Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

NATURAL LANGUAGE PROCESSING PER IL MIGLIORAMENTO DI SERVIZI SOFTWARE INTEGRATI

Relatore:
Chiar.mo Prof.
ANDREA ASPERTI

Presentata da:
FEDERICA SANTISI

II Sessione
Anno Accademico 2024/2025

A mio fratello Stefano

Abstract

Questa tesi esplora lo sviluppo e l'integrazione di assistenti virtuali altamente specializzati, con l'obiettivo di migliorare l'efficienza e l'interattività dei servizi software. Nello specifico, il progetto descritto in questo lavoro, riguarda la creazione di un assistente virtuale basato sulle API di OpenAI, progettato per supportare gli utenti nell'utilizzo di un software dedicato alla gestione delle operazioni di una carrozzeria, come la creazione di preventivi per la riparazione di veicoli danneggiati e l'esecuzione di azioni automatizzate su richiesta. L'assistente sarà in grado non solo di rispondere a domande e offrire informazioni dettagliate sul funzionamento del software, ma anche di interagire con esso in modo dinamico, eseguendo compiti predefiniti e migliorando l'esperienza utente attraverso un'interfaccia conversazionale intuitiva.

Attraverso l'analisi delle tecnologie di intelligenza artificiale e di integrazione API, questo lavoro fornirà un'analisi approfondita delle sfide tecniche e delle soluzioni adottate per implementare un assistente virtuale efficace in contesti software complessi.

Indice

Appendice A Elenco API utilizzate

Elenco delle figure

Capitolo 0

Introduzione

Negli ultimi anni, l'uso di chatbot e assistenti virtuali è cresciuto esponenzialmente in diversi settori, dimostrandosi strumenti efficaci per migliorare l'efficienza e la qualità delle interazioni uomo-macchina. Questi strumenti non solo rispondono a domande o forniscono informazioni, ma sono anche in grado di gestire compiti complicati e automatizzare processi che tradizionalmente richiederebbero un intervento umano. I chatbot facilitano l'uso di software complessi grazie a interfacce intuitive, automatizzando attività come la gestione di ordini, la comunicazione con i clienti e la raccolta di dati. Integrati in questi sistemi, come nel progetto sviluppato per questa tesi, consentono agli utenti di accedere facilmente a funzionalità avanzate, semplificando le operazioni ripetitive e ottimizzando l'efficienza.

Tuttavia, lo scetticismo riguardo all'effettiva efficacia di questi sistemi è ancora ampiamente diffuso. Molti sviluppatori e professionisti del settore ritengono che affidare all'intelligenza artificiale compiti che potrebbero essere svolti in modo più empirico, attraverso metodi tradizionali o algoritmi ben definiti, sia un approccio più sicuro e controllabile. Spesso preferiscono soluzioni più manuali o programmazioni specifiche, in quanto le considerano più affidabili e meglio testate. D'altra parte, ci sono anche sviluppatori entusiasti delle potenzialità offerte dai più recenti progressi in campo AI, che vedono in queste tecnologie non solo uno strumento di automazione, ma quasi una capacità di pensiero indipendente. Questa iper-euforia può portare a sopravvalutare le capacità dei chatbot o degli assistenti virtuali, attribuendo loro caratteristiche che superano le loro attuali possibilità, come la capacità di ragionamento complesso o decisionale autonomo, quando in realtà restano strumenti guidati da modelli di apprendimento automatico

basati su dati preesistenti. Questa dicotomia tra scetticismo e iper-entusiasmo rappresenta uno degli ostacoli principali nello sviluppo e nell'adozione equilibrata di queste tecnologie, e sottolinea l'importanza di una visione più oggettiva e pragmatica nel loro utilizzo.

Il progetto sviluppato per questa tesi consiste in un chatbot realizzato per il software *Oxygen*. Oxygen è un software progettato appositamente per le carrozzerie automobilistiche, pensato per essere utilizzato direttamente dai carrozzieri per gestire una vasta gamma di funzionalità. L'assistente virtuale sfrutta la funzionalità di *retrieval* per rispondere a domande sul *Regolamento Ricambi*, un servizio offerto dal software, permettendo al carrozziere di concentrarsi esclusivamente sul proprio lavoro, senza preoccuparsi di come utilizzare il software. Grazie all'integrazione con il tool *function*, l'assistente è in grado di riconoscere richieste specifiche dell'utente, come il recupero di informazioni riguardanti un utente o una targa. Inoltre, l'assistente mantiene il contesto della conversazione, similmente a ChatGPT, permettendo di riprendere il dialogo senza la necessità di avviare nuovi thread, ottimizzando così l'uso delle risorse.
Questo chatbot si colloca esattamente a metà tra le due correnti di pensiero descritte, dimostrando che tali meccanismi possono realmente funzionare. Infatti, il chatbot riesce a fornire risposte pertinenti e a svolgere determinate azioni in modo autonomo quando riconosce che è necessario farlo.

La tesi è suddivisa in cinque capitoli. Il Capitolo 1 e il Capitolo 2 forniscono un background teorico, concentrandosi sugli assistenti virtuali e sul modello GPT utilizzato per il progetto. Il Capitolo 3 esamina in dettaglio la struttura del progetto, descrivendo le varie fasi di sviluppo e le tecniche adottate. Nel Capitolo 4 viene presentata un'analisi dei risultati ottenuti dalle diverse funzionalità implementate. Infine, il Capitolo 5 è dedicato ai possibili sviluppi futuri di questo progetto, considerando le numerose opportunità di evoluzione che emergono grazie al continuo progresso dei sistemi di intelligenza artificiale.

Capitolo 1

Chatbot

Negli ultimi anni, i chatbot hanno trovato applicazione in numerosi settori, grazie alla loro capacità di migliorare l'efficienza delle interazioni con gli utenti. Questi strumenti non solo ottimizzano i processi aziendali, ma consentono anche di gestire grandi volumi di richieste in maniera automatizzata, riducendo tempi e costi operativi.

