AI Chatbot for School Queries

Complete Learning Guide & Tutorial

A Comprehensive Educational Resource

Course Information:

- **Duration:** 20-25 hours of study
- **Level:** Beginner to Intermediate
- **Prerequisites:** Basic computer skills, interest in programming
- **Goal:** Master AI chatbot development from scratch

📚 Table of Contents

PART 1: FOUNDATIONS

Chapter 1: Introduction to AI Chatbots

1.1 What is a Chatbot?

A **chatbot** is a computer program designed to simulate human conversation. Think of it as a virtual assistant that can understand questions and provide helpful answers.

Real-World Example: When you ask Siri "What's the weather today?" or ask Alexa to "Play some music," you're interacting with a chatbot. Our school chatbot works the same way, but it answers questions about school schedules, homework, and events.

1.2 How Do Chatbots Work?

Let's break down the process step by step:

Step 1: User Input

- A student types: "What's my math homework?"
- The text is captured by the application

Step 2: Processing

- The application sends the question to an AI (Artificial Intelligence) service
- It also includes context about your school (schedules, assignments, etc.)

Step 3: AI Understanding

- The AI reads the question
- It searches through the school information
- It formulates a helpful answer

Step 4: Response

- The AI sends back an answer like: "Your math homework is to complete exercises 1-20 on page 45"
- The application displays this to the student

Visual Flow:

```
Student → Types Question → App Receives → Adds School Info →
Sends to AI → AI Thinks → AI Responds → App Shows Answer → Student Reads
```

1.3 Why Build a School Chatbot?

Problem it Solves:

- Students forget homework assignments
- Parents need quick access to school schedules
- Staff spend time answering repetitive questions
- Information is scattered across different documents

Our Solution:

- 24/7 instant answers
- All school information in one place
- No waiting for office hours
- Consistent, accurate responses

1.4 What Makes Our Chatbot Special?

1. Intelligent Responses

- Uses OpenAI's GPT (Generative Pre-trained Transformer)
- Understands natural language (you can ask questions normally)
- Provides context-aware answers

2. User-Friendly Interface

- Clean, modern design
- Easy to use on any device
- Chat-style interface everyone understands

3. Comprehensive Features

- Search through conversation history
- Download chat transcripts
- Multi-language support
- Quick action buttons

Chapter 2: Understanding the Technology

2.1 The Technology Stack

A "technology stack" is like the ingredients in a recipe. Here are ours:

2.1.1 Python (The Programming Language)

What is Python? Python is a programming language - a way to write instructions for computers. It's popular because it reads almost like English.

Example:

```
# This is Python code
name = "Alice"
```

```
print("Hello, " + name)
# Output: Hello, Alice
```

Why Python?

- **Easy to Learn:** Syntax is simple and readable
- **Powerful:** Can build complex applications
- **Popular:** Used by Google, Netflix, NASA
- **Great Libraries:** Pre-built tools for AI, web, data

Real-World Analogy: If building an app is like building a house:

- Python is the language you use to give instructions to workers
- It's easier than other languages (like trying to speak clearly vs. using complicated technical jargon)

2.1.2 Streamlit (The Web Framework)

What is Streamlit? Streamlit is a Python library that turns your Python scripts into web applications. It's like magic - you write Python, and it creates a website automatically!

Without Streamlit:

```
<!-- You'd need to write HTML, CSS, JavaScript -->
<html>
  <head><title>My App</title></head>
  <body>
    <input type="text" id="name">
    <button onclick="sayHello()">Click</button>
  </body>
  <script>
    function sayHello() { /* complex JavaScript */ }
  </script>
</html>
```

With Streamlit:

```
import streamlit as st
name = st.text_input("Enter your name:")
if st.button("Click"):
    st.write(f"Hello, {name}!")
```

See the difference? Much simpler!

Key Concepts:

Widgets: Interactive elements users can click/type in

```
st.text_input()  # Creates a text box
st.button()      # Creates a button
st.selectbox()   # Creates a dropdown menu
```

Layout: How things are arranged on the page

```
st.sidebar       # Puts content in a sidebar
st.columns()     # Divides page into columns
```

Display: Showing information

```
st.write()       # Shows text
st.markdown()    # Shows formatted text
st.image()       # Shows images
```

2.1.3 OpenAI API (The AI Brain)

What is an API? API stands for "Application Programming Interface." Think of it as a menu at a restaurant:

- You (your app) look at the menu (API documentation)
- You order food (make a request)
- The kitchen (OpenAI's servers) prepares it
- The waiter (API) brings it to you (sends response)

What is OpenAI? OpenAI is a company that created powerful AI models like ChatGPT. Their API lets us use this AI in our own applications.

How It Works:

1. You send a request:

```
"Hey AI, answer this question about school schedules:
What time is lunch?"
```

2. AI processes it:

- Reads your question
- Looks at the school data you provided
- Formulates a natural response

3. You get a response:

```
"Lunch time is from 12:30 PM to 1:15 PM every day."
```

Key Terms:

Model: The AI brain (we use "gpt-4o-mini")

- Like choosing which chef to cook your meal
- Different models have different capabilities

Prompt: The instructions you give the AI

```
"You are a helpful school assistant.
Answer questions about homework and schedules."
```

Tokens: Units of text the AI processes

- Roughly 1 token = 4 characters
- "Hello World" ≈ 3 tokens
- APIs charge per token used

Temperature: How creative the AI should be

- 0.0 = Very focused, factual
- 1.0 = Creative, varied
- We use 0.7 (balanced)

2.2 How These Technologies Work Together

The Complete Flow:

```
USER INTERACTION LAYER (What user sees)
↓
Browser displays Streamlit web page
User types: "What's my homework?"
↓
STREAMLIT LAYER (Python code)
Captures user input
Formats the question
↓
APPLICATION LOGIC (Our code)
Adds school context (schedules, homework data)
Prepares API request
↓
API COMMUNICATION
Sends request to OpenAI servers
Waits for response
↓
AI PROCESSING (OpenAI's servers)
Reads question + context
```

```
Generates intelligent answer
↓
RESPONSE HANDLING
Receives AI answer
Formats for display
↓
DISPLAY (Back to user)
Shows answer in chat interface
Saves to conversation history
```

Real Example:

1. User Action: Student clicks "What homework is assigned?"

2. Streamlit Captures:

```
if st.button("Homework"):
    query = "What homework is assigned?"
```

3. Our Code Adds Context:

```
context = f"""
School: Smart Academy
Homework Data: {HOMEWORK_ASSIGNMENTS}
"""
```

4. Send to OpenAI:

```
response = openai.chat.create(
    messages=[
        {"role": "system", "content": context},
        {"role": "user", "content": query}
    ]
)
```

5. Display Answer:

```
st.write(response.content)
# Shows: "Math: Exercises 1-20, page 45
#         English: Read chapters 5-7..."
```

Chapter 3: Python Programming Basics

3.1 Understanding Variables

What is a Variable? A variable is like a labeled box that stores information.

Real-World Analogy: Think of your school locker:

- Locker number = Variable name
- Contents inside = Variable value
- You can change what's inside anytime

In Python:

```
# Creating variables
student_name = "Alice"
grade = 10
has_homework = True

# Using variables
print(student_name)  # Output: Alice
print(grade)         # Output: 10
```

Variable Types:

1. Strings (Text):

```
school_name = "Smart Academy"
message = "Hello, World!"
address = "123 Main Street"

# You can combine strings
full_message = "Welcome to " + school_name
# Result: "Welcome to Smart Academy"
```

2. Integers (Whole Numbers):

```
student_count = 500
classrooms = 25
grade_level = 10

# You can do math
total = student_count + 50  # 550
```

3. Floats (Decimal Numbers):

```
temperature = 72.5
grade_percentage = 95.8
price = 19.99
```

4. Booleans (True/False):

```
is_student = True
has_permission = False
school_is_open = True
```

5. Lists (Collections):

```
# A list of items
students = ["Alice", "Bob", "Charlie"]
grades = [90, 85, 95, 88]
days = ["Monday", "Tuesday", "Wednesday"]

# Access items by position (starts at 0)
first_student = students[0]  # "Alice"
second_student = students[1]  # "Bob"
```

6. Dictionaries (Key-Value Pairs):

```
# Like a real dictionary: word → definition
student = {
    "name": "Alice",
    "grade": 10,
    "age": 15
}

# Access by key
student_name = student["name"]  # "Alice"
student_grade = student["grade"]  # 10
```

3.2 Functions

What is a Function? A function is a reusable piece of code that performs a specific task. It's like a recipe - once written, you can use it many times.

Real-World Analogy: Making a sandwich:

- **Inputs:** Bread, cheese, lettuce (ingredients)
- **Process:** Put cheese and lettuce between bread
- **Output:** Sandwich

Basic Function:

```
# Define the function
def greet_student(name):
    message = f"Hello, {name}! Welcome to school!"
    return message
```

```
# Use the function
result = greet_student("Alice")
print(result)  # Output: Hello, Alice! Welcome to school!
```

Breaking it Down:

1. `def` = "define" - tells Python you're creating a function
2. `greet_student` = function name (you choose this)
3. `(name)` = parameter - information the function needs
4. `return message` = what the function gives back

Function with Multiple Parameters:

```
def calculate_grade(homework, test, final):
    total = (homework * 0.3) + (test * 0.3) + (final * 0.4)
    return total

# Use it
grade = calculate_grade(90, 85, 92)
print(grade)  # Output: 89.5
```

Why Use Functions?

1. Reusability:

```
# Without function - repetitive
print("Hello, Alice! Welcome!")
print("Hello, Bob! Welcome!")
print("Hello, Charlie! Welcome!")

# With function - clean
def greet(name):
    print(f"Hello, {name}! Welcome!")

greet("Alice")
greet("Bob")
greet("Charlie")
```

2. Organization: Functions keep code organized and easy to understand.

3. Easier Testing: You can test one function at a time.

3.3 Control Flow (Making Decisions)

If Statements:

Programs need to make decisions based on conditions.

Real-World Example: "IF it's raining, THEN bring an umbrella, ELSE wear sunglasses"

In Python:

```
weather = "rainy"

if weather == "rainy":
    print("Bring an umbrella")
else:
    print("Wear sunglasses")
```

Multiple Conditions:

```
grade = 85

if grade >= 90:
    letter = "A"
elif grade >= 80:
    letter = "B"
elif grade >= 70:
    letter = "C"
else:
    letter = "F"

print(f"Your grade is: {letter}")
# Output: Your grade is: B
```

Comparison Operators:

```
==  # Equal to
!=  # Not equal to
>   # Greater than
<   # Less than
>=  # Greater than or equal
<=  # Less than or equal

# Examples
age = 15
age == 15   # True
age > 18    # False
age >= 15   # True
```

Logical Operators:

```
# AND - both must be true
age = 15
has_permission = True
```

```
if age >= 13 and has_permission:
    print("Can proceed")

# OR - at least one must be true
is_student = True
is_teacher = False

if is_student or is_teacher:
    print("Can access school system")

# NOT - reverse the condition
is_weekend = False

if not is_weekend:
    print("School day!")
```

3.4 Loops (Repeating Actions)

For Loops:

When you need to do something multiple times.

Real-World Example: Checking each student's attendance in a class roster.

Basic For Loop:

```
# Loop through a list
students = ["Alice", "Bob", "Charlie"]

for student in students:
    print(f"Good morning, {student}!")

# Output:
# Good morning, Alice!
# Good morning, Bob!
# Good morning, Charlie!
```

Loop Through Numbers:

```
# Count from 0 to 4
for i in range(5):
    print(f"Count: {i}")

# Output:
# Count: 0
# Count: 1
# Count: 2
# Count: 3
# Count: 4
```

Loop Through Dictionary:

```
homework = {
    "Math": "Page 45",
    "English": "Essay",
    "Science": "Lab report"
}

for subject, assignment in homework.items():
    print(f"{subject}: {assignment}")

# Output:
# Math: Page 45
# English: Essay
# Science: Lab report
```

While Loops:

Repeat while a condition is true.

```
count = 0

while count < 3:
    print(f"Count is: {count}")
    count = count + 1

# Output:
# Count is: 0
# Count is: 1
# Count is: 2
```

3.5 Working with Strings

String Formatting:

1. f-strings (Modern way):

```
name = "Alice"
age = 15

# Put variables directly in strings
message = f"My name is {name} and I'm {age} years old"
print(message)
# Output: My name is Alice and I'm 15 years old
```

2. String Methods:

```
text = "Hello World"

# Make uppercase
upper = text.upper()  # "HELLO WORLD"

# Make lowercase
lower = text.lower()  # "hello world"

# Replace text
new = text.replace("World", "Python")  # "Hello Python"

# Check if contains
has_hello = "Hello" in text  # True

# Split into list
words = text.split(" ")  # ["Hello", "World"]
```

3. Multi-line Strings:

```
long_text = """
This is a long message
that spans multiple lines.
Very useful for large text!
"""
```

3.6 Lists and Dictionaries (In Depth)

Lists - Ordered Collections:

```
# Create a list
grades = [90, 85, 95, 88]

# Add to list
grades.append(92)  # [90, 85, 95, 88, 92]

# Remove from list
grades.remove(85)  # [90, 95, 88, 92]

# Get length
count = len(grades)  # 4

# Sort list
grades.sort()  # [88, 90, 92, 95]

# Loop through
for grade in grades:
    print(f"Grade: {grade}")
```

List Comprehension (Advanced but useful):

```
# Create new list from existing
numbers = [1, 2, 3, 4, 5]

# Double each number
doubled = [n * 2 for n in numbers]
# Result: [2, 4, 6, 8, 10]

# Filter list
even_only = [n for n in numbers if n % 2 == 0]
# Result: [2, 4]
```

Dictionaries - Key-Value Storage:

```
# Create dictionary
student = {
    "name": "Alice",
    "grade": 10,
    "subjects": ["Math", "English", "Science"]
}

# Access values
name = student["name"]  # "Alice"

# Add new key-value
student["age"] = 15

# Update value
student["grade"] = 11

# Check if key exists
if "name" in student:
    print(student["name"])

# Get all keys
keys = student.keys()  # dict_keys(['name', 'grade', 'subjects', 'age'])

# Get all values
values = student.values()

# Loop through
for key, value in student.items():
    print(f"{key}: {value}")
```

Nested Dictionaries:

```
school = {
    "students": {
```

```
        "Alice": {"grade": 10, "age": 15},
        "Bob": {"grade": 11, "age": 16}
    },
    "classes": {
        "Math": {"room": 101, "teacher": "Mr. Smith"}
    }
}

# Access nested data
alice_grade = school["students"]["Alice"]["grade"]  # 10
```

Chapter 4: Web Development Fundamentals

4.1 How Websites Work

The Client-Server Model:

Analogy: Restaurant Service

- **Client (Your Browser):** The customer placing an order
- **Server (Web Server):** The kitchen preparing your meal
- **Response:** The waiter bringing your food

In Web Terms:

1. **Client (Browser) Requests:**
    - You type: `www.school.com`
    - Browser sends request to server
2. **Server Processes:**
    - Finds the website files
    - Runs necessary code
    - Prepares response
3. **Server Responds:**
    - Sends back HTML, CSS, JavaScript
    - Browser receives and displays
4. **Browser Renders:**
    - Shows the website to you

Our Application:

```
Student's Browser → Sends Question →  Our Streamlit App →
Processes → Calls OpenAI → Gets Answer → Sends to Browser → Displays Chat
```

4.2 HTML Basics (Structure)

What is HTML? HTML (HyperText Markup Language) is the structure of a web page. It's like the skeleton of a building.

