopyEdit
generation_start = time.time()
response = llm(
    prompt,
    return_full_text=False,
    max_new_tokens=max_tokens,
    temperature=0.8,
    top_p=0.8
)[0]['generated_text']
generation_time = time.time() - generation_start
```

Parameters:

- `max_new_tokens=1000` : up to 1000 tokens in the output.
- `temperature=0.8` : some creativity.
- `top_p=0.8` : nucleus sampling for balanced diversity.

Stop timing

```
python
CopyEdit
total_time = time.time() - total_start
```

Computes total time including retrieval + generation.

Clean up GPU memory again

```
python
CopyEdit
gc.collect()
```

```
torch.cuda.empty_cache()
```

Avoids memory leaks or accumulation between calls.

Return answer + timing info

```
python
CopyEdit
return response, {
    'retrieval_time': retrieval_time,
    'generation_time': generation_time,
    'total_time': total_time
}
```

Very useful for monitoring performance.

C) Quality Test

```
!pip install -q nltk scikit-learn python-Levenshtein PyMuPDF
import nltk
nltk.download('punkt')
```

Command Explanation

```
bash
CopyEdit
!pip install -q nltk scikit-learn python-Levenshtein PyMuPDF
```

Packages installed:

1. **nltk**
    - Natural Language Toolkit.
    - Tokenization, stopwords, stemming, etc.
2. **scikit-learn**
    - Machine learning tools (e.g., clustering, vectorizers, cosine similarity).
3. **python-Levenshtein**
    - Super-fast edit distance (Levenshtein distance) computations.
    - Useful for fuzzy matching text.
4. **PyMuPDF**
    - PDF parsing library (alternative to PyPDF2 or pdfminer).
    - Can extract text with layout preservation.

NLTK Download

```
python
CopyEdit
import nltk
nltk.download('punkt')
```

Downloads the **Punkt tokenizer models**:

- Required for:
    - `nltk.sent_tokenize()`
    - `nltk.word_tokenize()`
- Without this step, tokenization will fail.

```
import fitz

def read_pdf_text(pdf_path):
    doc = fitz.open(pdf_path)
    text = ""
```

```
    for page in doc:
        text += page.get_text()
    return text
```

Code Walkthrough

```
python
CopyEdit
import fitz

def read_pdf_text(pdf_path):
    doc = fitz.open(pdf_path)
    text = ""
    for page in doc:
        text += page.get_text()
    return text
```

How It Works

`fitz.open(pdf_path)`

- Opens the PDF document at the given path.
- `doc` is a `fitz.Document` object.

Iterate over all pages

```
python
CopyEdit
for page in doc:
```

- Each `page` is a `fitz.Page` object.

Extract text from each page

```
python
CopyEdit
```

```
page.get_text()
```

- Extracts text from the page (including layout).
- **Alternative modes**:
  - `get_text("text")` (default): plain text.
  - `get_text("blocks")`: list of text blocks.
  - `get_text("html")`: HTML representation.
  - `get_text("dict")`: detailed dictionary structure.
  - `get_text("json")`: JSON.

Accumulate all text

- Concatenates text from each page into one big string.

Return the complete text

- You get the full document text as a single string.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100
)

chatbot_text = read_pdf_text("/content/drive/MyDrive/Quantum_Tutor_250.pd
gemini_text = read_pdf_text("/content/drive/MyDrive/Gold_Standard_Gemini.p

chatbot_chunks = splitter.split_text(chatbot_text)
gemini_chunks = splitter.split_text(gemini_text)

print(f"Chatbot Chunks: {len(chatbot_chunks)}")
print(f"Gemini Chunks: {len(gemini_chunks)}")
```

Code Walkthrough

Initialize the Text Splitter

```
python
CopyEdit
from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100
)
```

What this does:

- **chunk_size=1000**:

  Each chunk will be **up to 1,000 characters long**.

- **chunk_overlap=100**:

  Each chunk will overlap the previous by **100 characters** to maintain context across chunk boundaries.

- `RecursiveCharacterTextSplitter`:
  - Splits text hierarchically (tries splitting on paragraphs, then sentences, then characters).

Load the text from your PDFs

```
python
CopyEdit
chatbot_text = read_pdf_text("/content/drive/MyDrive/Quantum_Tutor_250.
pdf")
gemini_text = read_pdf_text("/content/drive/MyDrive/Gold_Standard_Gemin
i.pdf")
```

This uses your `read_pdf_text()` function to get raw text strings from both PDFs.

Split text into chunks

```
python
CopyEdit
chatbot_chunks = splitter.split_text(chatbot_text)
gemini_chunks = splitter.split_text(gemini_text)
```

Result:

- `chatbot_chunks`: list of ~1,000-character chunks from the first document.
- `gemini_chunks`: list of ~1,000-character chunks from the second document.

This is perfect for:

- Embedding (e.g., `HuggingFaceEmbeddings`)
- Similarity search
- Cross-document comparison
- Feeding into LLMs with context window limits

```
python
CopyEdit
print(f"Chatbot Chunks: {len(chatbot_chunks)}")
print(f"Gemini Chunks: {len(gemini_chunks)}")
```

- How many total chunks were created.
- Whether your documents were split properly.

```
# Recompute embeddings if you didn't already keep them
chatbot_embeddings = model.encode(chatbot_chunks, normalize_embedding
gemini_embeddings = model.encode(gemini_chunks, normalize_embeddings=
```

Helps us switch from Euclidean (L2) distance to **cosine similarity**, or just ensure consistent scaling across embeddings.

```
python
CopyEdit
normalize_embeddings=True
```

we are using **L2-normalizing** each embedding vector:

you are **L2-normalizing** each embedding vector:

$$\text{normalized\_vector} = \frac{\text{vector}}{||\text{vector}||}$$

✅ **What this does:**

- Makes every vector have unit length.
- **Cosine similarity** and **inner product** become equivalent (since cosine similarity is just the dot product of normalized vectors).
- Ensures distances are consistent across queries.