1.1 Cos'è un chatbot

I chatbot, o interfacce conversazionali, forniscono un nuovo modo per le persone di collaborare con i sistemi informatici. In generale, per ottenere una risposta a una domanda da un programma software, si utilizza un motore di ricerca o si compila un modulo. Un chatbot consente a una persona di porre domande nello stesso modo in cui si rivolgerebbe a un essere umano.

La tecnologia fondamentale che guida questa crescita è il Natural Language Processing (NLP). I recenti progressi nel machine learning hanno notevolmente migliorato l'accuratezza e l'efficacia del NLP, rendendo i chatbot una scelta praticabile per molte organizzazioni. Questi sviluppi stanno stimolando ulteriori ricerche, promettendo un miglioramento continuo nell'efficacia dei chatbot nel prossimo futuro.[2]

1.2 Simple vs Intelligent chatbot

Si può affermare che fondamentalmente esistono due tipi di chatbot: *semplici* e *intelligenti*.

I chatbot *semplici* operano seguendo regole predefinite e script. Sono progettati per gestire conversazioni limitate e guidare l'utente verso una risposta specifica o l'esecuzione di un'azione. Rispondono solo a input programmati e non possono interpretare domande che esulano dai loro script. Di solito vengono utilizzati per compiti ben definiti, come lo shopping online o la fornitura di informazioni sul meteo. Sono più semplici da progettare e forniscono risposte prevedibili.

I chatbot *intelligenti*, invece, sono basati su tecnologie di *intelligenza artificiale*, *machine learning* e *natural language processing*. Anche se il termine "intelligente" potrebbe suggerire capacità simili alla capacità di pensare e ragionare del pensiero umano, questo non è chiaramente possibile. I chatbot intelligenti infatti, si distinguono per la loro capacità di comprendere il linguaggio naturale, oltre alle semplici parole chiave, apprendono dalle interazioni e migliorano nel tempo. Utilizzano pattern per interpretare le richieste degli utenti e necessitano di un ampio database di esempi per l'addestramento. Tuttavia, programmare questi pattern per rispondere a una varietà di intenti può risultare complesso. [1]

1.3 Assistenti Virtuali

Gli *Assistenti Virtuali* rappresentano una categoria di *Intelligent chatbot* alimentati dall'intelligenza artificiale, progettati per simulare conversazioni umane tramite testo e voce. Questi sistemi non si limitano a comprendere le richieste degli utenti, ma sono anche in grado di adattarsi e migliorare continuamente grazie alla capacità di *apprendere automaticamente* dalle interazioni precedenti. Analizzano il *linguaggio naturale* con grande precisione, considerando le sfumature lessicali, grammaticali, sintattiche e semantiche, e regolano il proprio comportamento in base al contesto conversazionale.

Grazie a tecnologie avanzate come l'apprendimento automatico (Machine Learning, ML) e l'elaborazione e comprensione del linguaggio naturale (Natural-Language Processing e Understanding, NLP e NLU), questi assistenti possono non solo rispondere in modo coerente, ma anche mantenere una continuità nel flusso di dialogo, ricordando informazioni fornite dall'utente in precedenti conversazioni. La loro versatilità permette di svolgere una serie di attività, che vanno dall'organizzazione di appuntamenti alla gestione di dispositivi smart, rendendo più intuitiva l'interazione con la tecnologia. Assistenti come Siri o Alexa sono esempi concreti di come questi strumenti siano diventati parte integrante della vita quotidiana, semplificando numerose operazioni attraverso una costante e naturale interazione con l'utente. [5]

1.4 Perché investire in un chatbot

Risultano numerosi i motivi per cui un azienda dovrebbe investire in un chatbot a supporto dei proprio servizi.

Innanzitutto, un chatbot consente di offrire un'assistenza continua, attiva 24 ore su 24, 7 giorni su 7, il che significa che i clienti possono ottenere supporto immediato a qualsiasi ora, anche al di fuori degli orari di lavoro tradizionali. Questa disponibilità costante non solo migliora l'esperienza del cliente, ma crea un senso di affidabilità verso il servizio.

In termini di efficienza, un chatbot è in grado di gestire simultaneamente un gran numero di conversazioni, a differenza degli operatori umani che possono assistere solo un cliente alla volta. Ciò consente di scalare il servizio senza dover aumentare proporzionalmente le risorse umane, mantenendo comunque una reattività immediata. L'esperienza utente, quindi, risulta non solo più fluida, ma anche più soddisfacente, soprattutto quando il chatbot è in grado di fornire risposte personalizzate basate sulle preferenze o sulla cronologia dell'utente.

Un ulteriore vantaggio è rappresentato dalla capacità di raccogliere e analizzare i dati delle interazioni. Ogni conversazione con il chatbot genera informazioni che possono essere utilizzate per comprendere meglio le esigenze e i comportamenti dei clienti, consentendo all'azienda di prendere decisioni più informate e di migliorare continuamente il servizio offerto.

Infine, un chatbot accelera i tempi di risposta, eliminando le attese tipiche del supporto umano e migliorando così la percezione del servizio. La velocità e la reattività delle interazioni non solo riducono la frustrazione del cliente, ma rafforzano l'immagine di un'azienda moderna e all'avanguardia, capace di rispondere prontamente alle esigenze del mercato.[1]

1.5 Implicazioni per il progetto

Lo scopo principale del progetto presentato in questa tesi è semplificare l'interazione dell'utente con un servizio software specificamente progettato per le carrozzerie. Il chatbot implementato ha l'obiettivo di fornire un'assistenza immediata e intuitiva ai carrozzieri, rispondendo a domande sulle funzionalità del software e guidandoli attraverso le numerose operazioni disponibili. Questo approccio è stato scelto per ridurre il tempo e lo sforzo necessari per comprendere come utilizzare al meglio il software, permettendo ai professionisti di concentrarsi maggiormente sui loro compiti quotidiani anziché sui dettagli operativi del sistema.