Basic HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Page</title>
  </head>
  <body>
    <h1>Welcome to School</h1>
    <p>This is a paragraph of text.</p>
    <button>Click Me</button>
  </body>
</html>
```

Common HTML Elements:

Headings:

```
<h1>Main Title</h1>
<h2>Subtitle</h2>
<h3>Smaller heading</h3>
```

Paragraphs:

```
<p>This is a paragraph of text.</p>
```

Links:

```
<a href="https://school.com">Visit School Website</a>
```

Images:

```
<img src="logo.png" alt="School Logo">
```

Divisions (Containers):

```
<div class="container">
  <p>Content inside a container</p>
```

```
</div>
```

In Our App: We use HTML within Python (Streamlit allows this):

```
st.markdown("""
<div class="main-header">
  <h1>🎓 Smart Academy</h1>
  <p>Welcome to our chatbot!</p>
</div>
""", unsafe_allow_html=True)
```

4.3 CSS Basics (Styling)

What is CSS? CSS (Cascading Style Sheets) makes websites look good. It's like paint and decoration for a building.

CSS Syntax:

```
selector {
  property: value;
}
```

Example:

```
h1 {
  color: blue;
  font-size: 24px;
  text-align: center;
}
```

Common CSS Properties:

Colors:

```
color: red;              /* Text color */
background-color: blue;  /* Background color */
```

Sizing:

```
width: 300px;
height: 200px;
font-size: 16px;
```

Spacing:

```
margin: 10px;      /* Space outside */
padding: 15px;     /* Space inside */
```

Borders:

```
border: 1px solid black;
border-radius: 10px;     /* Rounded corners */
```

Layout:

```
text-align: center;
display: flex;
```

In Our App:

```
st.markdown("""
<style>
  .user-message {
    background-color: #007bff;
    color: white;
    padding: 10px;
    border-radius: 15px;
  }
</style>
""", unsafe_allow_html=True)
```

CSS Colors:

Named Colors:

```
color: red;
color: blue;
color: green;
```

Hex Colors:

```
color: #007bff;  /* Blue */
color: #FF0000;  /* Red */
color: #00FF00;  /* Green */
```

RGB Colors:

```
color: rgb(0, 123, 255);  /* Blue */
```

Gradients (Special Effects):

```
background: linear-gradient(90deg, #667eea 0%, #764ba2 100%);
/* Creates smooth color transition from purple-blue to purple */
```

4.4 Understanding Web Forms

What is a Form? A form collects user input and sends it somewhere to be processed.

Real-World Analogy: A paper form you fill out at school:

- Name field (text input)
- Grade level (dropdown)
- Submit button

HTML Form:

```
<form>
  <input type="text" placeholder="Enter your name">
  <select>
    <option>Grade 9</option>
    <option>Grade 10</option>
  </select>
  <button type="submit">Submit</button>
</form>
```

In Streamlit (Much Easier!):

```
with st.form("student_form"):
    name = st.text_input("Name:")
    grade = st.selectbox("Grade:", [9, 10, 11, 12])
    submit = st.form_submit_button("Submit")

    if submit:
        st.write(f"Hello {name} from grade {grade}!")
```

Why Use Forms?

1. **Groups related inputs together**
2. **Submits all at once (not one at a time)**
3. **Better user experience**

4. **Prevents accidental submissions**

In Our Chat Application:

```
with st.form("chat_form", clear_on_submit=True):
    user_input = st.text_input("Ask a question:")
    submit = st.form_submit_button("Send")

    if submit and user_input:
        # Process the question
        get_ai_response(user_input)
```

The form:

- Takes the question
- User clicks Send (or presses Enter)
- Clears the input box
- Processes the question

PART 2: BUILDING BLOCKS

Chapter 5: Setting Up Your Development Environment

5.1 Understanding Your Computer's Setup

What is an Operating System?

- **Windows, macOS, Linux** are operating systems
- They manage your computer's resources
- Run programs like Python

What is a Terminal/Command Line?

Think of it as a text-based way to control your computer, instead of clicking with a mouse.

On Windows: Command Prompt or PowerShell **On Mac/Linux:** Terminal

Why Use Terminal?

- Install software
- Run programs
- Manage files
- Professional development tool

Basic Commands:

Windows (Command Prompt):

```
dir                 # List files in directory
cd folder_name      # Change directory
mkdir new_folder    # Create folder
```

Mac/Linux (Terminal):

```
ls                  # List files
cd folder_name      # Change directory
mkdir new_folder    # Create folder
```

5.2 Installing Python

What is Python Installation? Installing Python gives your computer the ability to understand and run Python code.

Step-by-Step Installation:

1. Download Python:

- Go to: https://www.python.org/downloads/
- Click "Download Python 3.x.x" (latest version)
- File will download (about 25-30 MB)

2. Run Installer:

On Windows:

- Run downloaded file
- ⚠️ **IMPORTANT:** Check "Add Python to PATH"
- Click "Install Now"
- Wait for installation
- Click "Close"

On Mac:

- Run downloaded `.pkg` file
- Follow installer steps
- Enter password when prompted
- Click "Install"

3. Verify Installation:

Open Terminal/Command Prompt and type:

```
python --version
```

You should see:

```
Python 3.11.x
```

If you see this, Python is installed correctly!

Understanding PATH: "Adding to PATH" tells your computer where to find Python when you type `python` in the terminal.

Analogy: Like putting Python in a common folder where your computer always looks for programs.

5.3 Understanding pip (Package Manager)

What is pip? pip is Python's package installer. It's like an app store for Python code libraries.

Real-World Analogy:

- Your phone has an App Store
- You download apps you need
- pip does the same for Python packages

Check if pip is installed:

```
pip --version
```

Should show:

```
pip 23.x.x from ...
```

Installing Packages:

Single Package:

```
pip install streamlit
```

This downloads and installs Streamlit.

Multiple Packages:

```
pip install streamlit openai pandas
```

From requirements.txt:

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

This installs all packages listed in the file.

What is requirements.txt? A list of all Python packages your project needs.

Our requirements.txt:

```
streamlit==1.28.1
openai==1.3.0
python-dotenv==1.0.0
pandas==2.1.3
datetime
```

Format: `package_name==version_number`

Why Specify Versions?

- Ensures everyone uses same versions
- Prevents compatibility issues
- Makes project reproducible

5.4 Virtual Environments (Best Practice)

What is a Virtual Environment? A separate, isolated Python environment for your project.

Why Use It?

Problem:

- Project A needs package version 1.0
- Project B needs package version 2.0
- They conflict!

Solution:

- Create separate environment for each project
- Each has its own packages
- No conflicts!

Analogy: Like having separate toolboxes for different projects. The tools in one box don't interfere with another.

Creating Virtual Environment:

Windows:

```
python -m venv venv
venv\Scripts\activate
```

Mac/Linux:

```
python3 -m venv venv
source venv/bin/activate
```

When activated, you'll see:

```
(venv) C:\your\folder>
```

The `(venv)` shows it's active.

Installing Packages in Virtual Environment:

```
(venv) pip install -r requirements.txt
```

Now packages install only in this environment!

Deactivating:

```
deactivate
```

5.5 Code Editor Setup

What is a Code Editor? A program designed for writing code. It has features like:

- Syntax highlighting (colors code)
- Auto-completion
- Error detection

Recommended: Visual Studio Code (VS Code)

Why VS Code?

- Free and open source
- Powerful but beginner-friendly
- Great Python support
- Extensions for everything

Installation:

1. Go to: https://code.visualstudio.com/
2. Download for your OS
3. Run installer
4. Follow prompts

Essential Extensions:

1. Python Extension:

- Open VS Code
- Click Extensions icon (or Ctrl+Shift+X)
- Search "Python"
- Click "Install" on Microsoft's Python extension

2. Pylance:

- Provides better code completion
- Real-time error checking
- Type hints

3. Streamlit Extension (Optional):

- Syntax highlighting for Streamlit
- Code snippets

Using VS Code:

Opening a Project:

- File → Open Folder
- Select your project folder
- Files appear in left sidebar

Creating a File:

- Right-click in Explorer
- New File
- Name it `app.py`

Running Python:

- Open terminal in VS Code (Ctrl+`)
- Type: `python app.py`

Helpful Shortcuts:

- `Ctrl+S` - Save file
- `Ctrl+/` - Comment/uncomment line
- `Ctrl+Space` - Auto-complete
- `F5` - Run with debugger

5.6 Getting an OpenAI API Key

What is an API Key? A unique code that identifies you to OpenAI's service.

Analogy: Like a library card - it identifies you and tracks your usage.

Step-by-Step:

1. Create OpenAI Account:

- Go to: https://platform.openai.com/
- Click "Sign Up"
- Enter email and password
- Verify email

2. Add Payment Method:

- Go to Billing
- Add credit card
- OpenAI charges based on usage
- Very affordable for learning (a few dollars)

3. Generate API Key:

- Click your profile
- "API Keys"
- "Create new secret key"
- Name it (e.g., "School Chatbot")
- Copy the key (shows once!)

4. Save API Key Securely: Create `.env` file in your project:

```
OPENAI_API_KEY=sk-proj-your-key-here
```

⚠️ **SECURITY WARNING:**

- Never share your API key
- Never commit .env to GitHub
- Treat it like a password

Understanding API Costs:

OpenAI charges per "token" used:

- 1,000 tokens ≈ 750 words
- GPT-4o-mini: ~$0.00015 per 1K tokens
- Very cheap for learning!

Example:

- 100 questions/answers
- ~200,000 tokens total
- Cost: ~$0.03 (three cents!)

5.7 Project Folder Structure

Creating Your Project Folder:

```
My School Chatbot/
├── venv/                    # Virtual environment (created by you)
```

```
├── app.py                    # Main application
├── config.py                 # Configuration
├── school_data.py            # School information
├── requirements.txt          # Dependencies
├── .env                      # API keys (you create this)
├── .gitignore               # What to ignore in Git
└── README.md                # Project documentation
```

Why This Structure?

1. **Organized:** Easy to find files
2. **Standard:** Other developers understand it
3. **Maintainable:** Easy to update and debug
4. **Professional:** Industry best practice

Creating the Structure:

```
# Create project folder
mkdir "AI Chatbot for School Queries"
cd "AI Chatbot for School Queries"

# Create virtual environment
python -m venv venv

# Activate it
source venv/bin/activate  # Mac/Linux
venv\Scripts\activate     # Windows

# Create files
touch app.py
touch config.py
touch school_data.py
touch requirements.txt
touch .env
touch .gitignore
touch README.md
```

5.8 Understanding .gitignore

What is .gitignore? A file that tells Git which files NOT to track.

Why Needed?

- Don't upload sensitive data (API keys)
- Don't upload large files (virtual environment)
- Keep repository clean

Our .gitignore:

```
# Environment variables
.env

# Virtual environment
venv/
env/

# Python cache
__pycache__/
*.pyc

# Chat history
chat_history.pkl

# OS files
.DS_Store
```

What Each Line Means:

`.env`: Your API keys (security!) `venv/`: Virtual environment (too large, others create their own) `__pycache__/`: Python's compiled files (auto-generated) `*.pyc`: Python compiled files (not needed) `chat_history.pkl`: User data (privacy!) `.DS_Store`: Mac system file (unnecessary)

Chapter 6: Understanding Streamlit Framework

6.1 What Makes Streamlit Special?

Traditional Web Development:

```
Learn HTML → Learn CSS → Learn JavaScript →
Learn Backend (Flask/Django) → Combine Everything
= Months of learning
```

Streamlit:

```
Know Python → Use Streamlit → Create Web App
= Days of learning
```

Core Philosophy: "If you can write a Python script, you can create a web app"

6.2 Streamlit Basics

Creating Your First Streamlit App:

1. Create `hello.py`:

```
import streamlit as st

st.title("Hello, World!")
st.write("This is my first Streamlit app!")
```

2. Run it:

```
streamlit run hello.py
```

3. Browser Opens: You see your app running at `http://localhost:8501`

Understanding the Flow:

Behind the Scenes:

1. Streamlit starts a local web server
2. Server runs your Python script
3. Converts Python to HTML/CSS/JavaScript
4. Displays in browser
5. Re-runs script on any interaction

Key Concept: Script Re-runs

When you click a button or type in a box:

- Entire script runs again from top to bottom
- This is why we need `st.session_state` (covered later)

6.3 Essential Streamlit Widgets

6.3.1 Text Display:

```
import streamlit as st

# Title (biggest text)
st.title("🎓 School Chatbot")

# Header (medium text)
st.header("Welcome Students!")

# Subheader (smaller text)
st.subheader("Ask me anything")

# Regular text
st.write("This is regular text")

# Markdown (formatted text)
st.markdown("**Bold** and *italic* text")
```

```
# Code
st.code("print('Hello')", language="python")
```

Output Comparison:

- `title`: # Very Large Text
- `header`: ## Large Text
- `subheader`: ### Medium Text
- `write`: Normal text

6.3.2 Input Widgets:

Text Input:

```
# Simple text input
name = st.text_input("Enter your name:")
st.write(f"Hello, {name}!")

# With placeholder
question = st.text_input(
    "Ask a question:",
    placeholder="e.g., What's my homework?"
)
```

What Happens:

1. User types "Alice"
2. Variable `name` = "Alice"
3. Script re-runs
4. Shows "Hello, Alice!"

Number Input:

```
age = st.number_input("Enter your age:", min_value=0, max_value=120)
st.write(f"You are {age} years old")
```

Date Input:

```
from datetime import date

chosen_date = st.date_input("Select a date:")
st.write(f"You selected: {chosen_date}")
```

Text Area (Multi-line):

```
essay = st.text_area("Write your essay:")
word_count = len(essay.split())
st.write(f"Word count: {word_count}")
```

6.3.3 Selection Widgets:

Selectbox (Dropdown):

```
grade = st.selectbox(
    "Select your grade:",
    options=[9, 10, 11, 12]
)
st.write(f"You selected grade {grade}")
```

Radio Buttons:

```
subject = st.radio(
    "Favorite subject:",
    options=["Math", "English", "Science"]
)
```

Checkbox:

```
agree = st.checkbox("I agree to terms")

if agree:
    st.write("Thank you for agreeing!")
```

Multi-select:

```
languages = st.multiselect(
    "Languages you speak:",
    options=["English", "Spanish", "French", "Hindi"]
)
st.write(f"You speak: {languages}")
```

Slider:

```
confidence = st.slider(
    "How confident are you?",
    min_value=0,
    max_value=100,
```

```
    value=50  # Default value
)
```

6.3.4 Button Widgets:

Simple Button:

```
if st.button("Click Me"):
    st.write("Button was clicked!")
```

Understanding Button Behavior:

```
# This runs every time script runs
st.write("Script started")

# This only shows when button is clicked
if st.button("Say Hello"):
    st.write("Hello!")

# This runs every time script runs
st.write("Script ended")
```

Button with State:

```
if "count" not in st.session_state:
    st.session_state.count = 0

if st.button("Increment"):
    st.session_state.count += 1

st.write(f"Count: {st.session_state.count}")
```

Download Button:

```
data = "This is my file content"

st.download_button(
    label="Download File",
    data=data,
    file_name="myfile.txt",
    mime="text/plain"
)
```

6.4 Layout Components

6.4.1 Sidebar:

```
# Main area
st.title("Main Content")

# Sidebar
with st.sidebar:
    st.header("Sidebar")
    option = st.selectbox("Choose:", ["A", "B", "C"])
    st.button("Click")
```

Result:

- Left: Sidebar with controls
- Right: Main content area

6.4.2 Columns:

```
col1, col2 = st.columns(2)

with col1:
    st.header("Column 1")
    st.write("This is in column 1")

with col2:
    st.header("Column 2")
    st.write("This is in column 2")
```

Result: Page split into two equal columns

Unequal Columns:

```
col1, col2 = st.columns([3, 1])  # 3:1 ratio

with col1:
    st.write("This is wider")

with col2:
    st.write("Smaller")
```

6.4.3 Tabs:

```
tab1, tab2, tab3 = st.tabs(["Tab 1", "Tab 2", "Tab 3"])

with tab1:
    st.write("Content for tab 1")
```

```
with tab2:
    st.write("Content for tab 2")

with tab3:
    st.write("Content for tab 3")
```

6.4.4 Containers:

```
# Create container
container = st.container()

# Add to container
with container:
    st.write("This is in a container")
    st.button("Container button")

# Later, add more to same container
with container:
    st.write("Adding more content!")
```

6.4.5 Expander (Collapsible Section):

```
with st.expander("Click to expand"):
    st.write("Hidden content")
    st.write("Only shows when expanded")
```

6.5 Forms in Streamlit

Why Use Forms?

Without Form:

```
name = st.text_input("Name:")  # Re-runs on each keystroke!
age = st.number_input("Age:")  # Re-runs again!
# Inefficient!
```

With Form:

```
with st.form("myform"):
    name = st.text_input("Name:")
    age = st.number_input("Age:")
    submit = st.form_submit_button("Submit")

    if submit:
```

```
        st.write(f"Name: {name}, Age: {age}")
# Only re-runs when form is submitted!
```

Benefits:

1. More efficient (fewer re-runs)
2. Better user experience
3. Prevents partial submissions
4. Groups related inputs

Form Features:

Clear on Submit:

```
with st.form("form1", clear_on_submit=True):
    text = st.text_input("Enter text:")
    submit = st.form_submit_button("Submit")

    if submit:
        st.write(f"You entered: {text}")
        # Input clears after submit
```

Multiple Submit Buttons:

```
with st.form("form2"):
    text = st.text_input("Enter text:")
    save = st.form_submit_button("Save")
    cancel = st.form_submit_button("Cancel")

    if save:
        st.write("Saved!")
    if cancel:
        st.write("Cancelled!")
```

6.6 Session State (Crucial Concept!)

The Problem:

```
# This DOESN'T work as expected!
count = 0  # Resets to 0 every time

if st.button("Add"):
    count += 1  # Seems to work...

st.write(count)  # Always shows 0!
```

Why? Script re-runs completely on each interaction, resetting `count` to 0.

The Solution: Session State

```
# Initialize (runs once)
if "count" not in st.session_state:
    st.session_state.count = 0

# Modify
if st.button("Add"):
    st.session_state.count += 1  # Persists!

# Display
st.write(st.session_state.count)  # Shows correct value!
```

How it Works:

Session State is like a persistent dictionary:

```
st.session_state = {
    "count": 0,
    "name": "Alice",
    "messages": [],
    # ... any data you want to persist
}
```

Common Patterns:

1. Initialize:

```
if "key" not in st.session_state:
    st.session_state.key = initial_value
```

2. Read:

```
value = st.session_state.key
```

3. Update:

```
st.session_state.key = new_value
```

4. Check Existence:

```
if "key" in st.session_state:
```

```
    # Do something
```

Real Example (Chat Messages):

```
# Initialize messages list
if "messages" not in st.session_state:
    st.session_state.messages = []

# Display all messages
for msg in st.session_state.messages:
    st.write(f"{msg['role']}: {msg['content']}")

# Add new message
user_input = st.text_input("Type message:")
if st.button("Send"):
    st.session_state.messages.append({
        "role": "user",
        "content": user_input
    })
    # Messages persist across re-runs!
```

6.7 Streamlit Status Messages

Success:

```
if form_submitted:
    st.success("✅ Form submitted successfully!")
```

Info:

```
st.info("ℹ️ Please fill out all fields")
```

Warning:

```
st.warning("⚠️ Your session will expire in 5 minutes")
```

Error:

```
if password_wrong:
    st.error("❌ Incorrect password")
```

Exception (for developers):

```
try:
    risky_operation()
except Exception as e:
    st.exception(e)  # Shows full error details
```

6.8 Progress and Spinners

Spinner (Loading):

```
import time

with st.spinner("Processing..."):
    time.sleep(3)  # Simulate long operation

st.success("Done!")
```

Progress Bar:

```
progress = st.progress(0)

for i in range(100):
    time.sleep(0.01)
    progress.progress(i + 1)

st.success("Complete!")
```

In Our Chatbot:

```
with st.spinner("🤔 Thinking..."):
    response = get_ai_response(question)

st.write(response)
```

This shows a spinning animation while waiting for AI response.

6.9 Caching (Performance)

The Problem:

```
def load_large_data():
    # Takes 10 seconds to load
    time.sleep(10)
    return data
```

```
# This runs EVERY time script re-runs!
data = load_large_data()  # 10 seconds every time!
```

The Solution: Caching

```
@st.cache_data
def load_large_data():
    time.sleep(10)
    return data

# First run: Takes 10 seconds
# Subsequent runs: Instant! (uses cached version)
data = load_large_data()
```

How Caching Works:

1. **First Call:** Function runs, result is saved
2. **Next Calls:** If inputs are same, returns saved result
3. **Different Inputs:** Runs again, caches new result

When to Cache:

✅ **Good for:**

- Loading data files
- API calls (that don't change often)
- Complex calculations
- Database queries

❌ **Don't cache:**

- Random number generation
- Current time/date
- User-specific data that changes

Cache Types:

`@st.cache_data`**:** For data (lists, dictionaries, DataFrames)

```
@st.cache_data
def load_school_data():
    return SCHOOL_SCHEDULE
```

`@st.cache_resource`**:** For resources (connections, models)

```
@st.cache_resource
def init_openai_client():
```

```
    return openai.OpenAI(api_key="...")
```

Time-based Cache:

```
@st.cache_data(ttl=3600)  # Cache for 1 hour
def fetch_latest_events():
    return api.get_events()
```

6.10 Custom HTML/CSS in Streamlit

Adding HTML:

```
st.markdown("""
<div style="background: blue; padding: 10px; color: white;">
    This is custom HTML!
</div>
""", unsafe_allow_html=True)
```

⚠️ **`unsafe_allow_html=True` is required!**

Adding CSS:

```
st.markdown("""
<style>
.custom-class {
    background: linear-gradient(90deg, #667eea, #764ba2);
    padding: 2rem;
    border-radius: 10px;
    color: white;
}
</style>

<div class="custom-class">
    Beautiful gradient box!
</div>
""", unsafe_allow_html=True)
```

In Our Chatbot: We use this to create:

- Gradient header
- Chat message bubbles
- Avatar icons
- Custom styling

Chapter 7: Working with OpenAI API

7.1 Understanding APIs (Deep Dive)

What Exactly is an API?

Real-World Analogy - Restaurant:

You (Customer) want food but can't go into the kitchen.

Solution: A waiter (API)

- You tell waiter what you want (request)
- Waiter goes to kitchen (server)
- Kitchen prepares food (processes)
- Waiter brings food back (response)

In Programming:

Your App → Request → API → OpenAI's Servers → AI Processing → Response → API → Your App

Why Not Just Talk Directly to OpenAI?

1. **Security:** API handles authentication
2. **Simplicity:** API provides easy interface
3. **Standardization:** Same format for everyone
4. **Control:** API can rate-limit, log, manage

7.2 HTTP Basics (How APIs Communicate)

HTTP = HyperText Transfer Protocol

The "language" computers use to talk over the internet.

HTTP Methods:

GET: Retrieve data

```
"Hey server, give me the weather data"
```

POST: Send data to create something

```
"Hey server, here's a new user to create"
```

PUT: Update existing data

```
"Hey server, update user #123's email"
```

DELETE: Remove data

```
"Hey server, delete user #123"
```

For OpenAI, we use POST:

```
"Hey OpenAI, here's a question, send back an answer"
```

HTTP Request Structure:

```
POST https://api.openai.com/v1/chat/completions

Headers:
  Authorization: Bearer sk-your-api-key
  Content-Type: application/json

Body:
{
  "model": "gpt-4o-mini",
  "messages": [
    {"role": "user", "content": "Hello!"}
  ]
}
```

HTTP Response:

```
Status: 200 OK

Body:
{
  "choices": [
    {
      "message": {
        "content": "Hello! How can I help you?"
      }
    }
  ]
}
```

7.3 OpenAI API Structure

Installing OpenAI Package:

```
pip install openai==1.3.0
```

Basic Usage:

```python
import openai

# Initialize client
client = openai.OpenAI(api_key="sk-your-key")

# Make request
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "user", "content": "What is 2+2?"}
    ]
)

# Get answer
answer = response.choices[0].message.content
print(answer)  # "2+2 equals 4"
```

Understanding the Parameters:

1. model: Which AI to use:

- `gpt-4o-mini`: Fast, cheap, good quality
- `gpt-3.5-turbo`: Faster, cheaper, decent
- `gpt-4`: Best quality, slower, expensive

Think of it like cars:

- gpt-3.5-turbo: Economy car (cheap, efficient)
- gpt-4o-mini: Mid-range (balanced)
- gpt-4: Luxury car (expensive, best)

2. messages: The conversation history.

```python
messages=[
    {"role": "system", "content": "You are a helpful assistant"},
    {"role": "user", "content": "What's the capital of France?"},
    {"role": "assistant", "content": "Paris"},
    {"role": "user", "content": "What's its population?"}
]
```

Roles:

- **system:** Instructions for the AI
- **user:** What the human asks
- **assistant:** What the AI responds

3. temperature: How "creative" the AI should be (0.0 to 1.0)

```
temperature=0.0  # Very focused, deterministic
# Q: "What's 2+2?"
# A: "4" (always the same)

temperature=0.7  # Balanced (our choice)
# Q: "Write a greeting"
# A: "Hello!", "Hi there!", "Greetings!" (varies)

temperature=1.0  # Very creative, random
# Q: "Write a greeting"
# A: "Salutations, friend!", "Howdy partner!", etc. (very varied)
```

For school chatbot, we use 0.7:

- Creative enough to be natural
- Focused enough to be accurate

4. max_tokens: Maximum length of response

```
max_tokens=100  # Short response (~75 words)
max_tokens=1000  # Longer response (~750 words)
```

Why limit?

- Costs money per token
- Prevents extremely long responses
- Controls response time

Our choice: 1000 tokens

- Enough for detailed answers
- Not so long it's overwhelming

7.4 Advanced OpenAI Features

System Messages (Critical!):

The system message is like training the AI for your specific use case.

Generic (Not Good):

```
{"role": "system", "content": "You are helpful"}
```

Specific (Better):

```
{"role": "system", "content": """
You are a helpful assistant for Smart Academy high school.
```

```
You answer questions about:
- Class schedules
- Homework assignments
- School events
- School policies

Always be friendly and concise.
If you don't know something, say so.
"""}
```

With Data (Best!):

```
{"role": "system", "content": f"""
You are a helpful assistant for {school_name}.

Available Information:
- Today's Schedule: {schedule}
- Upcoming Events: {events}
- Homework Due: {homework}

Guidelines:
1. Be friendly and helpful
2. Use the provided data to answer
3. Be concise but informative
4. If unsure, suggest contacting the office
"""}
```

Conversation History:

AI doesn't remember previous messages unless you include them!

Single Message (No Context):

```
# First question
response1 = client.chat.completions.create(
    messages=[
        {"role": "user", "content": "What's the capital of France?"}
    ]
)
# Answer: "Paris"

# Second question
response2 = client.chat.completions.create(
    messages=[
        {"role": "user", "content": "What's its population?"}
    ]
)
# Answer: "I don't know what you're referring to" ❌
```

With Context (Works!):

```
messages = [
    {"role": "user", "content": "What's the capital of France?"},
    {"role": "assistant", "content": "Paris"},
    {"role": "user", "content": "What's its population?"}
]

response = client.chat.completions.create(messages=messages)
# Answer: "Paris has about 2.1 million people" ✅
```

In Our Chatbot:

```
# Build conversation history
conversation = [{"role": "system", "content": system_prompt}]

# Add all previous messages
for msg in st.session_state.messages:
    conversation.append({
        "role": msg["role"],
        "content": msg["content"]
    })

# Add new question
conversation.append({
    "role": "user",
    "content": user_question
})

# Get response with full context
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=conversation
)
```

7.5 Error Handling with APIs

Common API Errors:

1. Authentication Error:

```
# Wrong API key
openai.AuthenticationError: Invalid API key
```

Fix:

```
# Check .env file
# Verify API key is correct
# Ensure key starts with 'sk-'
```

2. Rate Limit Error:

```
# Too many requests
openai.RateLimitError: Rate limit exceeded
```

Fix:

```
# Wait before retrying
time.sleep(1)
```

3. Connection Error:

```
# Internet problem
openai.ConnectionError: Connection failed
```

Fix:

```
# Check internet connection
# Try again
```

4. Timeout Error:

```
# Request took too long
openai.TimeoutError: Request timed out
```

Fix:

```
# Increase timeout
# Simplify request
```

Handling Errors (Basic):

```
try:
    response = client.chat.completions.create(...)
```

```
except openai.AuthenticationError:
    print("Check your API key!")
except openai.RateLimitError:
    print("Slow down, too many requests!")
except openai.ConnectionError:
    print("Check your internet!")
except Exception as e:
    print(f"Something went wrong: {e}")
```

Handling Errors (Advanced - With Retry):

```
import time

def get_response_with_retry(messages, max_retries=3):
    """Try up to 3 times before giving up"""

    for attempt in range(max_retries):
        try:
            response = client.chat.completions.create(
                model="gpt-4o-mini",
                messages=messages
            )
            return response.choices[0].message.content

        except openai.RateLimitError:
            if attempt < max_retries - 1:
                wait_time = (attempt + 1) * 2  # 2, 4, 6 seconds
                print(f"Rate limited. Waiting {wait_time} seconds...")
                time.sleep(wait_time)
            else:
                return "Too many requests. Please try again later."

        except openai.ConnectionError:
            if attempt < max_retries - 1:
                print(f"Connection failed. Retrying...")
                time.sleep(1)
            else:
                return "Connection failed. Check your internet."

        except Exception as e:
            return f"Error: {str(e)}"

    return "Unable to get response after multiple attempts."
```

Using it:

```
answer = get_response_with_retry(messages)
st.write(answer)
```

Why Retry?

- Networks are unreliable
- Temporary glitches happen
- Most errors resolve if you try again
- Better user experience

7.6 Token Management

What are Tokens?

Tokens are pieces of text the AI processes.

Examples:

```
"Hello" = 1 token
"Hello, world!" = 4 tokens
"artificial intelligence" = 2 tokens
```

Rule of Thumb:

- 1 token ≈ 4 characters
- 1 token ≈ 0.75 words
- 100 tokens ≈ 75 words

Why Care About Tokens?

1. **Cost:** OpenAI charges per token
2. **Limits:** Models have max token limits
3. **Speed:** More tokens = slower response

Token Limits:

gpt-4o-mini:

- Max: 16,385 tokens
- Input + Output combined

Example:

```
Input: 1000 tokens (your question + context)
Output: 500 tokens (AI's answer)
Total: 1500 tokens used
```

Counting Tokens:

```
# Install tiktoken
# pip install tiktoken
```

```
import tiktoken

def count_tokens(text, model="gpt-4o-mini"):
    encoding = tiktoken.encoding_for_model(model)
    return len(encoding.encode(text))

text = "Hello, how are you?"
tokens = count_tokens(text)
print(f"Tokens: {tokens}")  # Output: 6
```

Managing Token Usage:

1. Limit max_tokens:

```
response = client.chat.completions.create(
    max_tokens=500  # Keep responses short
)
```

2. Trim old messages:

```
# Keep only last 10 messages
recent_messages = conversation_history[-10:]
```

3. Summarize long context:

```
if len(context) > 1000:
    # Summarize instead of including everything
    context = summarize(context)
```

Cost Calculation:

gpt-4o-mini Pricing (as of 2024):

- Input: $0.000150 per 1K tokens
- Output: $0.000600 per 1K tokens

Example Calculation:

```
Question with context: 1,000 tokens input
AI Response: 500 tokens output

Cost:
Input: 1,000 × $0.00015 = $0.00015
Output: 500 × $0.0006 = $0.0003
Total: $0.00045 (less than a penny!)
```

```
For 100 questions:
100 × $0.00045 = $0.045 (about 5 cents)
```

Very affordable for learning and small projects!

Chapter 8: Data Management & Storage

8.1 Understanding Data Structures

What is Data? Information your application needs to function.

Our School Chatbot Needs:

- School schedules
- Homework assignments
- Events calendar
- Contact information
- School policies

Where to Store Data?

Option 1: Hard-coded in Python

```
HOMEWORK = "Math: Page 45, English: Essay"
```

❌ Hard to update ❌ Not scalable ✅ Simple for small data

Option 2: Python Dictionaries (Our Choice)

```
HOMEWORK = {
    "Math": {
        "assignment": "Complete exercises 1-20",
        "page": 45,
        "due_date": "2024-10-25"
    },
    "English": {
        "assignment": "Write essay on Shakespeare",
        "due_date": "2024-10-27"
    }
}
```

✅ Organized ✅ Easy to access ✅ Flexible structure

Option 3: Database (Advanced)

```
# SQLite, PostgreSQL, MongoDB
```

```
db.query("SELECT * FROM homework WHERE subject = 'Math'")
```

✅ Best for large data ✅ Can handle millions of records ❌ More complex setup

For Learning: Dictionaries are Perfect!

8.2 School Data Structure

Let's Build Our School Data:

File: `school_data.py`

```
# School Schedule
SCHOOL_SCHEDULE = {
    "Monday": {
        "1st Period": {"time": "8:00 AM - 8:50 AM", "subject": "Math"},
        "2nd Period": {"time": "9:00 AM - 9:50 AM", "subject": "English"},
        "3rd Period": {"time": "10:00 AM - 10:50 AM", "subject":
"Science"},
        "Lunch": {"time": "11:00 AM - 11:45 AM"},
        "4th Period": {"time": "12:00 PM - 12:50 PM", "subject":
"History"},
        "5th Period": {"time": "1:00 PM - 1:50 PM", "subject": "PE"}
    },
    "Tuesday": {
        # Similar structure
    },
    # ... other days
}
```

Why This Structure?

1. **Nested Dictionary:**
    - Top level: Day of week
    - Second level: Period
    - Third level: Details
2. **Easy Access:**

