Costruzione di un Chatbot CLI con Python e OpenAI

Benvenuti a questo corso dedicato alla costruzione di un chatbot a riga di comando utilizzando Python e le API di OpenAI. Impareremo passo dopo passo come creare un'applicazione interattiva che comunica con i modelli linguistici avanzati di OpenAI, gestisce le conversazioni e si comporta in modo professionale anche in presenza di errori.

Questo corso è pensato per programmatori con conoscenze base di Python che desiderano esplorare le potenzialità dell'intelligenza artificiale nelle loro applicazioni. Non è richiesta esperienza pregressa con OpenAI.

Creazione dello Script Python Base

Il primo passo per costruire il nostro chatbot CLI è creare uno script Python base che conterrà la struttura fondamentale del nostro programma. Inizieremo importando le librerie necessarie e configurando l'ambiente di sviluppo.

Le librerie essenziali includono **openai** per interagire con l'API, **os** per gestire variabili d'ambiente, e **dotenv** per caricare configurazioni da file esterni. Questo approccio ci permetterà di mantenere privati i nostri dati sensibili come la chiave API.

1 Installazione delle dipendenze

Utilizziamo pip per installare i pacchetti necessari:

```
pip install openai python-dotenv
```

2 Creazione dello script base

Creiamo un file chiamato **chatbot.py** con la struttura iniziale:

```
import openai
import os
from dotenv import load_dotenv

load_dotenv()  # Carica variabili da .env
openai.api_key = os.getenv("OPENAI_API_KEY")
```

3 Configurazione della chiave API

Creiamo un file **.env** nella stessa directory e inseriamo la nostra chiave API:

```
OPENAI_API_KEY=sk-xxxxxxxxxxxxxxxxxxxxxxx
```

Gestione dell'Input Utente da Terminale

Per rendere interattivo il nostro chatbot, dobbiamo implementare la gestione dell'input utente da terminale. Questo è un aspetto fondamentale per creare un'esperienza conversazionale fluida tra l'utente e il modello di intelligenza artificiale.

Utilizzeremo la funzione **input()** di Python che permette di richiedere e catturare testo inserito dall'utente. Possiamo personalizzare il prompt visualizzato per rendere l'interfaccia più chiara e intuitiva.

Implementazione dell'Input

```
def get_user_input():
    user_input = input("Tu: ")
    return user_input

# Nel ciclo principale
while True:
    user_message = get_user_input()
    if user_message.lower() == "exit":
        print("Arrivederci!")
        break
    # Qui invieremo il messaggio all'API
```

Considerazioni Importanti

- Aggiungere un comando di uscita (es. "exit") per terminare la conversazione
- Gestire input vuoti o non validi prima di inviarli all'API
- Considerare l'aggiunta di un prompt personalizzabile per l'input
- Implementare la pulizia dell'input per rimuovere spazi extra o caratteri problematici

Un'interfaccia utente ben progettata migliora significativamente l'esperienza d'uso del chatbot e riduce la probabilità di errori durante l'interazione.

Invio dell'Input a OpenAI API

Una volta ottenuto l'input dell'utente, il passo successivo è inviarlo all'API di OpenAI per generare una risposta appropriata. Utilizzeremo la funzione **openai.resources.chat.completions.create** per comunicare con il modello linguistico.

Questa funzione richiede parametri specifici come il modello da utilizzare (es. "gpt-3.5-turbo") e una lista di messaggi che rappresentano la conversazione fino a quel momento. La risposta ottenuta conterrà il testo generato dal modello AI.

Preparazione del Payload

Creiamo una lista di messaggi nel formato richiesto dall'API:

```
messages = [
    {"role": "system", "content": "Sei un assistente amichevole."},
    {"role": "user", "content": user_message}
]
```

Invio della Richiesta

Utilizziamo il client OpenAI per inviare la richiesta all'API:

```
from openai import OpenAI
client = OpenAI()

response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=messages
)
```

Estrazione della Risposta

Estraiamo il contenuto della risposta dal modello:

```
assistant_response = response.choices[0].message.content
print(f"Assistente: {assistant_response}")
```

Costruzione del Contesto Iniziale (System Prompt)

Il system prompt è un elemento cruciale che definisce la personalità, le capacità e i limiti del nostro chatbot. Questo messaggio iniziale, invisibile all'utente finale, fornisce al modello linguistico le istruzioni su come comportarsi durante la conversazione.