Capitolo 2

Modello pre-addestrato GPT

Il modello *Generative Pre-Trained Transformer* (GPT) ha rivoluzionato il campo dell'elaborazione del linguaggio naturale,grazie alla capacità di generare testo coerente e contestualmente rilevante, basandosi su una grande quantità di dati pre-addestrati.

2.1 Cos'è GPT

GPT è un modello di deep learning pre-addestrato su grandi quantità di dati testuali, ed è in grado di svolgere numerose attività legate all'elaborazione del linguaggio naturale (NLP). Uno degli aspetti più significativi di GPT è la sua capacità di comprensione del linguaggio naturale (NLU). Grazie alla sua sofisticata architettura, GPT è in grado di cogliere il significato del testo e di riconoscere le relazioni tra le parole, creando un contesto coerente.

Oltre alla comprensione, GPT eccelle anche nella generazione di testo in modo creativo e nell'elaborare risposte efficaci alle domande che gli vengono poste. Non si limita solo alla generazione di testo: GPT è anche addestrato a produrre codice in diversi linguaggi di programmazione, come Python, Java, C e altri.

Nel complesso, la versatilità di GPT, unita alla sua elevata accuratezza nelle attività di NLP, lo rende uno strumento prezioso e applicabile a numerosi settori professionali e contesti del mondo reale.[9]

2.2 Architettura del transformer GPT

L'architettura Transformer, introdotta da Vaswani et al. nel 2017 nel documento "Attention is All You Need" [8], rappresenta un'innovazione significativa nel campo delle reti neurali, in particolare per le applicazioni del linguaggio naturale come la traduzione automatica. Questo modello ha rapidamente guadagnato popolarità grazie alla sua efficienza e alle sue prestazione superiori rispetto ai modelli tradizionali, come le reti neurali (RNN) o quelle convoluzionali (CNN), soprattutto nella capacità di catturare le relazioni a "lungo raggio" tra le parole.

Una delle caratteristiche principali che distingue il Transformer è il suo utilizzo del meccanismo di *self-attention*. Questo meccanismo permette al modello di considerare, in modo dinamico, le relazione tra tutte le parole di una sequenza, indipendentemente dalla distanza tra di loro. Grazie a questo il Transformer è particolarmente efficace nel catturare le informazioni contestuali, rendendolo ideale per compiti come la traduzione, dove il significato delle parole può dipendere da parole lontane nel testo.

L'architettura del Transformer, mostrata i figura 2.1, è composta da due componenti fondamentali: l'encoder e il decoder. Entrambi sono costituiti da stack multistrato di reti neurali, progettati per elaborare e trasformare i dati in modo progressivo. Gli encoder prendono in input la sequenza e ne estraggono rappresentazioni utili, mentre i decoder utilizzano tali rappresentazioni per generare la sequenza di output, come una traduzione.

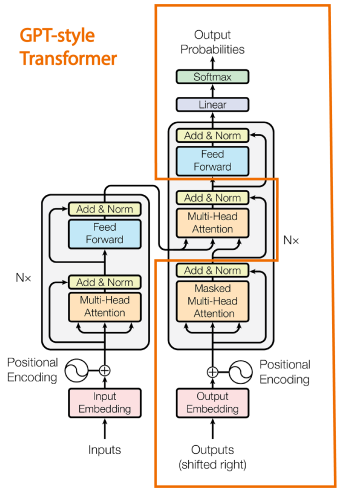

Figura 2.1: Architettura del Transformer.

Il Transformer è altamente configurabile grazie alla sua struttura, in quanto sono presente iperparametri che possono essere ottimizzati per migliorare le prestazioni del modello. Tra questi, vi sono il numero di livelli negli stack di encoder e decoder, la dimensione dei livelli nascosti e il numero di teste di attenzione utilizzate nel meccanismo di self-attention. Questi parametri possono essere adattati per ottene un equilibrio ottimale tra precisione, velocità di elaborazione e capacità di generazione.[3]

2.2.1 Self Attention

L'aspetto innovativo di questo modello è il concetto di *Self-Attention*, che si collega strettamente al meccanismo di Attention. Mentre Attention è una funzione che mappa una query e un insieme di coppie chiave-valore a un output, dove query, chiavi, valori e output sono rappresentati come vettori, *Self-Attention* è una forma specifica in cui query, chiavi e valori provengono tutti dalla stessa sequenza.

In questo meccanismo, ogni token della sequenza interagisce con gli altri token per calcolare l'output. Questo output è ottenuto attraverso una somma ponderata dei valori, con i pesi determinati da una funzione di compatibilità che valuta quanto la query sia "compatibile" con ciascuna chiave. Questo processo viene realizzato tramite un prodotto scalare o una funzione simile, chiamata "similarità". Se la query e la chiave risultano simili (cioè, il loro prodotto scalare è elevato), il valore associato alla chiave avrà un'importanza maggiore nell'output; viceversa, se la similarità è bassa, il valore avrà un peso ridotto.

Multi-Head Attention

L'altra innovazione fondamentale del modello Transformer è rappresentata dal concetto di *multi-head*. Il meccanismo di attenzione multi-testa si basa sul principio dello Scaled Dot-Product Attention, combinando l'idea di calcolare l'attenzione su più proiezioni delle query, chiavi e valori per ottenere un output più ricco e informativo. In particolare, le query, le chiavi e i valori vengono proiettati in più spazi di rappresentazione, ciascuno con dimensione differenti. Questo processo avviene molte volte, utilizzando diverse proiezioni apprese durante l'addestramento.