```
# Get Monday's first period
monday_first = SCHOOL_SCHEDULE["Monday"]["1st Period"]
# Returns: {"time": "8:00 AM - 8:50 AM", "subject": "Math"}

# Get just the subject
subject = SCHOOL_SCHEDULE["Monday"]["1st Period"]["subject"]
# Returns: "Math"
```

3. **Easy to Loop:**

```
for day, periods in SCHOOL_SCHEDULE.items():
    print(f"\n{day}:")
    for period, details in periods.items():
        print(f"  {period}: {details}")
```

Homework Assignments:

```
HOMEWORK_ASSIGNMENTS = {
    "Math": {
        "assignment": "Complete exercises 1-20 on page 45",
        "due_date": "2024-10-25",
        "points": 20,
        "class": "Algebra II"
    },
    "English": {
        "assignment": "Write a 500-word essay on Shakespeare's influence",
        "due_date": "2024-10-27",
        "points": 50,
        "class": "English Literature"
    },
    "Science": {
        "assignment": "Lab report on chemical reactions",
        "due_date": "2024-10-28",
        "points": 30,
        "class": "Chemistry"
    }
}
```

Events Calendar:

```
UPCOMING_EVENTS = [
    {
        "name": "Science Fair",
        "date": "2024-11-15",
        "time": "2:00 PM - 5:00 PM",
        "location": "Gymnasium",
        "description": "Annual science fair showcasing student projects"
    },
    {
        "name": "Parent-Teacher Conference",
        "date": "2024-11-20",
        "time": "4:00 PM - 7:00 PM",
        "location": "Classrooms",
        "description": "Meet with teachers to discuss student progress"
    },
    {
        "name": "Winter Break",
        "date": "2024-12-20 to 2025-01-05",
        "description": "School closed for winter holidays"
```

```
    }
]
```

Why List of Dictionaries?

- Events are sequential (ordered)
- Each event has same structure
- Easy to add/remove events

School Policies:

```
SCHOOL_POLICIES = {
    "attendance": {
        "title": "Attendance Policy",
        "details": """
        - Students must attend at least 90% of classes
        - 3 tardies = 1 absence
        - Absences must be excused with a note
        - Contact office for extended absences
        """
    },
    "homework": {
        "title": "Homework Policy",
        "details": """
        - Late homework loses 10% per day
        - Maximum 3 days late accepted
        - Extensions available with teacher approval
        """
    },
    "dress_code": {
        "title": "Dress Code",
        "details": """
        - Appropriate length shorts/skirts (fingertip rule)
        - No offensive language on clothing
        - Closed-toe shoes required
        - ID badges must be visible
        """
    }
}
```

Contact Information:

```
CONTACT_INFO = {
    "main_office": {
        "department": "Main Office",
        "phone": "(555) 123-4567",
        "email": "office@smartacademy.edu",
        "hours": "7:30 AM - 4:00 PM"
    },
    "counseling": {
```

```
        "department": "Counseling",
        "phone": "(555) 123-4568",
        "email": "counseling@smartacademy.edu",
        "hours": "8:00 AM - 3:30 PM"
    },
    "nurse": {
        "department": "Nurse's Office",
        "phone": "(555) 123-4569",
        "email": "nurse@smartacademy.edu",
        "hours": "8:00 AM - 3:00 PM"
    }
}
```

8.3 Importing and Using Data

In `app.py`, import the data:

```
from school_data import (
    SCHOOL_SCHEDULE,
    HOMEWORK_ASSIGNMENTS,
    UPCOMING_EVENTS,
    SCHOOL_POLICIES,
    CONTACT_INFO
)
```

Now you can use it:

```
# Display today's schedule
st.write("Today's Schedule:")
for period, details in SCHOOL_SCHEDULE["Monday"].items():
    st.write(f"{period}: {details}")

# Show homework
st.write("Homework Due:")
for subject, hw in HOMEWORK_ASSIGNMENTS.items():
    st.write(f"{subject}: {hw['assignment']}")
```

8.4 Persisting Chat History (Saving to File)

The Problem: When you close the app, all chat history is lost!

The Solution: Save chat history to a file.

Python's Pickle Module:

Pickle converts Python objects to binary files.

```
import pickle

# Save to file
with open("chat_history.pkl", "wb") as f:
    pickle.dump(st.session_state.messages, f)

# Load from file
with open("chat_history.pkl", "rb") as f:
    messages = pickle.load(f)
```

Understanding the Code:

`"wb"`: Write Binary mode `"rb"`: Read Binary mode `pickle.dump()`: Save object to file `pickle.load()`: Load object from file

Complete Implementation:

```
import pickle
import os

CHAT_HISTORY_FILE = "chat_history.pkl"

def save_chat_history():
    """Save messages to file"""
    try:
        with open(CHAT_HISTORY_FILE, "wb") as f:
            pickle.dump(st.session_state.messages, f)
    except Exception as e:
        print(f"Error saving: {e}")

def load_chat_history():
    """Load messages from file"""
    if os.path.exists(CHAT_HISTORY_FILE):
        try:
            with open(CHAT_HISTORY_FILE, "rb") as f:
                return pickle.load(f)
        except Exception as e:
            print(f"Error loading: {e}")
            return []
    return []

def clear_chat_history():
    """Delete chat history file"""
    if os.path.exists(CHAT_HISTORY_FILE):
        os.remove(CHAT_HISTORY_FILE)
    st.session_state.messages = []
```

Using It:

```
# Initialize messages from saved history
if "messages" not in st.session_state:
    st.session_state.messages = load_chat_history()

# After adding new message
st.session_state.messages.append(new_message)
save_chat_history()  # Save immediately!

# Clear button
if st.button("Clear Chat"):
    clear_chat_history()
    st.rerun()  # Refresh the page
```

Why `st.rerun()`? Refreshes the page to show changes immediately.

8.5 Working with JSON (Alternative to Pickle)

What is JSON? JavaScript Object Notation - a human-readable data format.

JSON vs Pickle:

JSON:

```
{
  "name": "Alice",
  "age": 15,
  "grades": [90, 85, 95]
}
```

✅ Human-readable ✅ Works across languages ❌ Limited data types

Pickle:

```
Binary gibberish: x80\x03]q\x00...
```

✅ Saves any Python object ✅ Preserves exact structure ❌ Not human-readable ❌ Python-only

Using JSON:

```
import json

# Save to JSON
data = {
    "name": "Alice",
    "messages": ["Hello", "How are you?"]
}
```

```
with open("data.json", "w") as f:
    json.dump(data, f, indent=2)  # indent=2 makes it pretty

# Load from JSON
with open("data.json", "r") as f:
    loaded_data = json.load(f)
```

JSON File Output:

```
{
  "name": "Alice",
  "messages": [
    "Hello",
    "How are you?"
  ]
}
```

When to Use Each:

Use Pickle for:

- Chat history (complex structure)
- Session state
- Python-specific data

Use JSON for:

- Configuration files
- Data sharing with other apps
- Human-editable data

8.6 Environment Variables (.env)

What are Environment Variables? Configuration values stored outside your code.

Why Use Them?

❌ **Bad - API Key in Code:**

```
api_key = "sk-proj-abc123xyz"  # NEVER DO THIS!
```

Problems:

- Anyone who sees code gets your key
- If you share code on GitHub, key is public
- Security nightmare!

✅ **Good - API Key in .env:**

.env file:

```
OPENAI_API_KEY=sk-proj-abc123xyz
SCHOOL_NAME=Smart Academy
```

In code:

```
import os
from dotenv import load_dotenv

# Load .env file
load_dotenv()

# Access variables
api_key = os.getenv("OPENAI_API_KEY")
school_name = os.getenv("SCHOOL_NAME")
```

Benefits:

- API key stays private
- Easy to change configuration
- Different settings for different environments

Setting Up .env:

1. Install python-dotenv:

```
pip install python-dotenv
```

2. Create `.env` file:

```
OPENAI_API_KEY=your-key-here
SCHOOL_NAME=Smart Academy
DEBUG_MODE=False
```

3. Add to `.gitignore`:

```
.env
```

This ensures `.env` never goes to GitHub!

4. Create `.env.example`:

```
OPENAI_API_KEY=your-api-key-here
SCHOOL_NAME=Your School Name
DEBUG_MODE=False
```

This template helps others set up their own `.env`.

5. Use in code:

```
from dotenv import load_dotenv
import os

load_dotenv()

API_KEY = os.getenv("OPENAI_API_KEY")
SCHOOL_NAME = os.getenv("SCHOOL_NAME", "Default School")  # Default if not
found
DEBUG = os.getenv("DEBUG_MODE") == "True"
```

Chapter 9: Project Architecture Explained

9.1 Understanding Project Structure

What is Architecture? How you organize your code and files.

Real-World Analogy: Building a house:

- Foundation (core functionality)
- Rooms (separate modules)
- Wiring (connections between parts)
- Blueprint (documentation)

Our Project Structure:

```
AI Chatbot for School Queries/
│
├── venv/                      # Virtual environment (isolated packages)
│   └── ...
│
├── app.py                     # Main application (the house)
├── config.py                  # Configuration (settings)
├── school_data.py             # School information (database)
│
├── requirements.txt           # Dependencies (materials list)
├── .env                       # Secrets (keys to the house)
├── .gitignore                # What to ignore (trash)
│
├── README.md                  # Documentation (user manual)
├── SETUP.md                   # Setup guide (assembly instructions)
```

```
├── LICENSE                    # License (legal stuff)
│
├── chat_history.pkl           # Saved chats (memory)
│
└── COMPLETE_LEARNING_GUIDE.md # This guide!
```

9.2 Separation of Concerns

Principle: Each file should have ONE clear purpose.

Why?

❌ **Everything in One File (Bad):**

```
# app.py - 2000 lines of code!

OPENAI_API_KEY = "sk-..."  # Config
HOMEWORK = {...}           # Data
def get_response():        # Logic
    ...
st.title("School")         # UI
```

Problems:

- Hard to find things
- Hard to maintain
- Hard to test
- Confusing!

✅ **Separated (Good):**

config.py:

```
# ONLY configuration
OPENAI_API_KEY = "..."
SCHOOL_NAME = "Smart Academy"
MODEL = "gpt-4o-mini"
```

school_data.py:

```
# ONLY data
HOMEWORK = {...}
SCHEDULE = {...}
EVENTS = [...]
```

app.py:

```
# ONLY application logic + UI
from config import *
from school_data import *

def get_response():
    # Logic here

# UI here
st.title("School Chatbot")
```

Benefits:

- Easy to find things
- Easy to update
- Can test separately
- Clear organization

9.3 File-by-File Breakdown

9.3.1 config.py - Configuration

Purpose: Store all settings in one place

```
import os
from dotenv import load_dotenv

# Load environment variables
load_dotenv()

# OpenAI Configuration
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY", "")

# School Configuration
SCHOOL_NAME = "Smart Academy"
SCHOOL_ADDRESS = "123 Education Street"
SCHOOL_PHONE = "(555) 123-4567"
SCHOOL_EMAIL = "info@smartacademy.edu"

# Chat Configuration
MAX_TOKENS = 1000
TEMPERATURE = 0.7
MODEL = "gpt-4o-mini"
```

Why Separate File?

- Easy to change settings
- No need to hunt through code
- Can swap configurations easily

9.3.2 school_data.py - Data Storage

Purpose: All school information

```
# School schedules
SCHOOL_SCHEDULE = {...}

# Homework
HOMEWORK_ASSIGNMENTS = {...}

# Events
UPCOMING_EVENTS = [...]

# Policies
SCHOOL_POLICIES = {...}

# Contacts
CONTACT_INFO = {...}
```

Why Separate File?

- Data changes frequently
- Non-programmers can update
- Keeps app.py clean

9.3.3 app.py - Main Application

Purpose: The brain - combines everything

Structure:

```
# 1. Imports
import streamlit as st
import openai
from config import *
from school_data import *

# 2. Helper Functions
def save_chat_history():
    """Save chats to file"""
    ...

def load_chat_history():
    """Load chats from file"""
    ...

def get_openai_response(message):
    """Get AI response"""
    ...

def display_chat_message(message, is_user):
    """Display a chat bubble"""
    ...
```

```
# 3. Initialize Session State
if "messages" not in st.session_state:
    st.session_state.messages = load_chat_history()

if "selected_language" not in st.session_state:
    st.session_state.selected_language = "English"

# 4. Page Configuration
st.set_page_config(
    page_title="School Chatbot",
    page_icon="🎓",
    layout="wide"
)

# 5. Custom CSS
st.markdown("""<style>...</style>""", unsafe_allow_html=True)

# 6. Main Application
def main():
    # Sidebar
    with st.sidebar:
        # Controls

    # Main Area
    # Header
    # Chat display
    # Input form

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

Why This Structure?

1. **Imports at Top:** Easy to see dependencies
2. **Functions First:** Reusable code
3. **Initialization:** Set up before UI
4. **Configuration:** Page settings
5. **Styling:** Visual appearance
6. **Main Function:** The actual app

9.4 Data Flow

Understanding How Data Moves:

```
User Types Question
        ↓
Streamlit Captures Input (app.py)
        ↓
Add School Context (school_data.py)
        ↓
Format for OpenAI (app.py)
        ↓
```

```
Send to OpenAI API (openai package)
        ↓
Receive Response
        ↓
Format for Display (app.py)
        ↓
Show to User (Streamlit)
        ↓
Save to History (pickle file)
```

Code Flow:

```python
# 1. User input
user_question = st.text_input("Ask:")

# 2. Process
if user_question:
    # 3. Add to session
    st.session_state.messages.append({
        "role": "user",
        "content": user_question
    })

    # 4. Get AI response
    response = get_openai_response(user_question)

    # 5. Add response to session
    st.session_state.messages.append({
        "role": "assistant",
        "content": response
    })

    # 6. Save history
    save_chat_history()

    # 7. Display
    display_chat_message(user_question, is_user=True)
    display_chat_message(response, is_user=False)
```

9.5 Function Design Principles

Good Functions are:

1. Single Purpose

```python
# ❌ Bad - does too much
def process_everything(message):
    response = call_api(message)
    save_to_file(response)
    display_message(response)
```

```
    send_email(response)
    return response

# ✅ Good - one thing
def get_api_response(message):
    return call_api(message)
```

2. Clear Names

```
# ❌ Bad
def do_stuff(x):
    ...

# ✅ Good
def get_openai_response(user_message):
    ...
```

3. Small (10-50 lines ideal)

```
# ✅ Good - focused and readable
def save_chat_history():
    """Save messages to pickle file"""
    try:
        with open(CHAT_HISTORY_FILE, "wb") as f:
            pickle.dump(st.session_state.messages, f)
    except Exception as e:
        st.error(f"Error saving: {e}")
```

4. Well-Documented

```
def get_openai_response(user_message, context="", max_retries=3):
    """
    Get response from OpenAI API with retry mechanism

    Args:
        user_message (str): The question from the user
        context (str): Additional context for the AI
        max_retries (int): Number of retry attempts

    Returns:
        str: AI's response or error message
    """
    # Implementation...
```

5. Handle Errors

```
def load_chat_history():
    """Load chat history from file"""
    try:
        if os.path.exists(CHAT_HISTORY_FILE):
            with open(CHAT_HISTORY_FILE, "rb") as f:
                return pickle.load(f)
    except Exception as e:
        st.warning(f"Could not load history: {e}")
    return []  # Return empty list on error
```

Chapter 10: Building the User Interface

10.1 UI/UX Principles

UI = User Interface: What users see **UX = User Experience:** How users feel

Good UI/UX Principles:

1. Simplicity

- Don't overwhelm with options
- Clear, obvious controls
- Minimal clicks to accomplish tasks

2. Consistency

- Same colors throughout
- Same button styles
- Predictable behavior

3. Feedback

- Show loading states
- Confirm actions
- Display errors clearly

4. Accessibility

- Good color contrast
- Large enough text
- Clear labels

10.2 Page Configuration

```
st.set_page_config(
    page_title="🎓 Smart Academy Chatbot",  # Browser tab title
    page_icon="🎓",                          # Browser tab icon
    layout="wide",                           # Use full width
```

```
    initial_sidebar_state="expanded"         # Sidebar starts open
)
```

Why Each Setting:

`page_title`: Shows in browser tab, bookmarks **`page_icon`:** Visual identifier in tabs **`layout="wide"`:** More space for content **`initial_sidebar_state`:** Controls visibility