Un system prompt ben progettato può trasformare il modello generico in un assistente specializzato, adattandolo alle esigenze specifiche dell'applicazione. Può includere tono, stile di risposta, conoscenze specifiche e regole comportamentali.

Personalizzazione

Definisce il ruolo del chatbot (esperto, assistente, coach), il tono di voce (formale, amichevole, tecnico) e lo stile di risposta (conciso, dettagliato, semplice).

Limiti e Sicurezza

Stabilisce cosa il chatbot può o non può fare, argomenti da evitare e come gestire richieste inappropriate o al di fuori del suo ambito di competenza.

Conoscenze Specifiche

Indica al modello le conoscenze specifiche che dovrebbe utilizzare o le fonti da cui attingere informazioni, rendendo le risposte più accurate nel contesto dell'applicazione.

```
system_message = {
    "role": "system",
    "content": "Sei un assistente esperto di programmazione Python. Fornisci risposte concise e tecnicamente accurate. Includi esempi di codice quando appropriato. Se non conosci la risposta, ammettilo chiaramente invece di inventare informazioni."
}
```

Memorizzazione dello Storico dei Messaggi

Per creare conversazioni coerenti e contestuali, è fondamentale memorizzare lo storico dei messaggi scambiati tra l'utente e il chatbot. Questa memoria conversazionale permette al modello di comprendere il contesto precedente e fornire risposte pertinenti.

Implementeremo questa funzionalità utilizzando una struttura dati lista che conterrà dizionari rappresentanti ciascun messaggio. Ogni dizionario avrà due chiavi principali: "role" (che può essere "system", "user" o "assistant") e "content" (il contenuto testuale del messaggio).

Implementazione della Memoria

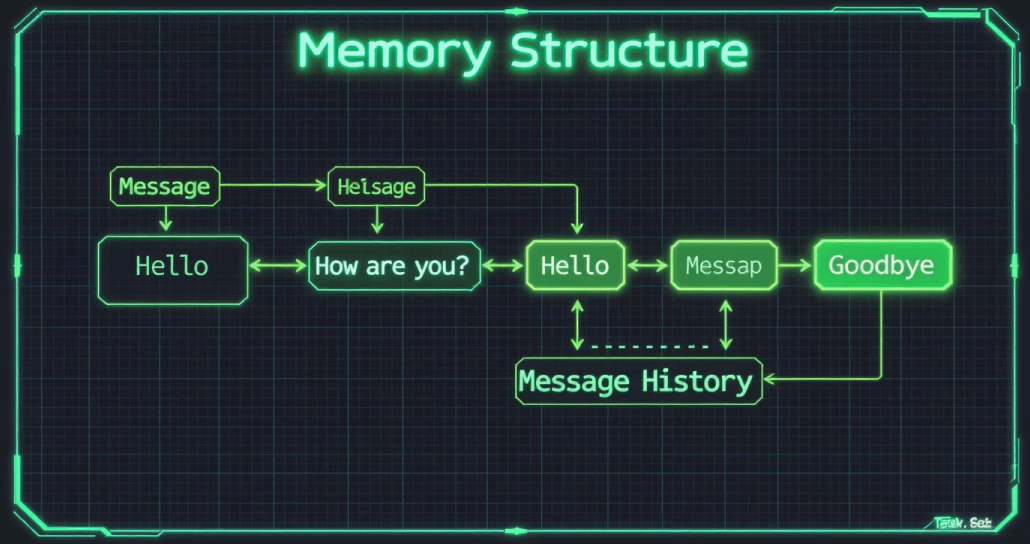
```
# Inizializzazione della memoria
messages = [
    {"role": "system", "content": system_message}
]

# Durante la conversazione
def chat_with_ai(user_input):
    # Aggiungi il messaggio dell'utente
    messages.append({"role": "user", "content": user_input})

    # Ottieni risposta dall'API
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=messages
    )

    # Estrai e memorizza la risposta
    assistant_response = response.choices[0].message.content
    messages.append({"role": "assistant", "content": assistant_response})

    return assistant_response
```



Mantenere lo storico completo dei messaggi permette al modello di fare riferimento a informazioni precedenti, rispondere a follow-up e mantenere una conversazione naturale. Tuttavia, è importante considerare i limiti di token dell'API per evitare errori quando la conversazione diventa troppo lunga.