Dopo questa proiezione, le diverse versioni di query, chiavi e valori vengono elaborate in parallelo attraverso la funzione di attenzione. Questo significa che il modello può considerare simultaneamente molteplici aspetti delle informazioni, il che porta a un output più ricco e in-

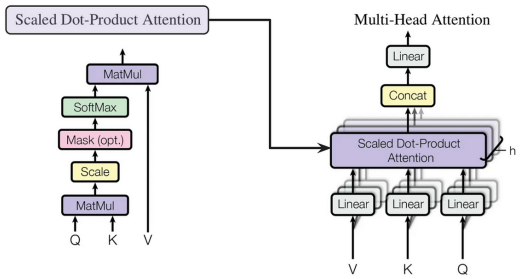

Figura 2.2: Scaled Dot-Product Attention (left) and Multi-Head Attention (right).

formativo. I risultati ottenuti da queste elaborazioni parallele vengono poi combinati insieme per formare il risultato finale.

Il grande vantaggio di questo tipo di approccio è che consente al modello di prestare attenzione a diverse informazioni in posizioni diverse contemporaneamente. Se si utilizzasse un un'unica testa di attenzione, il modello potrebbe perdere dettagli importanti, poiché le informazioni verrebbero mediate. Al contrario, l'attenzione multi-testa permette di raccogliere e analizzare informazioni da più angolazioni, migliorando notevolmente la capacità del modello di comprendere relazioni complesse nei dati. [8]

2.2.2 Encoder e Decoder

Come menzionato precedentemente, l'encoder e il decoder sono due parti fondamentali dell'architettura del Transformer.

L'*encoder* ha il compito di elaborare l'input e creare rappresentazioni ricche di significato. È composto da uno stack di sei livelli identici, ognuno dei quali contiene due sotto-livelli: un livello di *feed-forward* posizionale e un livello di *multi-head self-attention*.

Il livello *multi-head self-attention* consente al modello di analizzare diverse "prospettive" o relazioni all'interno dei dati simultaneamente. Questo migliora la capacità del modello di catturare dipendenze a lungo e breve termine. D'altra parte, il livello *feed-forward* è un livello in cui i dati fluiscono in una sola direzione, dall'input verso l'output, senza cicli o retroazioni. Questo livello introduce non linearità, consentendo al modello di identificare schemi e relazioni complesse nei dati.

Il *decoder*, invece, ha il compito di generare l'output basandosi sulla rappresentazione creata

dall'encoder. Anche lo stack del decoder è composto da sei livelli, ma contiene un ulteriore sotto-livello: il masked multi-head self-attention, come mostrato in figura 2.1. Questo livello impedisce al decoder di considerare le posizioni future nella sequenza durante l'addestramento, mentre il multi-head attention standard permette al modello di focalizzarsi su diverse parti dell'output dell'encoder.

Il livello feed-forward posizionale nel decoder è simile a quello dell'encoder, e ogni sotto-livello è seguito da uno strato di normalizzazione.

In sintesi, il decoder utilizza il masked multi-head self-attention per evitare che una determinata posizione di output incorpori informazioni su posizioni future durante l'addestramento, garantendo così l'integrità del processo.[3]

2.2.3 Positional Encoding

I Transformer GPT, a differenza delle reti reti neurali ricorrenti (RNN), non elaborano i dati in modo sequenziale. Non hanno quindi un meccanismo intrinseco per riconoscere la posizione delle parole all'interno della sequenza.

Positional Encoding è un metodo che consente ai modelli di codificare l'informazione posizionale delle parole all'interno delle sequenze di input. Questa mancanza viene compensata dall'aggiunta di embedding posizionali, vettori che rappresentano la posizione delle parole all'interno di una sequenza di input, come una frase o un paragrafo. Aggiungendo queste informazioni, i Transformer riescono a catturare la natura sequenziale dei dati, permettendo al modello di elaborare e comprendere efficacemente le relazioni all'interno della sequenza di input.[3]

Esistono tre metodi principali per calcolare gli embedding posizionali[4]:

- *Embedding appresi*: vengono appresi durante il processo di addestramento, analogamente a come il modello apprende i pesi per altre parti della rete. In questo caso il modello genera vettori posizionali ottimizzati per la specifica architettura e il dataset.
- *Embedding sinusoidali*: utilizzano funzioni seno e coseno con diverse frequenze per generare vettori posizionali. La posizione di ogni token viene trasformata in un vettore mediante queste funzioni, che sono costanti e non vengono apprese.
- *Embedding ALiBi* (Attention With Linear Biases): introducono bias lineari che rappresentano le distanze relative tra i token anziché la loro posizione assoluta. Viene aggiunto

un bias negativo ai punteggi di attenzione all'interno del meccanismo di self-attention, in modo che il modello possa tener conto delle distanze relative tra i token in una sequenza.

2.3 Processo di addestramento di GPT

L'addestramento dei modelli linguistici di grandi dimensioni può essere suddiviso in tre fasi principali: la raccolta e la preparazione dei dati, il pre-addestramento (che include la definizione dell'architettura del modello e l'utilizzo di algoritmi di training parallelo), e infine il fine-tuning e l'allineamento del modello per compiti specifici. In questa sezione verranno presentate le tecniche di addestramento, i dataset utilizzati e le metodologie comuni per la valutazione dei modelli.

2.3.1 Preparazione dei dati

L'addestramento dei modelli linguistici di grandi dimensione, come GPT, richiede enormi quantità di dati testuali, e la qualità di questi dati influisce notevolemente sulle prestazioni del modello. Il pre-addestramento su grandi quantità di dati offre una comprensione di base del linguaggio e alcune capacità generative. Le fonti utilizzate per il pre-addestramento sono varie, comprendendo spesso testi web, dati conversazionali e libri. Alcuni modelli vengono addestrati anche su dati specializzati provenienti da ambiti professionali, come il codice o i dati scientifici, per migliorare la capacità del modello in quei settori.

