OwlAI Bot: Full Documentation & Architecture.

# Introduction

OwlAI is a conversational AI bot developed to assist students in preparing for the UGC NET Paper-1 exam.   
Its internal logic is designed to be dynamic, modular, and intelligent—replicating how a human tutor understands tone, language, intent, and provides adaptive responses.  
This documentation will walk you through every line, function, and logic involved in building OwlAI. Even if you are non-technical, you will be able to understand exactly how the bot works.

# 🔄 Internal Bot Workflow – Step-by-Step

Imagine a student types: “Mujhe ICT topic samajhna hai”. Here’s exactly what happens inside the bot, like a relay race:

## 1. User sends query → ask\_route.py is triggered

This is the central controller. It receives session\_id, user\_id, and query, and begins the process of understanding and answering it.

## 2. Language Detection → language\_detect.py

Why? Because we need to reply in the same language the user spoke (English, Hindi, Hinglish).

## 3. Intent & Tone Detection → intent\_tone\_classifier.py

Why? Because OwlAI needs to know: What the user wants (e.g., quiz, explanation), and what is their tone (casual, confused, serious).

## 4. Context Fetching → vector\_search.py

Now we search our Pinecone vector DB to fetch relevant syllabus or concepts. These are added into the prompt to help the LLM generate accurate, specific answers.

## 5. Prompt Building → llm.py

This is the most important function. It constructs a smart prompt containing context, tone, language, previous session messages, and the user’s current query.

## 6. Query sent to OpenAI GPT → get\_response\_from\_llm() in llm.py

This function calls OpenAI's GPT model using the generated prompt to get a human-like answer.

## 7. Moderation Check → moderation.py

We run the LLM reply through a safety check to filter out inappropriate or unsafe content.

## 8. Response Enhancer → response\_control.yaml

We use this YAML config to add follow-ups and make replies more human-like and contextual based on the user's tone and language.

## 9. Session Memory → session\_memory.py

Stores and retrieves previous session conversations to allow contextual continuity across messages, like ChatGPT.

## 10. If Intent is Quiz → quiz\_generator.py

If user intent is a quiz, we skip LLM and dynamically generate questions using GPT based on topic and language.

# 📁 Files and Their Purpose

1. main.py: Starts FastAPI app.  
2. ask\_route.py: Defines the `/ask` endpoint that drives all logic.  
3. llm.py: Handles prompt creation, LLM querying, and fallback logic.  
4. language\_detect.py: Classifies input language as English, Hindi, or Hinglish.  
5. intent\_tone\_classifier.py: Extracts user's tone (confused, friendly, neutral, etc.).  
6. intent\_classifier.py: Classifies intent (ask syllabus, quiz, explanation, follow-up, etc.).  
7. moderation.py: Scans and blocks inappropriate messages.  
8. vector\_search.py: Retrieves relevant info from vector DB (FAISS/Chroma etc.).  
9. session\_memory.py: Loads last 5 messages to build context-aware prompts.  
10. quiz\_generator.py: Generates dynamic quizzes and evaluates answers.  
11. professional\_handler.py: Specialized logic (ex. syllabus explanation or topic walkthrough).  
12. embedder.py: Used in chunk embedding and ingestion of custom docs.  
13. auto\_learn.py: Not used currently; legacy experimental logic.

# ⚙️ Core Logic & Key Functions

---

***### ask\_route.py***

This file contains the `/ask` route.

```python  
@ask\_router.post("")  
@ask\_router.post("/")  
async def ask\_question(request: AskRequest):  
 session\_id = request.session\_id.strip()  
 query = request.query.strip()  
 ...  
```

- Moderation Check: `run\_moderation\_check(query)` ensures safety.  
- Intent, Tone, Language Classification: Done before building the prompt.  
- Prompt Generation: `build\_prompt()` constructs context-aware prompt.  
- LLM Query: `get\_response\_from\_llm()` hits OpenAI's endpoint.  
- Chat Saving: `save\_chat()` logs conversation.

---

***### llm.py***

This is where intelligent prompting happens.

```python  
def build\_prompt(query, intent, tone, language, session\_id, user\_id):  
 memory = get\_session\_state(session\_id)  
 instruction = get\_instruction(intent, tone, language)  
 return f"{instruction}

{memory}

User: {query}"  
```

- Combines past memory, query, tone, and instruction.  
- Handles fallbacks (e.g. GPT-3.5 if GPT-4 fails).

```python  
def get\_instruction(intent, tone, language):  
 # Maps intent + tone + lang to specific instructions  
```

---

***### session\_memory.py***

Handles per-session memory:

```python  
def get\_session\_state(session\_id):  
 return get\_last\_chats(session\_id, limit=5)  
```

Keeps the LLM responses aware of context.

---

***### vector\_search.py***

When needed, retrieves chunks from DB:  
```python  
def query\_vector\_store(query):  
 return FAISS.similarity\_search(query)  
```

Returns data to enrich prompt context.

---

***### moderation.py***

Checks against OpenAI's moderation filter.

```python  
def run\_moderation\_check(text):  
 result = openai.Moderation.create(input=text)  
 return result["results"][0]["flagged"]  
```

---

# 📢 Prompt Engineering Strategy

Each prompt sent to GPT is made of:

1. System Prompt (via `get\_instruction()`):  
 - Guides model on how to behave. Example:  
 ```  
 You are OwlAI, a UGC NET Paper-1 expert. Use a friendly and professional tone. Speak in {language}. Handle queries about {intent} smartly.  
 ```

2. Memory Block:  
 - Last 5 messages from the session to keep the chat contextual.

3. User Query:  
 - Cleaned input message.

This way, every response is dynamic and guided by past history, tone, language, and intent.