Cosine similarity from normalized vectors

Recall:

$$\cos(\theta) = \frac{A \cdot B}{||A|| \cdot ||B||}$$

If you **normalize** `A` and `B`, then:

$$\cos(\theta) = A \cdot B$$

```
semantic_cosines_cb = []

for i in range(len(chatbot_embeddings)):
```

```
    v_chat = chatbot_embeddings[i]
    similarities = np.dot(gemini_embeddings, v_chat)
    best_sim = np.max(similarities)
    semantic_cosines_cb.append(best_sim)

print(f"Chatbot → Gemini Coverage: {np.mean(semantic_cosines_cb):.4f}")
```

For **every Chatbot chunk**, we:

1. Compute its cosine similarity with **all Gemini chunks**.
2. Keep **only the best (highest) similarity score**.
3. Store that score.
4. At the end, you take the **mean of these maximum similarities**.

Recall:

- Your embeddings are **normalized**, so:cosine similarity=vchat·vgemini

  cosine similarity=

$$\text{cosine similarity} = \mathbf{v}_{\text{chat}} \cdot \mathbf{v}_{\text{gemini}}$$

- Each similarity score is in the range [−1,+1]

This approach answers:

> For each Chatbot chunk, how similar is it to the most similar Gemini chunk?

This gives you a **coverage score**:

- 1.0 = perfect semantic match
- ~0 = unrelated
- Negative = opposed meaning

Results

```yaml
yaml
CopyEdit
Chatbot → Gemini Coverage: 0.78
```

This means:

> On average, every Chatbot chunk is ~78% semantically similar to at least one Gemini chunk.

```
thresholds = [0.3, 0.5, 0.7, 0.8]  # you can adjust these

max_similarities = []

for i in range(len(gemini_embeddings)):
    v_gemini = gemini_embeddings[i]
    similarities = np.dot(chatbot_embeddings, v_gemini)
    best_sim = np.max(similarities)
    max_similarities.append(best_sim)

embedding_recall = {}

for t in thresholds:
    recall = sum(s >= t for s in max_similarities) / len(max_similarities)
    embedding_recall[t] = recall

print("\n✅ Embedding Recall at Different Thresholds:\n")
for t, r in embedding_recall.items():
    print(f"Threshold {t:.2f}: Recall {r*100:.2f}%")
```

> For each Gemini chunk, does it have any Chatbot chunk with cosine similarity ≥ threshold?

And then you compute the % of Gemini chunks that meet that bar.

This is very similar to an **embedding recall curve**, which is extremely useful in retrieval evaluation.

Compute Maximum Similarity per Gemini Chunk

```
python
CopyEdit
max_similarities = []

for i in range(len(gemini_embeddings)):
    v_gemini = gemini_embeddings[i]
    similarities = np.dot(chatbot_embeddings, v_gemini)
    best_sim = np.max(similarities)
    max_similarities.append(best_sim)
```

- For each Gemini chunk embedding:
    - Computes cosine similarity to every Chatbot embedding.
    - Records the **highest similarity score**.

Evaluate Recall at Thresholds

```
python
CopyEdit
embedding_recall = {}

for t in thresholds:
    recall = sum(s >= t for s in max_similarities) / len(max_similarities)
    embedding_recall[t] = recall
```

- For each threshold `t` (e.g., 0.7), you check:
    - What fraction of Gemini chunks have at least one Chatbot chunk above this similarity.

- This shows how strictly aligned your content is.

```
python
CopyEdit
print("\n✅ Embedding Recall at Different Thresholds:\n")
for t, r in embedding_recall.items():
    print(f"Threshold {t:.2f}: Recall {r*100:.2f}%")
```

```
yaml
CopyEdit
✅ Embedding Recall at Different Thresholds:

Threshold 0.30: Recall 100.00%
Threshold 0.50: Recall 100.00%
Threshold 0.70: Recall 79.62%
Threshold 0.80: Recall 13.99%
```

Inference

1) **Threshold 0.30: Recall 100%**

- Every Gemini chunk had **at least one** Chatbot chunk with *some* loose semantic similarity (>0.3).
- Means your corpus is **topically related overall**.

2) **Threshold 0.50: Recall 100%**

- Even at a moderate similarity threshold (>0.5), every Gemini chunk found a reasonably related Chatbot chunk.
- This suggests **broad coverage**—your Chatbot content and Gemini content discuss similar topics at least in general terms.

3) **Threshold 0.70: Recall ~79.6%**

- About 80% of Gemini chunks have **strong alignment** (cosine similarity >0.7) to some Chatbot chunk.

- This is quite high—indicating many passages are *very similar in meaning*.

4) **Threshold 0.80: Recall ~14%**

- Only ~14% of Gemini chunks have **near-paraphrase or high similarity** (cosine >0.8).
- This means that while the content is broadly covering the same topics, **exact phrasing and expression differ significantly**.
- From an originality perspective, this is actually good—it shows you have **overlap in concepts but mostly unique language**.