LLMs	Datasets
GPT-3 [8]	CommonCrawl [67], WebText2 [8], Books1 [8], Books2 [8], Wikipedia [75]

Figura 2.3: Datasets utilizzati per addestrare il modello GPT-3

Una volta raccolto un adeguato corpus di dati, il passo successivo è i *pre-processamento* di questi dati. La qualità di questo passaggio influisce nettamente sulle prestazioni del modello. I passaggi specifici di questa fase includono *il filtraggio di testi di bassa-qualità* ovvero l'eliminazione di contenuti "tossici" e distorti che comprometterebbero l'addestramento del modello, l'*eliminazione di contenuti duplicati* e l'*esclusione di contenuti ridondanti*

Esistono vari metodi per eseguire il filtraggio dei contenuti di bassa qualità come *euristiche* o *classificatori*. I metodi *euristici* utilizzano regole definite manualmente, ad esempio regole che

prevedono che siano mantenuti solo testi che contengono cifre o che vengano eliminati tutti i testi scritti solo con lettere maiuscole. I *classificatori*, invece, addestrano un classificatore su un dataset di alta qualità per filtrare i dataset di bassa qualità.

Importante è anche il discorso relativo alla *privacy* durante la fase di pre-processamento. È fondamentale infatti rimuovere sistematicamente tutti i dati sensibili che implicano tecniche di *anonimizzazione* o *tokenizzazione* per eliminare dettagli identificabili, geo localizzazioni o informazioni personali.

Un altro aspetto fondamentale è garantire lo sviluppo di modelli linguistici equi e imparziali, attraverso tecniche di *analisi del sentiment*, *rilevamento dell'odio* e algoritmi di *identificazione del bias* [6]

2.3.2 Pre-addestramento

I modelli di linguaggio di grandi dimensioni apprendono rappresentazioni linguistiche attraverso un processo di *pre-addestramento*. Durante questa fase, i modelli utilizzano ampi corpus di dati, e vengono addestrati mediante metodi di apprendimento auto-supervisionato [1]. In questa fase viene eseguita una *modellizzazione del linguaggio*, ovvero il modello apprende come prevedere la parola successiva in una sequenza di testo, basandosi sul contesto fornito. Attraverso questo compito, il modello acquisisce la capacità di catturare informazioni relative al vocabolario, alla grammatica, alla semantica e alla struttura del testo.

Una volta completato il pre-addestramento, tutti i risultati ottenuti dalla modellizzazione, possono essere adattati per vari compiti di elaborazione del linguaggio naturale. L'obiettivo infatti di quest'ultima, è proprio quello di massimizzare la probabilità dei dati testuali.

Un altra modalità di pre-addestramento è quella di sostituire casualmente alcune porzioni del testo per poi far eseguire al modello metodi auto regressivi per recuperare i token sostituiti. [6]

[1] È un metodo di apprendimento automatico che utilizza i dati non etichettati per addestrare modelli, creando etichette artificiali dalle stesse osservazioni.

2.3.3 Addestramento

Addestramento parallelo

Il *training parallelo* è una strategia utilizzata per addestrare modelli di linguaggio su più GPU contemporaneamente, migliorando così l'efficienza e la velocità del processo. Durante questa fase, la comunicazione collettiva è fondamentale, poiché permette di sincronizzare i parametri del modello e i dati tra le GPU.

Il processo di *parallelismo dei dati* prevede l'uso di un server di parametri che memorizza i parametri del modello e un batch di dati. Ogni GPU riceve una porzione dei dati, esegue il forward e il backward propagation, calcolando i gradienti. Questi gradienti vengono poi aggregati e inviati al server di parametri per aggiornare il modello. Un approccio alternativo è il *parallelismo dei dati distribuito*, elimina il server di parametri e utilizza il metodo all-reduce per comunicare i gradienti direttamente tra le GPU, permettendo così aggiornamenti indipendenti dei modelli.

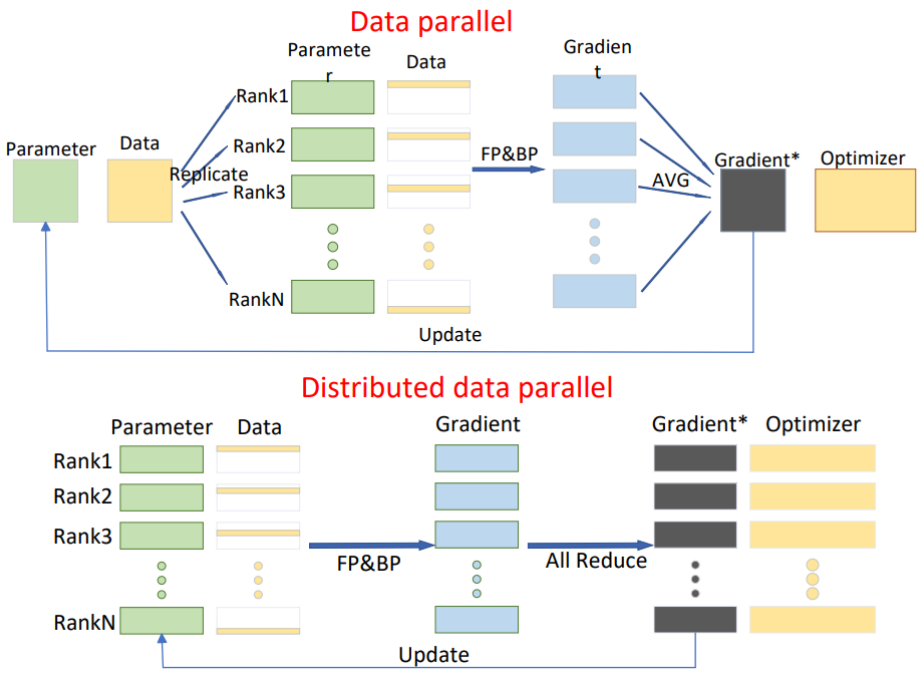

Figura 2.4: Parallelismo dei dati e parallelismo distribuito

In sintesi, il training parallelo ottimizza le risorse disponibili e accelera il processo di apprendimento del modello.[6]

Addestramento a precisione mista

L'*addestramento a precisione mista* è una strategia efficace per ottimizzare l'efficienza computazionale nell'addestramento di modelli di deep learning, in particolare nei modelli di linguaggio di grandi dimensioni. Questa tecnica combina numeri in virgola mobile a 16 bit (FP16) e a 32 bit (FP32) per massimizzare l'uso della memoria e velocizzare i calcoli.