Aggiunta di Loop per Conversazioni Multiple

Per rendere il nostro chatbot veramente interattivo, dobbiamo implementare un ciclo di conversazione che permetta scambi multipli tra l'utente e l'assistente AI. Questo loop continuerà fino a quando l'utente non deciderà di terminare la conversazione.

Il ciclo di base consiste nel raccogliere l'input dell'utente, inviarlo all'API di OpenAI, ricevere la risposta, mostrarla all'utente e quindi ripetere il processo. È importante includere anche un meccanismo di uscita che permetta all'utente di terminare la conversazione quando lo desidera.

Raccolta Input

Il programma attende che l'utente digiti un messaggio e lo raccoglie tramite la funzione input().

Invio all'API

Il messaggio dell'utente viene aggiunto allo storico e inviato all'API OpenAI per l'elaborazione.

Visualizzazione

La risposta dell'assistente viene mostrata all'utente e aggiunta allo storico della conversazione.

Elaborazione Risposta

L'API genera una risposta basata sul contesto della conversazione e la restituisce al programma.

```
def main():
    print("Benvenuto al chatbot CLI! Scrivi 'exit' per uscire.")

    while True:
        user_input = input("\nTu: ")

        if user_input.lower() in ["exit", "quit", "bye"]:
            print("Assistente: Arrivederci! Grazie per aver chattato con me.")
            break

        response = chat_with_ai(user_input)
        print(f"\nAssistente: {response}")
```

Gestione degli Errori delle API

Quando si lavora con API esterne come quella di OpenAI, è fondamentale implementare una robusta gestione degli errori. Le chiamate API possono fallire per vari motivi: problemi di connessione, errori di autenticazione, limiti di rate, o problemi sul lato server.

Gestire correttamente questi errori migliora l'affidabilità del nostro chatbot e fornisce un'esperienza utente migliore, evitando arresti imprevisti del programma e offrendo feedback utili quando qualcosa va storto.

Eccezioni Specifiche

OpenAI fornisce eccezioni specifiche come **openai.OpenAIError** che possiamo catturare. Ci sono anche sottoclassi più specifiche per vari tipi di errori:

- APIError: problemi generali con l'API
- AuthenticationError: chiave API non valida
- RateLimitError: troppe richieste in poco tempo
- APIConnectionError: problemi di connettività

Implementazione Try-Except

```
from openai import OpenAI, OpenAIError, APIError, RateLimitError

def chat_with_ai(user_input):
    try:
        # Aggiungi il messaggio dell'utente
        messages.append({"role": "user", "content": user_input})

        # Ottieni risposta dall'API
        response = client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=messages
        )

        # Estrai e memorizza la risposta
        assistant_response = response.choices[0].message.content
        messages.append({"role": "assistant", "content": assistant_response})

        return assistant_response

    except RateLimitError:
        return "Mi dispiace, ho raggiunto il limite di richieste. Riprova tra qualche minuto."
    except APIError as e:
        return f"Si è verificato un errore API: {str(e)}"
    except OpenAIError as e:
        return f"Si è verificato un errore: {str(e)}"
```

Gestione dell'Interruzione con KeyboardInterrupt

Un aspetto importante dell'usabilità del nostro chatbot CLI è la capacità di terminare l'esecuzione in modo pulito quando l'utente preme Ctrl+C. Questa combinazione di tasti genera un'eccezione KeyboardInterrupt in Python, che dobbiamo gestire adeguatamente.

Senza una gestione appropriata, un'interruzione da tastiera potrebbe terminare il programma bruscamente, potenzialmente perdendo dati o lasciando risorse aperte. Implementando un gestore per KeyboardInterrupt, possiamo eseguire operazioni di pulizia e fornire un messaggio di uscita elegante.