10.3 Custom Styling with CSS

Adding Custom CSS:

```
st.markdown("""
<style>
    /* Your custom styles here */
    .main-header {
        background: linear-gradient(90deg, #667eea 0%, #764ba2 100%);
        padding: 2rem;
        border-radius: 10px;
        color: white;
        text-align: center;
    }
</style>
""", unsafe_allow_html=True)
```

Key Styles in Our App:

1. Gradient Header:

```
.main-header {
    background: linear-gradient(90deg, #667eea 0%, #764ba2 100%);
    padding: 2rem;
    border-radius: 10px;
    color: white;
    text-align: center;
    margin-bottom: 2rem;
}
```

Breakdown:

- `linear-gradient`: Smooth color transition
- `90deg`: Left to right
- `#667eea` to `#764ba2`: Purple-blue gradient
- `padding: 2rem`: Space inside
- `border-radius: 10px`: Rounded corners
- `color: white`: White text
- `text-align: center`: Centered text

2. Chat Message Bubbles:

User Messages (Right side, blue):

```css
.user-message {
    background-color: #007bff;
    color: white;
    padding: 0.75rem 1rem;
    border-radius: 15px 15px 0 15px;
    margin-left: auto;
    max-width: 70%;
    box-shadow: 0 2px 5px rgba(0,0,0,0.1);
}
```

AI Messages (Left side, gray):

```css
.bot-message {
    background-color: #f0f0f0;
    color: #333;
    padding: 0.75rem 1rem;
    border-radius: 15px 15px 15px 0;
    max-width: 70%;
    box-shadow: 0 2px 5px rgba(0,0,0,0.1);
}
```

Why Different Border Radius? Creates speech bubble effect:

- User: `15px 15px 0 15px` (bottom-right sharp)
- Bot: `15px 15px 15px 0` (bottom-left sharp)

3. Avatars:

```css
.message-avatar {
    width: 40px;
    height: 40px;
    border-radius: 50%;
    display: flex;
    align-items: center;
    justify-content: center;
    font-size: 1.5rem;
    flex-shrink: 0;
}

.user-avatar {
    background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
}

.bot-avatar {
    background: linear-gradient(135deg, #f093fb 0%, #f5576c 100%);
}
```

Breakdown:

- `width/height: 40px`: Square size
- `border-radius: 50%`: Makes circle
- `display: flex`: Centers emoji
- Gradients: Pretty backgrounds

4. Quick Action Buttons:

```
.quick-action-btn {
    background: white;
    border: 2px solid #667eea;
    color: #667eea;
    padding: 0.5rem 1rem;
    border-radius: 20px;
    cursor: pointer;
    transition: all 0.3s ease;
}

.quick-action-btn:hover {
    background: #667eea;
    color: white;
    transform: translateY(-2px);
    box-shadow: 0 4px 8px rgba(0,0,0,0.2);
}
```

Interactive Effects:

- `transition`: Smooth animation
- `:hover`: Changes on mouse over
- `transform: translateY(-2px)`: Lifts up slightly
- `box-shadow`: Adds depth

10.4 Building the Header

```
st.markdown("""
<div class="main-header">
    <h1>🎓 Smart Academy Chatbot</h1>
    <p>Your AI assistant for school questions</p>
    <p style="font-size: 0.9rem; opacity: 0.9;">
        Ask about schedules, homework, events, and more!
    </p>
</div>
""", unsafe_allow_html=True)
```

Result:

- Eye-catching gradient
- Clear title

- Helpful description
- Professional appearance

10.5 Creating the Sidebar

```
with st.sidebar:
    # Logo/Title
    st.markdown("## 📚 Controls")

    # Message counter
    message_count = len(st.session_state.messages)
    st.metric("Total Messages", message_count)

    # Language selector
    st.session_state.selected_language = st.selectbox(
        "🌐 Language:",
        ["English", "Spanish", "French", "Hindi"]
    )

    # Search
    search_query = st.text_input("🔍 Search chats:", "")

    # Clear chat button
    if st.button("🗑️ Clear Chat", use_container_width=True):
        clear_chat_history()
        st.rerun()

    # Download transcript
    if st.session_state.messages:
        transcript = create_transcript()
        st.download_button(
            label="📥 Download Transcript",
            data=transcript,

file_name=f"chat_transcript_{datetime.now().strftime('%Y%m%d')}.txt",
            mime="text/plain",
            use_container_width=True
        )

    # Quick actions
    st.markdown("### ⚡ Quick Actions")

    quick_actions = [
        "📅 What's today's schedule?",
        "📝 What homework is assigned?",
        "🎉 What events are coming up?",
        "📞 How can I contact the school?"
    ]

    for action in quick_actions:
        if st.button(action, use_container_width=True):
```

```
            # Submit this query
            st.session_state.quick_action = action.split(" ", 1)[1]
```

Features:

1. **Message Counter:** Shows engagement
2. **Language Selector:** Accessibility
3. **Search:** Find old messages
4. **Clear Chat:** Start fresh
5. **Download:** Save conversations
6. **Quick Actions:** Common questions

10.6 Chat Display Area

```
# Container for chat messages
chat_container = st.container()

with chat_container:
    # Display all messages
    for i, message in enumerate(st.session_state.messages):
        is_user = message["role"] == "user"

        display_chat_message(
            message=message["content"],
            is_user=is_user,
            timestamp=message.get("timestamp", ""),
            message_id=f"msg_{i}"
        )

# Auto-scroll to bottom
st.markdown("""
<script>
    var element = window.parent.document.querySelector('[data-
testid="stVerticalBlock"]');
    element.scrollTop = element.scrollHeight;
</script>
""", unsafe_allow_html=True)
```

Why Container?

- Groups messages together
- Easier to style
- Better scrolling control

Auto-scroll JavaScript:

- Scrolls to newest message automatically
- Better UX (don't have to manually scroll)

10.7 Chat Message Display Function

```
def display_chat_message(message, is_user=True, timestamp=None,
message_id=None):
    """Display a single chat message with avatar and timestamp"""

    # Generate timestamp if not provided
    if timestamp is None:
        timestamp = datetime.now().strftime("%I:%M %p")

    # Generate unique ID
    if message_id is None:
        message_id = f"msg_{len(st.session_state.messages)}"

    if is_user:
        # User message (right-aligned, blue)
        st.markdown(f'''
        <div style="display: flex; align-items: flex-start; justify-
content: flex-end; margin: 0.75rem 0;">
            <div class="user-message">
                <div>{message}</div>
                <div style="font-size: 0.75rem; opacity: 0.7; margin-top:
0.25rem;">
                    {timestamp}
                </div>
            </div>
            <span class="message-avatar user-avatar">👤</span>
        </div>
        ''', unsafe_allow_html=True)
    else:
        # AI message (left-aligned, gray)
        st.markdown(f'''
        <div style="display: flex; align-items: flex-start; margin:
0.75rem 0;">
            <span class="message-avatar bot-avatar">🤖</span>
            <div class="bot-message">
                <div>{message}</div>
                <div style="font-size: 0.75rem; opacity: 0.7; margin-top:
0.25rem;">
                    {timestamp}
                </div>
            </div>
        </div>
        ''', unsafe_allow_html=True)
```

Features:

1. **Avatars:** Visual distinction
2. **Timestamps:** When message was sent
3. **Alignment:** User right, AI left (like iMessage)
4. **Styling:** Different colors for clarity

10.8 Input Form

```
# Chat input form
with st.form("chat_form", clear_on_submit=True):
    col1, col2 = st.columns([6, 1])

    with col1:
        user_input = st.text_input(
            "Message:",
            placeholder="Ask me anything about school...",
            label_visibility="collapsed"
        )

    with col2:
        submit_button = st.form_submit_button(
            "Send",
            use_container_width=True
        )

    # Process submission
    if submit_button and user_input:
        # Add user message
        new_message = {
            "role": "user",
            "content": user_input,
            "timestamp": datetime.now().strftime("%I:%M %p")
        }
        st.session_state.messages.append(new_message)

        # Get AI response with loading indicator
        with st.spinner("🤔 Thinking..."):
            response = get_openai_response(user_input)

        # Add AI response
        ai_message = {
            "role": "assistant",
            "content": response,
            "timestamp": datetime.now().strftime("%I:%M %p")
        }
        st.session_state.messages.append(ai_message)

        # Save and refresh
        save_chat_history()
        st.rerun()
```

Why Form?

1. **Enter Key Works:** Submits on Enter press
2. **Clear on Submit:** Input box clears after sending
3. **Grouped:** All inputs submit together
4. **Better UX:** Professional feel

Two Columns:

- Column 1 (wider): Text input
- Column 2 (narrow): Send button
- Creates clean layout

Chapter 11: Implementing AI Responses

11.1 Understanding the get_openai_response Function

This is the most critical function in our application - it connects everything together.

Complete Function:

```
def get_openai_response(user_message, context="", max_retries=3):
    """
    Get response from OpenAI API with retry mechanism

    Args:
        user_message (str): The user's question
        context (str): Additional context
        max_retries (int): Number of retry attempts

    Returns:
        str: AI's response or error message
    """
    import httpx

    for attempt in range(max_retries):
        try:
            # Clear proxy environment variables
            for key in list(os.environ.keys()):
                if 'PROXY' in key.upper() or 'proxy' in key:
                    del os.environ[key]

            # Set API key
            os.environ["OPENAI_API_KEY"] = config.OPENAI_API_KEY

            # Create HTTP client without proxies
            http_client = httpx.Client()

            # Initialize OpenAI client
            client = openai.OpenAI(http_client=http_client)

            # Add language instruction
            language_instruction = ""
            if st.session_state.selected_language != "English":
                language_instruction = f"\n\nIMPORTANT: Please respond in
{st.session_state.selected_language} language."

            # Create system prompt
            system_prompt = f"""
            You are a helpful AI assistant for {config.SCHOOL_NAME}.
            You help students, parents, and staff with school-related
```

```
queries.

            School Information:
            - Name: {config.SCHOOL_NAME}
            - Address: {config.SCHOOL_ADDRESS}
            - Phone: {config.SCHOOL_PHONE}
            - Email: {config.SCHOOL_EMAIL}

            Available Information:
            - School Schedule: {json.dumps(SCHOOL_SCHEDULE, indent=2)}
            - Upcoming Events: {json.dumps(UPCOMING_EVENTS, indent=2)}
            - Homework Assignments: {json.dumps(HOMEWORK_ASSIGNMENTS,
indent=2)}
            - School Policies: {json.dumps(SCHOOL_POLICIES, indent=2)}
            - Contact Information: {json.dumps(CONTACT_INFO, indent=2)}

            Guidelines:
            1. Be friendly, helpful, and professional
            2. Provide accurate information based on the school data
            3. If you don't know something, suggest contacting the school
directly
            4. Keep responses concise but informative
            5. Use the school's name appropriately
            {language_instruction}

            Context: {context}
            """

            # Make API call
            response = client.chat.completions.create(
                model=config.MODEL,
                messages=[
                    {"role": "system", "content": system_prompt},
                    {"role": "user", "content": user_message}
                ],
                max_tokens=config.MAX_TOKENS,
                temperature=config.TEMPERATURE
            )

            return response.choices[0].message.content

        except Exception as e:
            if attempt < max_retries - 1:
                # Wait before retrying (exponential backoff)
                time.sleep(1 * (attempt + 1))
                continue
            else:
                return f"⚠️ Error after {max_retries} attempts:
{str(e)}\n\nPlease check your internet connection and API key, then try
again."

    return "⚠️ Unable to get response. Please try again later."
```

Let's Break This Down:

11.2 Retry Mechanism

```
for attempt in range(max_retries):
    try:
        # Try to get response
        ...
    except Exception as e:
        if attempt < max_retries - 1:
            time.sleep(1 * (attempt + 1))  # Wait longer each time
            continue  # Try again
        else:
            return error_message  # Give up
```

How It Works:

Attempt 1: Try immediately **Attempt 2:** Wait 1 second, try again **Attempt 3:** Wait 2 seconds, try again **Failed:** Return error message

Why Retry?

- Network hiccups are common
- Temporary server issues
- Rate limiting
- Better user experience

Exponential Backoff:

- Wait 1 second first time
- Wait 2 seconds second time
- Wait 3 seconds third time
- Gives server time to recover

11.3 Proxy Handling

```
# Clear proxy environment variables
for key in list(os.environ.keys()):
    if 'PROXY' in key.upper() or 'proxy' in key:
        del os.environ[key]

# Create HTTP client without proxies
http_client = httpx.Client()

# Pass to OpenAI client
client = openai.OpenAI(http_client=http_client)
```

Why This is Needed:

Some systems have proxy settings that interfere with OpenAI API calls.

What It Does:

1. Removes all proxy-related environment variables
2. Creates a clean HTTP client
3. Passes it to OpenAI

Without This: `Error: Client.__init__() got an unexpected keyword argument 'proxies'`

With This: Works perfectly!

11.4 Building the System Prompt

The system prompt is like training instructions for the AI.

```
system_prompt = f"""
You are a helpful AI assistant for {config.SCHOOL_NAME}.
...
"""
```

Key Components:

1. Role Definition:

```
"You are a helpful AI assistant for Smart Academy."
```

Tells AI what it is.

2. School Information:

```
f"""
- Name: {config.SCHOOL_NAME}
- Address: {config.SCHOOL_ADDRESS}
...
"""
```

Gives AI basic facts to share.

3. Data Context:

```
f"""
- School Schedule: {json.dumps(SCHOOL_SCHEDULE, indent=2)}
- Events: {json.dumps(UPCOMING_EVENTS, indent=2)}
...
"""
```

Why json.dumps()? Converts Python dictionary to readable string format.

Without json.dumps:

```
{'Math': {'time': '8:00'}} # Not readable
```

With json.dumps:

```
{
  "Math": {
    "time": "8:00"
  }
}
```

Much clearer for AI!

4. Guidelines:

```
"""
1. Be friendly, helpful, and professional
2. Provide accurate information
3. If unsure, suggest contacting school
...
"""
```

Shapes how AI responds.

5. Language Instruction:

```
if st.session_state.selected_language != "English":
    language_instruction = f"\n\nIMPORTANT: Please respond in
{st.session_state.selected_language} language."
```

Enables multi-language support!

11.5 Making the API Call

```
response = client.chat.completions.create(
    model=config.MODEL,              # Which AI to use
    messages=[                        # Conversation history
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": user_message}
    ],
    max_tokens=config.MAX_TOKENS,    # Response length limit
    temperature=config.TEMPERATURE    # Creativity level
)
```

Message Structure:

System Message:

```
{"role": "system", "content": "You are a helpful assistant..."}
```

Instructions for the AI.

User Message:

```
{"role": "user", "content": "What's my homework?"}
```

The actual question.

AI Response (returned):

```
{"role": "assistant", "content": "Your math homework is..."}
```

The answer.

11.6 Extracting the Response

```
return response.choices[0].message.content
```

Understanding the Structure:

Full Response Object:

```
{
  "choices": [
    {
      "message": {
        "role": "assistant",
        "content": "Your homework is..."
      }
    }
  ]
}
```

Navigation:

- `response.choices` → List of responses
- `[0]` → First response
- `.message` → The message object
- `.content` → The actual text

11.7 Error Handling Strategies

Specific Errors vs General:

Too Specific (Bad):

```
except openai.AuthenticationError:
    return "Bad API key"
except openai.RateLimitError:
    return "Too many requests"
# ... 20 more specific errors ...
```

Too General (Also Bad):

```
except:
    return "Something broke"
```

Balanced (Good):

```
except Exception as e:
    if attempt < max_retries - 1:
        # Retry
    else:
        return f"Error: {str(e)}"  # Show actual error
```

Why This Works:

- Catches all errors
- Shows what went wrong
- Retries automatically
- User-friendly message

11.8 Testing the Function

Manual Testing:

```
# Test basic question
response = get_openai_response("What's my homework?")
print(response)

# Test with context
response = get_openai_response(
    "When is lunch?",
    context="User is a 10th grade student"
)
print(response)
```

```
# Test error handling (bad API key)
config.OPENAI_API_KEY = "invalid"
response = get_openai_response("Hello")
print(response)  # Should show error message
```

Chapter 12: Managing Conversation Flow

12.1 Understanding Conversation State

The Challenge: Chatbots need to remember previous messages to have coherent conversations.