Durante il processo di addestramento, i parametri del modello vengono inizialmente rappresentati in FP32, che offre una maggiore precisione e un intervallo numerico più ampio. Per migliorare la velocità computazionale, questi parametri vengono convertiti in FP16, il quale, pur essendo più veloce e meno ingombrante, presenta un intervallo numerico più ristretto.

Tuttavia, durante gli aggiornamenti dei parametri, il prodotto tra il gradiente e il tasso di apprendimento può risultare così piccolo da non rientrare nell'intervallo di FP16, causando un fenomeno noto come *underflow*, in cui i dati vengono persi. Per evitare questo problema, si rende necessario mantenere un parametro di precisione singola (FP32) all'interno dell'ottimizzatore.

Nella fase di addestramento, i parametri e i gradienti vengono gestiti in FP16, ma quando si tratta di aggiornare il modello, l'ottimizzatore utilizza FP32 per garantire la precisione. L'aggiornamento calcolato viene poi accumulato utilizzando un parametro temporaneo in FP32. Al termine di questo processo, i parametri aggiornati vengono riconvertiti in FP16 per essere utilizzati nel ciclo di addestramento successivo. [6]

CheckPoint

Per supportare la propagazione all'indietro in un modello di apprendimento profondo, è necessario salvare i risultati intermedi nella memoria della GPU durante la propagazione in avanti. Tuttavia, per ottimizzare l'uso della memoria, si utilizza un meccanismo di checkpoint che conserva solo determinati punti di controllo, piuttosto che tutti i risultati intermedi.

In un modello di tipo trasformatore, ogni blocco elabora l'input tramite processi complessi come l'attenzione e la rete feed-forward, producendo l'output del layer. Solo gli input di ciascun layer principale vengono mantenuti come checkpoint. Durante la propagazione all'indietro, si utilizza una tecnica chiamata recomputation, che consiste nel rieseguire la propagazione in avanti per calcolare i gradienti. I risultati intermedi ottenuti possono essere utilizzati per la propagazione all'indietro. Dopo il completamento della propagazione all'indietro, è possibile scartare sia il checkpoint che i risultati intermedi temporaneamente ricomputati dalla memoria della GPU. [6]

2.3.4 Tecniche di fine-tuning

Il fine-tuning di GPT si riferisce al processo di adattamento di un modello di linguaggio pre-addestrato su un nuovo set di dati specifico per compiti o applicazioni particolari, migliorando così le sue performance in contesti mirati. Questa fase consente di affinare le capacità del modello in modo che possa generare risposte più rilevanti e accurate in base alle esigenze specifiche degli utenti o delle applicazioni. [6]

Fine-tuning supervisionato

Il fine-tuning supervisionato (SFT) è un metodo per migliorare un modello di linguaggio già addestrato, rendendolo più adatto a svolgere compiti specifici. Questo processo richiede un dataset etichettato per il compito specifico da svolgere, che consiste in coppie di testi di input e le loro risposte corrette, permettendo così al modello di apprendere e adattarsi meglio alle richieste specifiche.

Una tecnica popolare nel fine-tuning è l'istruzione *tuning*, in cui il modello viene ulteriormente addestrato su un insieme di dati che include istruzioni e le relative risposte. Questa forma di addestramento aiuta il modello a comprendere e seguire le istruzioni umane in modo più efficace, migliorando le sue prestazioni e la sua controllabilità. I dataset utilizzati per questo tipo di addestramento sono vari e sono stati raccolti per aiutare i ricercatori a sviluppare modelli di linguaggio più reattivi e precisi. [6]

Alignment tuning

L'alignment tuning è una tecnica di fine-tuning che mira a garantire che le informazioni prodotte non si discostino dall'intento umano, come informazioni false o contenuti fuorvianti. [6]

In particolare mira a garantire che le informazioni siano *utili*, ovvero che le risposte generate siano realmente benefiche per l'utente e che forniscano informazioni che soddisfino i requisiti specifici del compito, *oneste* in quanto il modello deve generare output autentici e affidabili, evitando la fabbricazione di fatti e *innoque* perché il contenuto generato non deve essere dannoso o offensivo, garantendo la sicurezza degli utenti.

Parameter-efficient tuning

I modelli di linguaggio di grandi dimensioni (LLM) come Chat GPT continuano a crescere in scala, ma il fine-tuning completo su hardware consumer è diventato costoso e impraticabile per molti ricercatori. A differenza del fine-tuning supervisionato e dell'alignment tuning, il *parameter-efficient tuning* mira a ridurre l'overhead computazionale e di memoria, consentendo di adattare solo un sottoinsieme ridotto di parametri del modello, mantenendo la maggior parte dei parametri pre-addestrati fissi. Questo approccio abbassa significativamente i costi di calcolo e archiviazione, permettendo di ottenere prestazioni simili a quelle del fine-tuning completo. [6]

Fine-Tuning per la sicurezza (SFT)

Per migliorare la sicurezza e la responsabilità dei modelli di linguaggio di grandi dimensioni (LLMs), è fondamentale integrare tecniche di sicurezza aggiuntive durante il processo di fine-tuning. [6] Alcune tecniche principali sono:

- **Supervised Safety Fine-Tuning**: prevede che gli etichettatori generino dati dimostrativi contenenti input avversari ad alto rischio di sicurezza. Questi dati vengono poi integrati nella fase di SFT, migliorando così la capacità del modello di gestire i rischi legati alla sicurezza.
- **Safety Reinforcement Learning with Human Feedback**: si utilizzano prompt avversari, anche più aggressivi, per interrogare i modelli. La risposta più sicura, che mostra un comportamento di rifiuto, viene utilizzata per addestrare un modello per la sicurezza all'interno del framework RLHF.
- **Safety Context Distillation**: utilizza la distillazione del contesto, aggiungendo inizialmente pre-prompt di sicurezza, come "Sei un assistente sicuro e responsabile," a prompt avversari. Questo processo produce risposte generate più sicure. Successivamente, il modello viene fine-tuned su questi dati dimostrativi sicuri, ma senza includere i pre-prompt di sicurezza, migliorando ulteriormente le capacità di sicurezza del modello.