Implementazione della Gestione dell'Interruzione

```
def main():
    print("Benvenuto al chatbot CLI! Scrivi 'exit' per uscire o premi Ctrl+C.")

    try:
        while True:
            user_input = input("\nTu: ")

            if user_input.lower() in ["exit", "quit", "bye"]:
                break

            response = chat_with_ai(user_input)
            print(f"\nAssistente: {response}")

    except KeyboardInterrupt:
        print("\n\nOperazione interrotta dall'utente.")
    finally:
        print("Grazie per aver utilizzato il chatbot CLI. Arrivederci!")

        # Operazioni di pulizia
        # Ad esempio: salvataggio della conversazione
        if len(messages) > 1:  # Se c'è stata una conversazione
            save_conversation_to_log(messages)

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

Vantaggi della Gestione dell'Interruzione

- Fornisce un'esperienza utente più professionale
- Permette l'esecuzione di operazioni di pulizia
- Previene la perdita di dati in caso di interruzione
- Mostra un messaggio di uscita chiaro

Il blocco **finally** è particolarmente utile poiché viene eseguito sia in caso di uscita normale che in caso di interruzione, garantendo che le operazioni di chiusura vengano sempre eseguite.

Potrebbe essere utile aggiungere anche un messaggio che indica come uscire correttamente dal programma all'inizio dell'esecuzione, per guidare l'utente.

Logging delle Conversazioni su File

Il logging delle conversazioni è una funzionalità importante per il nostro chatbot CLI. Registrare gli scambi tra l'utente e l'assistente AI permette di analizzare le conversazioni, migliorare il sistema nel tempo e fornire una cronologia consultabile.

Implementeremo un sistema di logging che salva le conversazioni in file di testo, organizzati per data e ora. Ogni log conterrà l'intero scambio di messaggi, con chiara indicazione di chi ha detto cosa.

Configurazione del Logger

```
import logging
import datetime
import os

def setup_logger():
    # Crea la directory dei log se non esiste
    if not os.path.exists("logs"):
        os.makedirs("logs")

    # Configura il logger
    current_time = datetime.datetime.now().strftime("%Y-%m-
%d_%H-%M-%S")
    log_file = f"logs/chat_{current_time}.log"

    logging.basicConfig(
        filename=log_file,
        level=logging.INFO,
        format='%(asctime)s - %(levelname)s - %(message)s'
    )

    return logging.getLogger("ChatbotLogger")
```

Registrazione dei Messaggi

```
def log_message(logger, role, content):
    logger.info(f"{role.upper()}: {content}")
```

Integrazione nel Ciclo Principale

```
def main():
    logger = setup_logger()
    logger.info("--- NUOVA SESSIONE DI CHAT AVVIATA ---")

    try:
        while True:
            user_input = input("\nTu: ")

            # Registra l'input dell'utente
            log_message(logger, "user", user_input)

            if user_input.lower() in ["exit", "quit", "bye"]:
                break

            response = chat_with_ai(user_input)
            print(f"\nAssistente: {response}")

            # Registra la risposta dell'assistente
            log_message(logger, "assistant", response)

    except KeyboardInterrupt:
        logger.info("Chat interrotta dall'utente")
    finally:
        logger.info("--- SESSIONE DI CHAT TERMINATA ---")
```

Aggiunta del Parametro Temperature Configurabile

Il parametro **temperature** controlla la creatività e casualità delle risposte generate dal modello OpenAI. Valori più bassi (vicino a 0) producono risposte più deterministiche e focalizzate, mentre valori più alti (fino a 2) generano output più creativi e variabili.

Rendere questo parametro configurabile dall'utente permette di adattare il comportamento del chatbot alle esigenze specifiche: risposte precise e coerenti per compiti tecnici o informativi, oppure risposte più creative per brainstorming o intrattenimento.

Implementazione della Temperature Configurabile

```
def chat_with_ai(user_input, temperature=0.7):
    """
    Invia un messaggio all'API OpenAI e ottiene una risposta.

    Args:
        user_input (str): Il messaggio dell'utente
        temperature (float): Valore tra 0 e 2. Controlla la casualità.
            - Valori più bassi: risposte più deterministiche
            - Valori più alti: risposte più creative

    Returns:
        str: La risposta generata dall'AI
    """
    try:
        messages.append({"role": "user", "content": user_input})

        response = client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=messages,
            temperature=temperature
        )

        assistant_response = response.choices[0].message.content
        messages.append({"role": "assistant", "content": assistant_response})

        return assistant_response

    except Exception as e:
        return f"Si è verificato un errore: {str(e)}"
```

Comandi per Modificare la Temperature

Possiamo implementare comandi speciali che permettono all'utente di modificare la temperature durante la conversazione:

```
def main():
    temperature = 0.7  # Valore predefinito

    print("Benvenuto al chatbot CLI!")
    print("Comandi speciali:")
    print(" - /temp 0.2    : Imposta temperature bassa (più preciso)")
    print(" - /temp 1.5    : Imposta temperature alta (più creativo)")
    print(" - /exit        : Termina la conversazione")

    while True:
        user_input = input("\nTu: ")

        # Controllo per comandi speciali
        if user_input.startswith("/temp "):
            try:
                new_temp = float(user_input.split()[1])
                if 0 <= new_temp <= 2:
                    temperature = new_temp
                    print(f"Temperature impostata a {temperature}")
                else:
                    print("Temperature deve essere tra 0 e 2")
            except:
                print("Formato non valido. Usa '/temp 0.7'")
            continue

        elif user_input.lower() == "/exit":
            break

        response = chat_with_ai(user_input, temperature)
        print(f"\nAssistente: {response}")
```

Valori consigliati per temperature:

- **0.2-0.4:** Risposte molto precise e deterministiche
- **0.7-0.8:** Buon equilibrio (valore predefinito)
- **1.0-1.5:** Risposte più creative e varie

Modularizzazione del Codice

Con l'aumentare della complessità del nostro chatbot, diventa essenziale organizzare il codice in moduli logici e riutilizzabili. La modularizzazione migliora la leggibilità, facilita la manutenzione e consente una più facile estensione delle funzionalità in futuro.

Suddivideremo il nostro progetto in diversi file Python, ciascuno con una responsabilità specifica. Questo approccio favorisce il principio di "singola responsabilità" e rende il codice più gestibile, soprattutto quando il progetto cresce.

Vantaggi della Modularizzazione

- **Manutenibilità:** Più facile trovare e correggere bug
- **Riutilizzabilità:** Componenti possono essere usati in altri progetti
- **Testabilità:** Più facile scrivere test unitari per componenti isolati
- **Collaborazione:** Diversi sviluppatori possono lavorare su moduli diversi
- **Comprensibilità:** Struttura chiara e separazione delle responsabilità

Esempio di Organizzazione

```
chatbot_cli/
├── main.py          # Script principale
├── chatbot.py       # Logica del chatbot
├── openai_client.py  # Wrapper per OpenAI API
├── token_manager.py  # Gestione dei token
├── logger.py        # Sistema di logging
├── config.py        # Configurazioni
├── utils.py         # Funzioni di utilità
├── .env             # Variabili d'ambiente
└── requirements.txt  # Dipendenze
```

Ogni modulo dovrebbe avere una funzione specifica e un'interfaccia chiara per l'interazione con gli altri moduli. Questo approccio facilita la manutenzione e l'estensione del codice nel tempo.

Creazione di un File di Configurazione

Per rendere il nostro chatbot più flessibile e facilmente configurabile, implementeremo un sistema di configurazione centralizzato. Questo ci permetterà di modificare il comportamento dell'applicazione senza dover cambiare il codice sorgente, migliorando la manutenibilità e la personalizzazione.

Utilizzeremo un file di configurazione in formato JSON o YAML che conterrà tutte le impostazioni rilevanti come il modello da utilizzare, la temperature predefinita, il system prompt e altre opzioni specifiche del chatbot.

Esempio di File di Configurazione (config.json)

```
{
  "openai": {
    "model": "gpt-3.5-turbo",
    "temperature": 0.7,
    "max_tokens": 150,
    "system_prompt": "Sei un assistente Python esperto..."
  },
  "interface": {
    "user_prompt": "Tu: ",
    "assistant_prefix": "Assistente: ",
    "welcome_message": "Benvenuto! Scrivi 'exit' per uscire."
  },
  "logging": {
    "enabled": true,
    "log_dir": "logs",
    "log_level": "INFO"
  },
  "advanced": {
    "token_limit": 3000,
    "history_truncation": true,
    "save_conversations": true
  }
}
```

Implementazione del Caricamento della Configurazione

```
# config.py
import json
import os
from dotenv import load_dotenv

# Carica variabili d'ambiente
load_dotenv()

def load_config(config_path="config.json"):
    """Carica la configurazione dal file JSON."""
    try:
        with open(config_path, 'r') as f:
            config = json.load(f)