Example:

Without Memory:

```
User: What's the math homework?
AI: Complete exercises 1–20 on page 45.

User: When is it due?
AI: I don't know what you're referring to. 🤷
```

With Memory:

```
User: What's the math homework?
AI: Complete exercises 1–20 on page 45.

User: When is it due?
AI: The math homework is due on October 25th. ✅
```

12.2 Storing Messages in Session State

Message Structure:

```
{
    "role": "user",  # or "assistant"
    "content": "What's my homework?",
    "timestamp": "2:30 PM"
}
```

Initializing:

```
if "messages" not in st.session_state:
    st.session_state.messages = []
```

Adding Messages:

```
# User message
st.session_state.messages.append({
    "role": "user",
    "content": user_input,
    "timestamp": datetime.now().strftime("%I:%M %p")
})

# AI response
st.session_state.messages.append({
    "role": "assistant",
    "content": ai_response,
    "timestamp": datetime.now().strftime("%I:%M %p")
})
```

12.3 Building Conversation Context

Including History in API Calls:

```
def get_openai_response_with_history(user_message):
    """Get AI response with full conversation history"""

    # Build message list
    messages = [
        {"role": "system", "content": system_prompt}
    ]

    # Add conversation history
    for msg in st.session_state.messages:
        messages.append({
            "role": msg["role"],
            "content": msg["content"]
        })

    # Add new message
    messages.append({
        "role": "user",
        "content": user_message
    })

    # Get response
    response = client.chat.completions.create(
        model=config.MODEL,
        messages=messages
    )

    return response.choices[0].message.content
```

How This Works:

First Question:

```
messages = [
    {"role": "system", "content": "You are..."},
    {"role": "user", "content": "What's my homework?"}
]
```

Second Question (with context):

```
messages = [
    {"role": "system", "content": "You are..."},
    {"role": "user", "content": "What's my homework?"},
    {"role": "assistant", "content": "Math: Page 45..."},
    {"role": "user", "content": "When is it due?"}
]
```

AI can now reference previous messages!

12.4 Managing Context Length

Problem: Too much history → Too many tokens → Expensive & Slow

Solution Options:

1. Limit Number of Messages:

```
# Only include last 20 messages
recent_messages = st.session_state.messages[-20:]

for msg in recent_messages:
    messages.append({
        "role": msg["role"],
        "content": msg["content"]
    })
```

2. Summarize Old Conversations:

```
if len(st.session_state.messages) > 50:
    # Summarize older messages
    summary = summarize_conversation(st.session_state.messages[:-20])

    messages = [
        {"role": "system", "content": system_prompt},
        {"role": "system", "content": f"Previous conversation summary:
{summary}"},
        # Add recent messages
```

```
        ...
    ]
```

3. Remove Old Messages:

```
# Keep only last 100 messages
if len(st.session_state.messages) > 100:
    st.session_state.messages = st.session_state.messages[-100:]
```

12.5 Conversation Branching

Handling Different Topics:

```
def detect_topic_change(current_message, previous_messages):
    """Detect if user changed topic"""

    topics = {
        "homework": ["homework", "assignment", "due"],
        "schedule": ["schedule", "class", "period"],
        "events": ["event", "fair", "conference"]
    }

    # Simple keyword matching
    current_topic = None
    for topic, keywords in topics.items():
        if any(kw in current_message.lower() for kw in keywords):
            current_topic = topic
            break

    # Compare with previous topic
    # ... logic to detect change

    return topic_changed
```

Why Detect Topic Changes?

- Can clear irrelevant context
- Provide more focused responses
- Better user experience

12.6 Conversation Persistence

Saving Entire Conversations:

```
import pickle
from datetime import datetime

def save_conversation():
```

```
    """Save conversation with metadata"""

    conversation_data = {
        "messages": st.session_state.messages,
        "start_time": st.session_state.get("start_time", datetime.now()),
        "last_updated": datetime.now(),
        "language": st.session_state.selected_language
    }

    filename =
f"conversation_{datetime.now().strftime('%Y%m%d_%H%M%S')}.pkl"

    with open(filename, "wb") as f:
        pickle.dump(conversation_data, f)

    return filename
```

Loading Previous Conversations:

```
def load_conversation(filename):
    """Load a saved conversation"""

    try:
        with open(filename, "rb") as f:
            data = pickle.load(f)

        st.session_state.messages = data["messages"]
        st.session_state.selected_language = data.get("language",
"English")

        return True
    except Exception as e:
        st.error(f"Could not load conversation: {e}")
        return False
```

12.7 Creating Transcript Downloads

Generate Text Transcript:

```
def create_transcript():
    """Create downloadable text transcript"""

    transcript = f"""
SCHOOL CHATBOT CONVERSATION TRANSCRIPT
Generated: {datetime.now().strftime('%Y-%m-%d %I:%M %p')}
School: {config.SCHOOL_NAME}
Language: {st.session_state.selected_language}
{'=' * 60}

"""
```

```
    for msg in st.session_state.messages:
        role = "YOU" if msg["role"] == "user" else "ASSISTANT"
        timestamp = msg.get("timestamp", "")

        transcript += f"\n[{timestamp}] {role}:\n"
        transcript += f"{msg['content']}\n"
        transcript += "-" * 60 + "\n"

    transcript += f"\n\nEnd of Transcript\n"
    transcript += f"Total Messages: {len(st.session_state.messages)}\n"

    return transcript
```

Using in Streamlit:

```
if st.session_state.messages:
    transcript = create_transcript()

    st.download_button(
        label="📥 Download Transcript",
        data=transcript,

file_name=f"chat_transcript_{datetime.now().strftime('%Y%m%d')}.txt",
        mime="text/plain"
    )
```

Chapter 13: Adding Multi-Language Support

13.1 How Multi-Language Works

Simple Approach: Tell the AI to respond in a different language!

Implementation:

```
# Language selector
st.session_state.selected_language = st.selectbox(
    "🌐 Language:",
    ["English", "Spanish", "French", "Hindi", "Chinese", "Arabic"]
)

# Add to system prompt
language_instruction = ""
if st.session_state.selected_language != "English":
    language_instruction = f"\n\nIMPORTANT: Please respond in
{st.session_state.selected_language} language."

system_prompt = f"""
You are a helpful assistant...
```

```
{language_instruction}
"""
```

That's It! The AI handles translation automatically.

13.2 Language-Specific Data

For Better Results, Translate Data:

```
TRANSLATIONS = {
    "Spanish": {
        "greeting": "Hola",
        "homework": "Tarea",
        "schedule": "Horario"
    },
    "French": {
        "greeting": "Bonjour",
        "homework": "Devoirs",
        "schedule": "Emploi du temps"
    }
}

# Use translations
if language != "English":
    translated_data = translate_school_data(SCHOOL_SCHEDULE, language)
else:
    translated_data = SCHOOL_SCHEDULE
```

13.3 Testing Multi-Language

Quick Test:

```
# Select Spanish
st.session_state.selected_language = "Spanish"

# Ask question
response = get_openai_response("What's my homework?")

# Should respond in Spanish:
# "Tu tarea de matemáticas es..."
```

Supported Languages: GPT-4o-mini supports 50+ languages including:

- English
- Spanish (Español)
- French (Français)
- German (Deutsch)
- Italian (Italiano)

- Portuguese (Português)
- Russian (Русский)
- Japanese ( )
- Korean (한국어)
- Chinese ( )
- Arabic (العربية)
- Hindi (हिन्दी)

Chapter 14: Implementing Search & Filters

14.1 Basic Search Functionality

Simple Keyword Search:

```
def search_messages(query):
    """Search through chat history"""

    if not query:
        return st.session_state.messages

    query_lower = query.lower()

    results = []
    for msg in st.session_state.messages:
        if query_lower in msg["content"].lower():
            results.append(msg)

    return results
```

Using in UI:

```
# Search box
search_query = st.text_input("🔍 Search chats:", "")

# Filter messages
if search_query:
    filtered_messages = search_messages(search_query)
    st.info(f"Found {len(filtered_messages)} messages matching
'{search_query}'")
else:
    filtered_messages = st.session_state.messages

# Display filtered messages
for msg in filtered_messages:
    display_chat_message(msg["content"], msg["role"] == "user")
```

14.2 Advanced Search Features

Search by Date:

```
def search_by_date(target_date):
    """Find messages from specific date"""

    results = []
    for msg in st.session_state.messages:
        # Assuming timestamp format: "10/22/2024 2:30 PM"
        msg_date = msg.get("date", "")
        if target_date in msg_date:
            results.append(msg)

    return results
```

Search by Topic:

```
def search_by_topic(topic):
    """Find messages about specific topic"""

    topic_keywords = {
        "homework": ["homework", "assignment", "due", "page"],
        "schedule": ["schedule", "class", "period", "time"],
        "events": ["event", "fair", "conference", "meeting"]
    }

    keywords = topic_keywords.get(topic, [topic])

    results = []
    for msg in st.session_state.messages:
        content_lower = msg["content"].lower()
        if any(kw in content_lower for kw in keywords):
            results.append(msg)

    return results
```

Using Topic Search:

```
topic = st.selectbox(
    "Search by topic:",
    ["All", "Homework", "Schedule", "Events"]
)

if topic != "All":
    filtered = search_by_topic(topic.lower())
else:
    filtered = st.session_state.messages
```

14.3 Highlighting Search Results

Show Matching Text:

```
def highlight_text(text, query):
    """Highlight matching query in text"""

    if not query:
        return text

    # Simple replacement (case-insensitive)
    import re
    pattern = re.compile(re.escape(query), re.IGNORECASE)

    highlighted = pattern.sub(
        lambda m: f'<mark style="background-color: yellow;">{m.group()}
</mark>',
        text
    )

    return highlighted
```

Display with Highlights:

```
for msg in search_results:
    highlighted_content = highlight_text(msg["content"], search_query)

    st.markdown(f'''
    <div class="search-result">
        {highlighted_content}
    </div>
    ''', unsafe_allow_html=True)
```

Chapter 15: Error Handling & Reliability

15.1 Types of Errors

1. API Errors:

- Invalid API key
- Rate limiting
- Network issues
- Server errors

2. User Input Errors:

- Empty messages
- Too long messages
- Invalid characters

3. System Errors:

- File not found
- Permission denied
- Out of memory

15.2 Graceful Error Handling

API Key Validation:

```
def validate_api_key():
    """Check if API key is valid"""

    if not config.OPENAI_API_KEY:
        st.error("⚠️ No API key found! Please add your OpenAI API key to
the .env file.")
        st.code("OPENAI_API_KEY=your-key-here")
        st.stop()  # Stop execution

    if not config.OPENAI_API_KEY.startswith("sk-"):
        st.warning("⚠️ API key format looks incorrect. OpenAI keys start
with 'sk-'")
```

Input Validation:

```
def validate_user_input(text):
    """Validate user input before processing"""

    # Check if empty
    if not text or text.strip() == "":
        return False, "Please enter a message"

    # Check length
    if len(text) > 1000:
        return False, "Message too long (max 1000 characters)"

    # Check for invalid characters
    if any(char in text for char in ['<', '>', '{', '}']):
        return False, "Invalid characters detected"

    return True, ""

# Use it
user_input = st.text_input("Ask:")

if user_input:
    is_valid, error_msg = validate_user_input(user_input)

    if not is_valid:
        st.error(error_msg)
    else:
```

```
        # Process input
        ...
```

15.3 User-Friendly Error Messages

❌ **Bad Error Messages:**

```
"Error: NoneType object has no attribute 'choices'"
"Exception in thread main: IndexError at line 42"
```

✅ **Good Error Messages:**

```
"⚠️ Couldn't get a response. Please check your internet connection and try
again."

"⚠️ The message is too long. Please try a shorter question (max 500
characters)."

"⚠️ There was a problem connecting to the AI service. Please wait a moment
and try again."
```

Implementation:

```
try:
    response = get_openai_response(user_input)
except openai.AuthenticationError:
    st.error("⚠️ API key is invalid. Please check your configuration.")
    st.info("💡 Tip: Make sure your API key starts with 'sk-' and has no
extra spaces.")
except openai.RateLimitError:
    st.warning("⚠️ You're sending requests too quickly. Please wait a
moment.")
    st.info("💡 Tip: Rate limits reset every minute.")
except openai.APIConnectionError:
    st.error("⚠️ Couldn't connect to OpenAI. Please check your internet
connection.")
except Exception as e:
    st.error(f"⚠️ An unexpected error occurred: {str(e)}")
    st.info("💡 Tip: Try refreshing the page or restarting the
application.")
```

15.4 Logging Errors

Simple Logging:

```
import logging
from datetime import datetime

# Set up logging
logging.basicConfig(
    filename='chatbot_errors.log',
    level=logging.ERROR,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

# Log errors
try:
    response = get_openai_response(user_input)
except Exception as e:
    logging.error(f"Error processing request: {str(e)}")
    logging.error(f"User input: {user_input}")
    logging.error(f"Stack trace:", exc_info=True)
```

Log File Output:

```
2024-10-22 14:30:15 - ERROR - Error processing request: Connection timeout
2024-10-22 14:30:15 - ERROR - User input: What's my homework?
2024-10-22 14:30:15 - ERROR - Stack trace: ...
```

15.5 Fallback Responses

When AI Fails:

```
FALLBACK_RESPONSES = {
    "homework": "I couldn't get an AI response, but you can check homework
assignments in the school portal at portal.smartacademy.edu",

    "schedule": "I'm having trouble right now, but your schedule is
available in the student handbook or by calling the office at (555) 123-
4567",

    "default": "I'm having trouble connecting right now. Please try again
in a moment, or contact the school office at (555) 123-4567 for immediate
assistance."
}

def get_fallback_response(user_input):
    """Provide fallback when AI is unavailable"""

    user_input_lower = user_input.lower()

    for key, response in FALLBACK_RESPONSES.items():
        if key in user_input_lower:
            return f"⚠️ {response}"
```

```
    return f"⚠️ {FALLBACK_RESPONSES['default']}"

# Use it
try:
    response = get_openai_response(user_input)
except Exception:
    response = get_fallback_response(user_input)
```

Chapter 16: State Management

16.1 Understanding Session State

What is Session State? Streamlit's way of persisting data across script reruns.

Without Session State:

```
count = 0  # Resets to 0 every time!

if st.button("Add"):
    count += 1

st.write(count)  # Always shows 0 😞
```

With Session State:

```
if "count" not in st.session_state:
    st.session_state.count = 0

if st.button("Add"):
    st.session_state.count += 1

st.write(st.session_state.count)  # Persists! 😊
```

16.2 Session State Patterns

Pattern 1: Initialization

```
# Initialize all session state variables
if "messages" not in st.session_state:
    st.session_state.messages = []

if "user_name" not in st.session_state:
    st.session_state.user_name = ""
```

```
if "selected_language" not in st.session_state:
    st.session_state.selected_language = "English"
```

Pattern 2: Callbacks

```
def on_language_change():
    """Called when language changes"""
    st.success(f"Language changed to
{st.session_state.selected_language}")

language = st.selectbox(
    "Language:",
    ["English", "Spanish", "French"],
    key="selected_language",
    on_change=on_language_change
)
```

Pattern 3: Forms with State

```
with st.form("settings"):
    name = st.text_input("Name:", value=st.session_state.get("user_name",
""))
    submit = st.form_submit_button("Save")

    if submit:
        st.session_state.user_name = name
        st.success("Settings saved!")
```

16.3 Our Application's State

All Session State Variables:

```
# Messages
st.session_state.messages = []  # List of all chat messages

# Settings
st.session_state.selected_language = "English"  # Current language

# Temporary state
st.session_state.quick_action = ""  # Quick action button clicked

# User preferences (if we add them)
st.session_state.user_name = ""
st.session_state.grade_level = ""
st.session_state.dark_mode = False
```

16.4 Resetting State

Clear All State:

```
def reset_app():
    """Reset application to initial state"""

    # Clear specific items
    st.session_state.messages = []
    st.session_state.selected_language = "English"

    # Or clear everything
    for key in list(st.session_state.keys()):
        del st.session_state[key]

    st.success("Application reset!")
    st.rerun()

if st.button("Reset App"):
    reset_app()
```

PART 3: PROFESSIONAL DEVELOPMENT

Chapter 17: Code Organization & Best Practices

17.1 Writing Clean Code

Principles of Clean Code:

1. Meaningful Names:

```
# ❌ Bad
def fn(x, y):
    return x + y

# ✅ Good
def calculate_total_score(homework_score, test_score):
    return homework_score + test_score
```

2. Keep Functions Small:

```
# ❌ Bad - does too much
def process_user_request(input):
    validate_input(input)
    get_ai_response(input)
    save_to_database(input)
    send_email_notification(input)
    update_analytics(input)
```

```
    # 100 more lines...

# ✅ Good - single responsibility
def process_user_request(input):
    if not is_valid_input(input):
        return error_response()

    response = get_ai_response(input)
    save_message(input, response)

    return response
```