Le caratteristiche del modello GPT precedentemente descritte, lo rendono particolarmente utile per la creazione di assistenti virtuali efficienti e flessibili, che rappresentano due qualità fondamentali per il progetto sviluppato per questa tesi. L'obiettivo è garantire un funzionamento ottimale dell'assistente e una facile addestrabilità, facilitando così eventuali sviluppi futuri.

Capitolo 3

Il progetto

3.1 Introduzione al progetto

Nel contesto attuale di crescente digitalizzazione, è sempre più comune la ricerca di modalità che possano rendere l'esperienza utente più semplice. Sono molti i software e i siti web che dispongono di numerose funzionalità implementate, delle quali diventa difficile sfruttarne appieno le potenzialità. Gli utenti spesso si trovano a dover navigare attraverso interfacce complesse o documentazioni estese solo per ottenere informazioni su funzionalità specifiche. L'utente che si approccia a uno di questi software può trovarsi in difficoltà a causa della vasta gamma di funzionalità offerte, al punto da dover compiere uno sforzo maggiore del necessario nell'utilizzo del servizio implementato dagli sviluppatori, spesso sviluppato proprio per semplificare le attività dell'utente. Per queste ragioni, la tendenza a sviluppare soluzioni avanzate per migliorare l'efficienza e la qualità delle interazioni digitali è sempre più diffusa tra gli sviluppatori. In questo contesto, l'obiettivo del presente progetto è stato proprio quello di semplificare l'esperienza utente nel software dedicato alle carrozzerie, integrando un chatbot capace di assistere gli utenti in modo intuitivo e immediato.

Il progetto presentato in questa tesi si propone di sviluppare un chatbot avanzato utilizzando le API di OpenAI, integrandolo in un'applicazione realizzata con C# e dotata di un'interfaccia utente interattiva sviluppata in ASP.NET. L'obiettivo principale è quello di creare un sistema capace di gestire conversazioni in modo naturale, rispondendo a domande relative alle funzionalità offerte dal software. Attraverso l'integrazione di tecnologie moderne come SignalR, il

chatbot sarà in grado di fornire un'esperienza utente fluida e intuitiva, rendendo le interazioni più efficienti e personalizzate.

Il capitolo proseguirà con una descrizione dettagliata sugli aspetti implementativi del progetto.[7]

3.2 Strumenti e linguaggi di programmazione

In questa sezione verranno descritti i principali strumenti software, i linguaggi di programmazione e le librerie utilizzati per lo sviluppo del progetto.

3.2.1 Strumenti software

Per il progetto, è stato utilizzato Visual Studio. Questo IDE è stato scelto per la sua robustezza e le sue capacità avanzate di debugging e integrazione con altre tecnologie Microsoft, come ASP.NET e SignalR.

3.2.2 Linguaggi di programmazione

Il progetto è stato sviluppato utilizzando la programmazione a oggetti nel linguaggio di programmazione C#.

Per collegare il meccanismo client-server all'interfaccia utente è stato utilizzata una libreria JavaScript, jQuery, che semplifica la manipolazione del DOM, la gestione degli eventi e le chiamate AJAX.

3.2.3 Librerie utilizzate

Nel progetto sono state utilizzate diverse librerie per implementare funzionalità avanzate e migliorare l'efficienza dello sviluppo.

SignalR è una libreria per ASP.NET che facilita l'implementazione della comunicazione in tempo reale tra il client e il server. Questa funzionalità è fondamentale per aggiornare dinamicamente l'interfaccia utente in risposta agli eventi del server.

Kendo UI è una libreria di componenti UI per lo sviluppo di applicazioni web. È stata utilizzata per creare un'interfaccia utente interattiva, fornendo componenti predefiniti come form

per la chat. L'uso di Kendo UI ha permesso di accelerare lo sviluppo dell'interfaccia utente notevolmente rispetto all'utilizzo di *html* e *css*.

3.3 Oggetti implementati

Per facilitare la comprensione del progetto, è stato scelto di creare oggetti distinti per i principali componenti del sistema, anziché integrare tutte le funzionalità in un unico blocco. Questo approccio consente di strutturare il progetto in maniera più ordinata, migliorando la chiarezza dell'architettura complessiva e agevolando sia la manutenzione che l'estensione del codice. Qui di seguito, sono elencati i principali tre oggetti necessari per la creazione dell'assistente virtuale.

3.3.1 Assistant

```
public Assistant(string name)
{
    this.instructions = "You are a customer support chatbot. Use your
       knowledge base to best respond to customer queries.";
    this.name = name;
    this.model = "gpt-3.5-turbo-0125";
    this.tools = new[]
    {   new Tool("retrieval"),
        new ToolPlus("function", Function.Info),
    };
    this.file_ids = new string[] { "file-c5pbSSX6f9Y6i3ozqIfDZHSm","file-
       lO375yfqU3aZtw37bCsCVLfs" };
}
```

Figura 3.1: Costruttore della classe Assistant

Le principali caratteristiche dell'oggetto *Assistant* sono l'attributo *model* e *tools*.

Il modello GPT utilizzato è *gpt_3.5_turbo* in quanto è stato progettato per essere più veloce ed efficiente rispetto ai suoi predecessori, pur mantenendo un alto livello di capacità di comprensione e generazione del linguaggio naturale.