        # Sovrascrivi con variabili d'ambiente se presenti
        if os.getenv("OPENAI_API_KEY"):
            config["openai"]["api_key"] = os.getenv("OPENAI_API_KEY")

        if os.getenv("OPENAI_MODEL"):
            config["openai"]["model"] = os.getenv("OPENAI_MODEL")

        return config
    except Exception as e:
        print(f"Errore nel caricamento della configurazione: {e}")
        # Restituisci una configurazione predefinita
        return get_default_config()

def get_default_config():
    """Restituisce una configurazione predefinita in caso di errori."""
    return {
        "openai": {
            "model": "gpt-3.5-turbo",
            "temperature": 0.7,
            "api_key": os.getenv("OPENAI_API_KEY", "")
        },
        # Altre impostazioni predefinite...
    }
```

Utilizzare un file di configurazione esterno consente agli utenti di personalizzare facilmente il comportamento del chatbot senza dover modificare il codice sorgente. Questo è particolarmente utile per adattare il chatbot a diversi casi d'uso o per sperimentare con diverse impostazioni.

Debug e Testing del Chatbot

Il debug e il testing sono fasi cruciali nello sviluppo del nostro chatbot CLI. Implementeremo funzionalità di debug per visualizzare il payload inviato all'API e condurremo test sistematici con vari prompt per assicurarci che il chatbot funzioni correttamente in diverse situazioni.

Un buon sistema di debug ci aiuta a identificare e risolvere problemi rapidamente, mentre il testing sistematico garantisce che il nostro chatbot sia robusto e fornisca un'esperienza utente affidabile.

1 Implementazione della Modalità Debug

```
# Aggiunta di un flag di debug nella configurazione
config = {
    "debug": {
        "enabled": True,
        "show_tokens": True,
        "show_payload": True
    },
    # Altre configurazioni...
}

def chat_with_ai(user_input, temperature=0.7):
    global messages, config

    try:
        messages.append({"role": "user", "content": user_input})

        # Debug: visualizza token count
        if config["debug"]["show_tokens"]:
            token_count = num_tokens_from_messages(messages)
            print(f"\n[DEBUG] Token count: {token_count}")

        # Debug: visualizza payload
        if config["debug"]["show_payload"]:
            print("\n[DEBUG] Payload inviato all'API:")
            print(f"Model: {config['openai']['model']}")
            print(f"Temperature: {temperature}")
            print("Messages:")
            for msg in messages:
                role = msg["role"]
                content = msg["content"]
                if len(content) > 50:
                    content = content[:47] + "..."
                print(f"  - {role}: {content}")

        # Chiamata API...
        response = client.chat.completions.create(
            model=config["openai"]["model"],
            messages=messages,
            temperature=temperature
        )

        # Debug: informazioni sulla risposta
        if config["debug"]["enabled"]:
            finish_reason = response.choices[0].finish_reason
            completion_tokens = response.usage.completion_tokens
            prompt_tokens = response.usage.prompt_tokens
            print(f"\n[DEBUG] Finish reason: {finish_reason}")
            print(f"[DEBUG] Tokens: {prompt_tokens} prompt + {completion_tokens} completion = {prompt_tokens + completion_tokens} total")

        assistant_response = response.choices[0].message.content
        messages.append({"role": "assistant", "content": assistant_response})

        return assistant_response

    except Exception as e:
        return f"Si è verificato un errore: {str(e)}"
```

2 Creazione di Test Sistematici

Creiamo un set di prompt di test per verificare vari aspetti del chatbot:

- **Test di base:** Domande semplici e saluti
- **Test di contesto:** Domande di follow-up che richiedono memoria della conversazione
- **Test di robustezza:** Input lunghi, caratteri speciali, input vuoti
- **Test di limiti:** Conversazioni molto lunghe per verificare la gestione dei token
- **Test di errori:** Simulazione di errori API per verificare la gestione delle eccezioni

Possiamo automatizzare questi test creando uno script che esegue sequenze di prompt predefiniti e verifica che le risposte siano appropriate.

3 Suggerimenti per il Debugging

- Usa un logger dedicato per il debug anziché print
- Implementa livelli di verbosità configurabili
- Aggiungi timestamps ai messaggi di debug
- Salva i payload completi in file separati per analisi dettagliata
- Implementa un comando speciale (es. /debug) per attivare/disattivare il debug durante l'esecuzione

Un approccio sistematico al testing e debugging ci permette di identificare e risolvere problemi prima che raggiungano gli utenti finali, migliorando la qualità complessiva del nostro chatbot CLI.