3. Use Comments Wisely:

```
# ❌ Bad - obvious comment
# Increment count by 1
count = count + 1

# ✅ Good - explains WHY
# Retry with exponential backoff to avoid overwhelming the API
time.sleep(2 ** attempt)
```

4. Consistent Formatting:

```
# ✅ Good - consistent style
def get_openai_response(user_message, context="", max_retries=3):
    """Get AI response with retry mechanism"""

    for attempt in range(max_retries):
        try:
            response = client.chat.completions.create(
                model=config.MODEL,
                messages=build_messages(user_message, context)
            )
            return response.choices[0].message.content
        except Exception as e:
            if attempt < max_retries - 1:
                time.sleep(1 * (attempt + 1))
                continue
            else:
                return handle_error(e)
```

17.2 Documentation Standards

Function Documentation (Docstrings):

```
def get_openai_response(user_message, context="", max_retries=3):
    """
```

```
    Get response from OpenAI API with automatic retry on failure.

    This function sends a user's message to the OpenAI API along with
    school context and returns the AI's response. It implements
exponential
    backoff retry logic to handle temporary failures.

    Args:
        user_message (str): The question or message from the user
        context (str, optional): Additional context to provide to the AI.
            Defaults to empty string.
        max_retries (int, optional): Maximum number of retry attempts.
            Defaults to 3.

    Returns:
        str: The AI's response text, or an error message if all retries
fail

    Raises:
        None: All exceptions are caught and returned as error messages

    Example:
        >>> response = get_openai_response("What's my homework?")
        >>> print(response)
        "Your math homework is to complete exercises 1-20..."

    Note:
        This function clears proxy environment variables to avoid
        compatibility issues with some system configurations.
    """
    # Implementation...
```

Module Documentation:

```
"""
School Chatbot Application - Main Module

This module contains the core functionality for the AI-powered school
chatbot, including the user interface, chat message handling, and
integration with the OpenAI API.

The application uses Streamlit for the web interface and provides
features including:
- Real-time AI-powered responses to school-related questions
- Multi-language support
- Chat history persistence
- Search and filtering capabilities
- Download of conversation transcripts

Author: Your Name
Created: October 2025
Version: 1.0
"""
```

```
import streamlit as st
import openai
# ... rest of imports
```

17.3 Code Organization Patterns

Separate Concerns:

Before (Everything Mixed):

```
# app.py - 2000 lines
API_KEY = "sk-..."
HOMEWORK = {...}

def main():
    # UI code
    # API code
    # Data code
    # Everything mixed together
```

After (Well Organized):

```
project/
├── config.py          # Configuration only
├── school_data.py     # Data only
├── api.py             # API interactions only
├── ui.py              # UI components only
└── app.py             # Main application (glues everything)
```

Example api.py:

```
"""
API Module - Handles all OpenAI API interactions
"""

import openai
import time
from config import OPENAI_API_KEY, MODEL, MAX_TOKENS

class ChatbotAPI:
    """Handles OpenAI API communication"""

    def __init__(self):
        """Initialize the API client"""
        self.client = openai.OpenAI(api_key=OPENAI_API_KEY)

    def get_response(self, message, context="", max_retries=3):
```

```
        """Get AI response with retry logic"""
        # Implementation...

    def build_system_prompt(self, school_data):
        """Build the system prompt with school information"""
        # Implementation...
```

Example ui.py:

```
"""
UI Module - Streamlit UI components
"""

import streamlit as st

def display_header():
    """Display application header"""
    st.markdown("""
    <div class="main-header">
        <h1>🎓 School Chatbot</h1>
    </div>
    """, unsafe_allow_html=True)

def display_chat_message(message, is_user=True):
    """Display a single chat message"""
    # Implementation...

def create_sidebar():
    """Create and populate the sidebar"""
    # Implementation...
```

17.4 Constants and Configuration

Use Constants:

```
# ❌ Bad - magic numbers
if len(message) > 1000:
    return error

time.sleep(2 ** attempt)

# ✅ Good - named constants
MAX_MESSAGE_LENGTH = 1000
BASE_RETRY_DELAY = 2

if len(message) > MAX_MESSAGE_LENGTH:
    return error

time.sleep(BASE_RETRY_DELAY ** attempt)
```

Configuration File:

```
# config.py

# API Configuration
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY", "")
MODEL = "gpt-4o-mini"
MAX_TOKENS = 1000
TEMPERATURE = 0.7

# Application Settings
MAX_MESSAGE_LENGTH = 1000
MAX_RETRY_ATTEMPTS = 3
BASE_RETRY_DELAY = 2
MAX_HISTORY_MESSAGES = 100

# UI Configuration
CHAT_BUBBLE_MAX_WIDTH = "70%"
TIMESTAMP_FORMAT = "%I:%M %p"
DATE_FORMAT = "%Y-%m-%d"

# File Paths
CHAT_HISTORY_FILE = "chat_history.pkl"
ERROR_LOG_FILE = "errors.log"
```

17.5 Error Handling Best Practices

Specific Error Handling:

```
def process_user_message(message):
    """Process user message with comprehensive error handling"""

    # Validate input
    try:
        validated_message = validate_message(message)
    except ValueError as e:
        return f"Invalid input: {e}"

    # Get AI response
    try:
        response = get_ai_response(validated_message)
    except openai.AuthenticationError:
        return "Authentication failed. Please check API key."
    except openai.RateLimitError:
        return "Rate limit exceeded. Please wait a moment."
    except openai.APIConnectionError:
        return "Connection failed. Please check your internet."
    except Exception as e:
        logging.error(f"Unexpected error: {e}", exc_info=True)
        return "An unexpected error occurred. Please try again."
```

```
    # Save to history
    try:
        save_to_history(message, response)
    except IOError as e:
        logging.warning(f"Could not save to history: {e}")
        # Continue anyway - not critical

    return response
```

Chapter 18: Security & Privacy

18.1 Protecting API Keys

❌ **NEVER Do This:**

```
# Hardcoded in code
API_KEY = "sk-proj-abc123xyz..."

# In Git repository
config.py with API key committed

# Shared in public places
GitHub, Discord, emails with API key visible
```

✅ **Always Do This:**

1. Use Environment Variables:

```
# .env file (NEVER commit this)
OPENAI_API_KEY=sk-proj-your-key-here

# In code
from dotenv import load_dotenv
import os

load_dotenv()
API_KEY = os.getenv("OPENAI_API_KEY")
```

2. Add to .gitignore:

```
# .gitignore
.env
*.env
config_local.py
secrets/
```

3. Provide Template:

```
# .env.example (safe to commit)
OPENAI_API_KEY=your-api-key-here
SCHOOL_NAME=Your School Name
```

18.2 Input Sanitization

Prevent Injection Attacks:

```
def sanitize_input(user_input):
    """Clean user input to prevent injection attacks"""

    # Remove potentially dangerous characters
    dangerous_chars = ['<', '>', '{', '}', '\\', '`']
    cleaned = user_input

    for char in dangerous_chars:
        cleaned = cleaned.replace(char, '')

    # Limit length
    MAX_LENGTH = 1000
    cleaned = cleaned[:MAX_LENGTH]

    # Strip whitespace
    cleaned = cleaned.strip()

    return cleaned

# Use it
user_input = st.text_input("Ask a question:")
safe_input = sanitize_input(user_input)
```

Validate File Uploads (if you add this feature):

```
def validate_file_upload(uploaded_file):
    """Validate uploaded file is safe"""

    # Check file type
    ALLOWED_TYPES = ['text/plain', 'application/pdf']
    if uploaded_file.type not in ALLOWED_TYPES:
        raise ValueError("File type not allowed")

    # Check file size
    MAX_SIZE = 5 * 1024 * 1024  # 5 MB
    if uploaded_file.size > MAX_SIZE:
        raise ValueError("File too large")
```

```
    return True
```

18.3 Data Privacy

Protecting User Data:

1. Don't Log Sensitive Information:

```
# ❌ Bad
logging.info(f"User {student_name} asked: {question}")

# ✅ Good
logging.info(f"User query processed successfully")
```

2. Anonymize Data:

```
def anonymize_conversation(messages):
    """Remove personally identifiable information"""

    anonymized = []

    for msg in messages:
        content = msg["content"]

        # Remove names (simple approach)
        content = re.sub(r'\b[A-Z][a-z]+ [A-Z][a-z]+\b', '[NAME]',
content)

        # Remove email addresses
        content = re.sub(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]
{2,}\b',
                         '[EMAIL]', content)

        # Remove phone numbers
        content = re.sub(r'\b\d{3}[-.]?\d{3}[-.]?\d{4}\b', '[PHONE]',
content)

        anonymized.append({
            "role": msg["role"],
            "content": content,
            "timestamp": msg["timestamp"]
        })

    return anonymized
```

3. User Consent:

```
# Show privacy notice on first use
if "privacy_accepted" not in st.session_state:
    st.warning("""
    **Privacy Notice:**
    This chatbot stores conversation history locally on your device.
    Your conversations are sent to OpenAI for processing.
    We do not share your data with third parties.
    """)

    if st.button("I Understand"):
        st.session_state.privacy_accepted = True
        st.rerun()

    st.stop()  # Don't show app until accepted
```

18.4 Rate Limiting

Prevent Abuse:

```
import time
from collections import defaultdict

# Track requests per user/session
request_times = defaultdict(list)

def check_rate_limit(session_id, max_requests=10, window=60):
    """
    Check if user is within rate limits

    Args:
        session_id: Unique session identifier
        max_requests: Max requests allowed
        window: Time window in seconds

    Returns:
        tuple: (allowed: bool, wait_time: int)
    """
    now = time.time()

    # Get requests in time window
    requests = request_times[session_id]
    recent_requests = [t for t in requests if now - t < window]

    if len(recent_requests) >= max_requests:
        # Calculate wait time
        oldest_request = min(recent_requests)
        wait_time = int(window - (now - oldest_request))
        return False, wait_time

    # Add current request
    recent_requests.append(now)
```

```
    request_times[session_id] = recent_requests

    return True, 0

# Use it
session_id = st.session_state.get("session_id", "default")

allowed, wait_time = check_rate_limit(session_id)

if not allowed:
    st.warning(f"⚠️ Too many requests. Please wait {wait_time} seconds.")
    st.stop()
```

18.5 Secure Configuration

Environment-Specific Settings:

```
# config.py

import os

# Determine environment
ENVIRONMENT = os.getenv("ENVIRONMENT", "development")

# Base configuration
class Config:
    OPENAI_API_KEY = os.getenv("OPENAI_API_KEY", "")
    SCHOOL_NAME = os.getenv("SCHOOL_NAME", "Smart Academy")

    # Security settings
    MAX_MESSAGE_LENGTH = 1000
    RATE_LIMIT_REQUESTS = 10
    RATE_LIMIT_WINDOW = 60

# Development configuration
class DevelopmentConfig(Config):
    DEBUG = True
    LOG_LEVEL = "DEBUG"

# Production configuration
class ProductionConfig(Config):
    DEBUG = False
    LOG_LEVEL = "ERROR"
    REQUIRE_HTTPS = True

# Select configuration
if ENVIRONMENT == "production":
    config = ProductionConfig()
else:
    config = DevelopmentConfig()
```

Chapter 19: Testing & Debugging

19.1 Manual Testing

Test Checklist:

✅ **Basic Functionality:**

```
□ App starts without errors
□ Can type and send messages
□ AI responds correctly
□ Messages display properly
□ Timestamps show
```

✅ **Features:**

```
□ Quick actions work
□ Language selector changes responses
□ Search finds messages
□ Clear chat removes messages
□ Download transcript creates file
```

✅ **Error Handling:**

```
□ Empty message shows error
□ No API key shows helpful message
□ Network error shows retry option
□ Long message shows error
```

✅ **Edge Cases:**

```
□ Very long conversation (100+ messages)
□ Special characters in messages
□ Rapid clicking (rate limiting)
□ Browser refresh preserves state
```

19.2 Debugging Techniques

Print Debugging:

```python
# Add debug prints
def get_openai_response(user_message):
    print(f"DEBUG: Received message: {user_message}")
    print(f"DEBUG: API Key present: {bool(config.OPENAI_API_KEY)}")
```

```
    try:
        response = client.chat.completions.create(...)
        print(f"DEBUG: Got response: {response}")
        return response.choices[0].message.content
    except Exception as e:
        print(f"DEBUG: Error occurred: {e}")
        print(f"DEBUG: Error type: {type(e)}")
        raise
```

Streamlit Debugging:

```
# Show session state for debugging
if config.DEBUG:
    with st.expander("🐛 Debug Info"):
        st.write("Session State:")
        st.json(dict(st.session_state))

        st.write("Messages:")
        st.json(st.session_state.messages)
```

Logging:

```
import logging

# Configure logging
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('chatbot_debug.log'),
        logging.StreamHandler()
    ]
)

logger = logging.getLogger(__name__)

# Use throughout code
logger.debug(f"Processing message: {user_message}")
logger.info(f"Successfully got AI response")
logger.warning(f"Rate limit approaching: {request_count}/10")
logger.error(f"API call failed: {error}")
```

19.3 Common Issues & Solutions

Issue 1: "API key not found"

Solution:

```
# Check .env file exists
import os
print("Current directory:", os.getcwd())
print(".env exists:", os.path.exists(".env"))

# Check .env content
with open(".env") as f:
    print(f.read())

# Load and verify
from dotenv import load_dotenv
load_dotenv()
print("API Key:", os.getenv("OPENAI_API_KEY")[:10] + "...")
```

Issue 2: "Messages not persisting"

Solution:

```
# Verify session state initialization
if "messages" not in st.session_state:
    st.session_state.messages = load_chat_history()
    print(f"Loaded {len(st.session_state.messages)} messages")

# Verify saving
def save_chat_history():
    print(f"Saving {len(st.session_state.messages)} messages")
    with open(CHAT_HISTORY_FILE, "wb") as f:
        pickle.dump(st.session_state.messages, f)
    print("Save successful")
```

Issue 3: "Form not clearing"

Solution:

```
# Ensure clear_on_submit=True
with st.form("chat_form", clear_on_submit=True):  # ← IMPORTANT!
    user_input = st.text_input("Message:")
    submit = st.form_submit_button("Send")
```

Issue 4: "Auto-scroll not working"

Solution:

```
# Use JavaScript injection
st.markdown("""
<script>
    setTimeout(function() {
```

```
        var element = window.parent.document.querySelector('[data-
testid="stVerticalBlock"]');
        if (element) {
            element.scrollTop = element.scrollHeight;
        }
    }, 100);
</script>
""", unsafe_allow_html=True)
```

19.4 Performance Optimization

Caching Expensive Operations:

```
@st.cache_data
def load_school_data():
    """Cache school data so it only loads once"""
    return {
        "schedule": SCHOOL_SCHEDULE,
        "homework": HOMEWORK_ASSIGNMENTS,
        "events": UPCOMING_EVENTS
    }

# Use cached data
school_data = load_school_data()
```

Limit History Size:

```
# Keep only recent messages
MAX_HISTORY = 100

if len(st.session_state.messages) > MAX_HISTORY:
    # Keep only last MAX_HISTORY messages
    st.session_state.messages = st.session_state.messages[-MAX_HISTORY:]
    save_chat_history()
```

Optimize API Calls:

```
# Don't include entire history in every request
def build_messages_for_api(user_message, max_history=10):
    """Build message list with limited history"""

    messages = [{"role": "system", "content": system_prompt}]

    # Add only recent messages
    recent = st.session_state.messages[-max_history:]
    for msg in recent:
        messages.append({
            "role": msg["role"],
```

```
            "content": msg["content"]
        })

    # Add current message
    messages.append({"role": "user", "content": user_message})

    return messages
```

Chapter 20: Deployment & Maintenance

20.1 Preparing for Deployment

Pre-Deployment Checklist:

```
□ Remove all debug code
□ Remove API keys from code
□ Update .gitignore
□ Create requirements.txt
□ Write README.md
□ Add LICENSE file
□ Test in clean environment
□ Check error handling
```

Clean requirements.txt:

```
# Generate from your environment
pip freeze > requirements.txt

# Or manually create
streamlit==1.28.1
openai==1.3.0
python-dotenv==1.0.0
httpx
```

20.2 Deployment Options

Option 1: Streamlit Community Cloud (FREE!)

Steps:

1. Push code to GitHub
2. Go to share.streamlit.io
3. Connect GitHub account
4. Select your repository
5. Add secrets (API keys)
6. Deploy!

Secrets Configuration:

```
# In Streamlit Cloud dashboard, add:
OPENAI_API_KEY = "sk-your-key-here"
SCHOOL_NAME = "Smart Academy"
```

Option 2: Local Deployment

```
# Run on local network
streamlit run app.py --server.address 0.0.0.0 --server.port 8501

# Access from other devices
# http://your-ip-address:8501
```

Option 3: Heroku

```
# Install Heroku CLI
# Create Procfile
echo "web: streamlit run app.py --server.port $PORT" > Procfile

# Create runtime.txt
echo "python-3.11.0" > runtime.txt

# Deploy
heroku create my-school-chatbot
git push heroku main
```

20.3 Monitoring & Maintenance

Track Usage:

```
import json
from datetime import datetime

def log_usage(user_message, response):
    """Log usage statistics"""

    usage_log = {
        "timestamp": datetime.now().isoformat(),
        "message_length": len(user_message),
        "response_length": len(response),
        "language": st.session_state.selected_language
    }

    with open("usage_log.json", "a") as f:
        f.write(json.dumps(usage_log) + "\n")
```

Monitor Errors:

```
def log_error(error_type, error_message, context=None):
    """Log errors for monitoring"""

    error_log = {
        "timestamp": datetime.now().isoformat(),
        "type": error_type,
        "message": str(error_message),
        "context": context
    }

    with open("error_log.json", "a") as f:
        f.write(json.dumps(error_log) + "\n")
```

Analytics Dashboard:

```
def show_analytics():
    """Show usage analytics"""

    with open("usage_log.json") as f:
        logs = [json.loads(line) for line in f]

    st.metric("Total Queries", len(logs))
    st.metric("Today's Queries",
              len([l for l in logs if
l["timestamp"].startswith(datetime.now().strftime("%Y-%m-%d"))]))

    # Language breakdown
    languages = {}
    for log in logs:
        lang = log.get("language", "English")
        languages[lang] = languages.get(lang, 0) + 1

    st.bar_chart(languages)
```

20.4 Updating & Versioning

Semantic Versioning:

```
# version.py
VERSION = "1.0.0"
# 1 = Major (breaking changes)
# 0 = Minor (new features)
# 0 = Patch (bug fixes)

# In app
st.sidebar.text(f"Version {VERSION}")
```

Update Process:

1. **Make changes in development**
2. **Test thoroughly**
3. **Update version number**
4. **Commit to Git:**

```
git add .
git commit -m "v1.1.0: Add multi-language support"
git tag v1.1.0
git push origin main --tags
```

5. **Deploy to production**

20.5 Backup & Recovery

Backup Chat History:

```
import shutil
from datetime import datetime

def backup_chat_history():
    """Create backup of chat history"""

    if os.path.exists(CHAT_HISTORY_FILE):
        backup_name =
f"chat_history_backup_{datetime.now().strftime('%Y%m%d_%H%M%S')}.pkl"
        shutil.copy(CHAT_HISTORY_FILE, backup_name)
        return backup_name
    return None

# Auto-backup daily
if st.button("Create Backup"):
    backup_file = backup_chat_history()
    st.success(f"Backup created: {backup_file}")
```

Recovery:

```
def restore_from_backup(backup_file):
    """Restore chat history from backup"""

    try:
        shutil.copy(backup_file, CHAT_HISTORY_FILE)
        st.session_state.messages = load_chat_history()
        st.success("Restored successfully!")
        st.rerun()
    except Exception as e:
        st.error(f"Restore failed: {e}")
```

PART 4: EXERCISES & ASSESSMENTS

Hands-On Exercises

Exercise 1: Basic Modifications (Beginner)

Task: Customize the chatbot for your school

Steps:

1. Open `config.py`
2. Change `SCHOOL_NAME` to your school's name
3. Update contact information
4. Change the gradient colors in `app.py`

Challenge:

- Add your school logo
- Change the emoji icons
- Modify the welcome message

Exercise 2: Add New Features (Intermediate)

Task: Add a "Contact Teacher" quick action

Steps:

1. Add teacher contact info to `school_data.py`:

```
TEACHERS = {
    "Math": {
        "name": "Mr. Smith",
        "email": "smith@school.edu"
    },
    # Add more...
}
```

2. Add quick action button:

```
if st.button("📧 Contact Teacher"):
    st.session_state.quick_action = "How do I contact my teacher?"
```

3. Update system prompt to include teacher data

Challenge:

- Add office hours for each teacher

- Create a teacher directory page

Exercise 3: Implement Analytics (Advanced)

Task: Track popular questions

Steps:

1. Create analytics function:

```
def track_question_topics(user_message):
    topics = {
        "homework": ["homework", "assignment"],
        "schedule": ["schedule", "class"],
        "events": ["event", "fair"]
    }

    for topic, keywords in topics.items():
        if any(kw in user_message.lower() for kw in keywords):
            # Increment counter
            if topic not in st.session_state.topic_counts:
                st.session_state.topic_counts[topic] = 0
            st.session_state.topic_counts[topic] += 1
```

2. Display analytics in sidebar:

```
st.sidebar.subheader("Popular Topics")
st.sidebar.bar_chart(st.session_state.topic_counts)
```

Challenge:

- Add time-based analytics
- Export analytics to CSV
- Create charts with matplotlib

Exercise 4: Build a Plugin System (Expert)

Task: Create a modular plugin architecture

Concept:

```
# plugins/weather.py
class WeatherPlugin:
    def can_handle(self, message):
        return "weather" in message.lower()

    def process(self, message):
        # Call weather API
        return "It's sunny today!"
```

```
# plugins/calendar.py
class CalendarPlugin:
    def can_handle(self, message):
        return "event" in message.lower()

    def process(self, message):
        # Return events
        return "Upcoming: Science Fair on Nov 15"

# Load plugins
plugins = [WeatherPlugin(), CalendarPlugin()]

# Use in main app
for plugin in plugins:
    if plugin.can_handle(user_message):
        return plugin.process(user_message)
```

Knowledge Check Quizzes

Quiz 1: Python Fundamentals

Question 1: What will this code output?

```
student = {"name": "Alice", "grade": 10}
print(student["grade"])
```

A) `{"grade": 10}` B) `10` C) `Alice` D) Error

Answer: B) `10`

Question 2: What does this function return?

```
def calculate(x, y=5):
    return x + y

result = calculate(3)
```

A) `3` B) `5` C) `8` D) Error

Answer: C) `8` (uses default value y=5)

Question 3: Which is the correct way to initialize session state?

```
A) st.session_state.messages = []
B) if "messages" not in st.session_state:
       st.session_state.messages = []
```

```
C) st.init_session("messages", [])
D) session_state["messages"] = []
```

Answer: B) Checks if exists first, then initializes

Quiz 2: OpenAI API

Question 1: What does `temperature=0.0` do? A) Makes responses more creative B) Makes responses deterministic and focused C) Sets the API timeout D) Controls response length

Answer: B) More deterministic and focused

Question 2: What is a token? A) API authentication key B) Unit of text processed by AI (≈4 characters) C) A response from the API D) Error code

Answer: B) Unit of text

Question 3: What's the maximum context length for gpt-4o-mini? A) 1,000 tokens B) 4,096 tokens C) 16,385 tokens D) Unlimited

Answer: C) 16,385 tokens

Quiz 3: Streamlit

Question 1: Why use `st.form()`? A) To make the UI look better B) To group inputs and submit together C) Required for all inputs D) To add CSS styling

Answer: B) Groups inputs for single submission

Question 2: What does `st.rerun()` do? A) Restarts the server B) Re-executes the entire script C) Reloads the browser D) Clears session state

Answer: B) Re-executes the script from top

Question 3: How do you add custom CSS in Streamlit?

```
A) st.css("body { color: red; }")
B) st.style("<style>...</style>")
C) st.markdown("<style>...</style>", unsafe_allow_html=True)
D) Not possible
```

Answer: C) Using markdown with unsafe_allow_html=True

Project Challenges

Challenge 1: Multi-User Support

Goal: Allow multiple students to use the chatbot with separate histories

Requirements:

- Login system with student ID
- Separate chat history per student
- Student-specific responses (grade level, classes)

Hints:

```
# Session storage per user
user_id = st.text_input("Student ID:")

if user_id:
    history_file = f"chat_history_{user_id}.pkl"
    st.session_state.messages = load_history(history_file)
```

Challenge 2: Voice Integration

Goal: Add voice input and output

Requirements:

- Speech-to-text for questions
- Text-to-speech for responses
- Works in browser

Hints:

```
# Use streamlit-webrtc or similar
# Speech recognition with browser API
# Or integrate with OpenAI Whisper API
```

Challenge 3: Smart Notifications

Goal: Send reminders about homework and events

Requirements:

- Check upcoming deadlines
- Send notifications (email or push)
- User preferences for notifications

Hints:

```
from datetime import datetime, timedelta

def check_deadlines():
    tomorrow = datetime.now() + timedelta(days=1)

    for subject, hw in HOMEWORK_ASSIGNMENTS.items():
        due_date = datetime.strptime(hw["due_date"], "%Y-%m-%d")
        if due_date == tomorrow:
```

```
            send_notification(f"Reminder: {subject} homework due
tomorrow!")
```

Challenge 4: Integration with School Systems

Goal: Connect to real school database/API

Requirements:

- Fetch live homework assignments
- Get real-time schedule changes
- Pull current events from school calendar

Concept:

```
import requests

def get_live_homework(student_id):
    response = requests.get(
        f"https://school-api.edu/homework/{student_id}",
        headers={"Authorization": f"Bearer {API_KEY}"}
    )
    return response.json()

# Use live data instead of static
HOMEWORK_ASSIGNMENTS = get_live_homework(student_id)
```

Quick Reference Guide

Essential Streamlit Commands

```
# Text
st.title("Title")
st.header("Header")
st.subheader("Subheader")
st.write("Text")
st.markdown("**Bold** *italic*")

# Input Widgets
text = st.text_input("Label:")
number = st.number_input("Number:")
option = st.selectbox("Choose:", ["A", "B"])
checked = st.checkbox("Check me")
clicked = st.button("Click")

# Layout
with st.sidebar:
    # Sidebar content
```

```
col1, col2 = st.columns(2)
with col1:
    # Column 1
with col2:
    # Column 2

with st.expander("Click to expand"):
    # Hidden content

# Forms
with st.form("form_name"):
    input1 = st.text_input("Input:")
    submit = st.form_submit_button("Submit")
    if submit:
        # Process

# Status
st.success("Success!")
st.info("Info")
st.warning("Warning")
st.error("Error")

# Progress
with st.spinner("Loading..."):
    # Long operation

# Session State
st.session_state.key = value
value = st.session_state.key
```

OpenAI API Quick Reference

```
import openai

# Initialize
client = openai.OpenAI(api_key="sk-...")

# Simple request
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "You are helpful"},
        {"role": "user", "content": "Hello"}
    ]
)

answer = response.choices[0].message.content

# With parameters
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=messages,
```

```
    max_tokens=1000,      # Response length
    temperature=0.7,      # Creativity (0-1)
    top_p=1.0,           # Nucleus sampling
    frequency_penalty=0,  # Repetition penalty
    presence_penalty=0    # Topic diversity
)
```

Python Essentials

```
# Lists
my_list = [1, 2, 3]
my_list.append(4)
my_list.remove(2)
length = len(my_list)

# Dictionaries
my_dict = {"key": "value"}
my_dict["new_key"] = "new_value"
value = my_dict.get("key", "default")

# Loops
for item in my_list:
    print(item)

for key, value in my_dict.items():
    print(f"{key}: {value}")

# Conditionals
if condition:
    # Do something
elif other_condition:
    # Do something else
else:
    # Default

# Functions
def my_function(param1, param2="default"):
    """Docstring"""
    result = param1 + param2
    return result

# Error Handling
try:
    risky_operation()
except ValueError as e:
    print(f"Error: {e}")
except Exception as e:
    print(f"Unexpected: {e}")
finally:
    cleanup()
```

Common Patterns

Retry with Backoff:

```
for attempt in range(max_retries):
    try:
        result = api_call()
        break
    except Exception as e:
        if attempt < max_retries - 1:
            time.sleep(2 ** attempt)
        else:
            return error_message
```

Loading with Spinner:

```
with st.spinner("Processing..."):
    result = long_operation()
st.success("Done!")
```

Save/Load with Pickle:

```
import pickle

# Save
with open("file.pkl", "wb") as f:
    pickle.dump(data, f)

# Load
with open("file.pkl", "rb") as f:
    data = pickle.load(f)
```

Glossary

API (Application Programming Interface): A way for programs to communicate with each other

Token: Unit of text processed by AI, approximately 4 characters or 0.75 words

Session State: Streamlit's method of preserving data across script reruns

Pickle: Python library for saving objects to files

Environment Variable: Configuration stored outside code, usually in .env file

Callback: Function that runs when an event occurs (like button click)

Prompt Engineering: Crafting effective instructions for AI

Context Window: Maximum amount of text AI can process at once

Temperature: Controls randomness in AI responses (0=focused, 1=creative)

Retry Logic: Automatically trying again when something fails

Exponential Backoff: Waiting longer between each retry attempt

Sanitization: Cleaning user input to remove dangerous content

Rate Limiting: Restricting how many requests can be made in a time period

Deployment: Making your app available online for others to use

Semantic Versioning: Version numbering system (MAJOR.MINOR.PATCH)

Next Steps & Resources

What to Learn Next

1. Advanced Python:

- Object-Oriented Programming (Classes)
- List comprehensions
- Decorators
- Context managers

2. Database Integration:

- SQLite basics
- PostgreSQL
- MongoDB for chat storage

3. Advanced AI:

- Fine-tuning models
- Embeddings for semantic search
- RAG (Retrieval Augmented Generation)

4. Production Deployment:

- Docker containerization
- Cloud platforms (AWS, Google Cloud)
- CI/CD pipelines

Recommended Resources

Python:

- Python.org Official Tutorial
- Real Python
- Automate the Boring Stuff

Streamlit:

- Streamlit Documentation
- Streamlit Gallery
- Streamlit Forum

OpenAI:

- OpenAI Documentation
- OpenAI Cookbook
- Prompt Engineering Guide

General Programming:

- Stack Overflow - Q&A
- GitHub - Code repositories
- freeCodeCamp - Tutorials

Final Project Ideas

1. Subject-Specific Tutor

Build an AI tutor for a specific subject (Math, Science, History)

- Practice problems
- Step-by-step solutions
- Progress tracking

2. College Application Assistant

Help students with college applications

- Essay review
- Deadline tracking
- Recommendation letter templates

3. Study Group Coordinator

Coordinate study groups among students

- Schedule finding
- Topic matching
- Virtual meeting rooms

4. Career Guidance Bot

Guide students on career paths

- Career exploration
- College major suggestions
- Job market insights

5. Mental Health Support

Provide emotional support and resources

- Stress management tips
- Resource directory
- Crisis hotline information

Conclusion

Congratulations! 🎉

You've completed a comprehensive journey through AI chatbot development!

What You've Learned:

- ✅ Python programming fundamentals
- ✅ Web development with Streamlit
- ✅ AI integration with OpenAI
- ✅ Data management and storage
- ✅ User interface design
- ✅ Security and privacy
- ✅ Testing and deployment
- ✅ Professional best practices

You Can Now:

- Build AI-powered applications
- Integrate APIs into web apps
- Manage state and data persistence
- Deploy applications online
- Write clean, maintainable code
- Debug and troubleshoot issues

Keep Building!

The best way to learn is by doing. Take this foundation and:

1. Customize this chatbot for your school
2. Add your own creative features
3. Build entirely new applications
4. Share your creations with others

Remember:

- Start small, iterate often
- Don't be afraid of errors - they're learning opportunities
- Read documentation
- Ask questions in communities
- Keep practicing!

Happy Coding! 💻✨

About This Guide

Version: 1.0 **Last Updated:** October 2025 **Author:** AI Chatbot Learning Project **License:** MIT

How to Use This Guide:

1. Start from Chapter 1 and work through sequentially
2. Complete exercises after each chapter
3. Build the project as you learn
4. Refer back as a reference guide

Converting to PDF:

Using Pandoc:

```
pandoc COMPLETE_LEARNING_GUIDE.md -o LearningGuide.pdf
```

Using VS Code:

1. Install "Markdown PDF" extension
2. Open this file
3. Right-click → "Markdown PDF: Export (pdf)"

Using Online Tools:

- Dillinger.io
- Markdown-to-PDF.com
- CloudConvert.com

Thank you for using this learning guide!

If you found this helpful, please:

- ⭐ Star the repository
- 📤 Share with fellow students
- 💬 Provide feedback
- 🐛 Report any errors

Good luck with your coding journey! 🚀