L'attributo *tools* è un array che include le due funzionalità dell'assistente:

- **Retrieval**: Essendo i modelli GPT per definizione dei modelli pre-addestrati al solo scopo di elaborare linguaggio naturale, con la funzionalità *retrieval* è possibile aggiungere un livello di conoscenza specifica in base allo scopo dell'assistente nel software in cui viene integrato. Nel contesto del progetto, questa conoscenza è data da file [1] contenti manuali di istruzioni relative alle funzionalità del servizio Oxygen. Grazie a questa funzionalità, l'assistente potrà rispondere a domande specifiche riguardanti il software (se tali informazioni sono presenti nei file di testo caricati).

- **Function**: La funzionalità *function* consente all'Assistant di riconoscere pattern specifici nelle domande poste dagli utenti e di eseguire azioni o funzioni specifiche in risposta. La funzione implementata per il progetto, viene "attivata" dall'assistente quando riconosce nella domanda, che gli vengono richieste informazioni relative a un *Automobile* presente in una lista di veicoli definiti.

  Tra gli attributi richiesti dalla funzione per poter operare correttamente, l'attributo *Parameters* specifica i parametri necessari per l'esecuzione della funzione. I due parametri passati alla funzione sono l'oggetto *Targa* e l'oggetto *Proprietario*. Quando vengono effettuate domande di questo tipo:

Figura 3.2: Domanda tipo 1

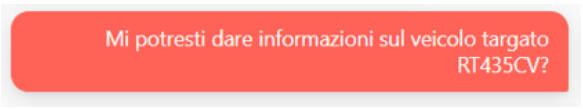

Figura 3.3: Domanda tipo 2

  la funzione riconosce il proprietario *Federica Santisi* e la targa *RT435CV*, e utilizza i due valori per eseguire una query specifica all'interno della lista di automobile definita.

[1] Elencati nell'attributo dell'oggetto *this.file_ids*

```
public static Function Info = new Function
{
    name = "recuperaInformazioni",
    description = "Se ti chiedo le informazioni sull'auto mi devi rispondere
        con le informazioni di tua conoscenza",
    parameters = new Parameters(
        "object",
        new Properties(
            new Targa(
                "string",
                "La targa di un auto e' una sequenza di 2 lettere, 3 numeri
                   e di nuovo 2 lettere"
                ),
            new Proprietario(
                "string",
                "Il proprietario del veicolo e' il nome della persona che
                   possiede il veicolo"
                )
        ),
        new string[] { "targa", "proprietario" }
    )
};
```

Figura 3.4: Function *"recuperaInformazioni"*

3.3.2 Thread

```
public Thread(string assistant_id)
{
    this.assistant_id = assistant_id;
}
```

Figura 3.5: Costruttore della classe Thread

L'oggetto thread rappresenta un'unità di esecuzione all'interno del programma, consentendo eventualmente l'esecuzione simultanea di più attività. Nell'ambito della creazione di assistenti intelligenti, i thread sono utilizzati per gestire richieste multiple in parallelo. Questo è partico-

larmente utile quando si inviano richieste a un API come quella di OpenAI, dove la latenza di rete può influenzare il tempo di risposta.

L'oggetto *Thread* è stato progettato per iniziare e gestire una conversazione con l'assistente. Questo oggetto contiene un unico attributo, *assistant_id*, che rappresenta l'ID dell'assistente con cui ha appena iniziato la conversazione.

3.3.3 Run

```
public Run(string assistant_id)
{
    this.assistant_id = assistant_id;
}
```

Figura 3.6: Costruttore della classe Run

L'oggetto *run* è stato progettato per rappresentare una singola esecuzione di una domanda posta all'assistente. Per questo motivo l'unico attributo necessario all'oggetto *run* è l'*assistant_id*, ovvero l'ID dell'assistente a cui è stata sottoposta la domanda.

L'oggetto *run*, durante il processo di esecuzione della domanda e generazione della risposta, assume diversi "stati". Gli stati sono dati da un'enumerazione definita all'interno della classe *RunStatusEnum* contenente i seguenti valori:

```
internal enum RunStatusEnum
{
    queued,
    in_progress,
    requires_action,
    cancelling,
    cancelled,
    failed,
    completed,
    expired,
}
```

Figura 3.7: Enumerazione per gli stati del Run

Ogni stato del run ha un significato associato, che permette di monitorare il progresso dell'esecuzione e di gestire eventuali errori o problemi che possono sorgere durante l'interazione con l'assistente.

Gli status *queued* o *in_progress* indicano che la domanda è stata ricevuta e che è in attesa che l'assistente possa eseguirla. I tempi di attesa della riposta dipendono da diversi fattori, tra cui la complessità della domanda (a livello di computazione o di elaborazione della conoscenza richiesta per rispondere), oppure dall'esecuzione di più thread in parallelo.

Gli stati *cancelling*, *cancelled*, *failed* ed *expired* indicano che si è verificato un problema durante l'esecuzione della domanda, causando l'interruzione del processo. Al contrario, lo stato *completed* indica che la domanda è stata eseguita con successo e che è stata generata una risposta.

Lo stato *requires_action* indica che nella domanda sono stati riconosciuti i pattern che attivano l'esecuzione della funzione sopra citata.

3.3.4 Altri oggetti implementati

Tool

L'oggetto *Tool*, caratterizzato dall'attributo *type*, è stato utilizzato nella classe *Assistant* per specificare il tipo di strumento di cui l'assistente avrebbe dovuto disporre. Nel contesto del progetto, questo attributo è stato impostato su *retrieval*, indicando che l'assistente è dotato di funzionalità di recupero delle informazioni.

ToolPlus

La classe *ToolPlus* è stata implementata come estensione della classe *Tool* per consentire l'utilizzo del tool *function*. Oltre all'attributo *type*, questo tool richiede la dichiarazione della funzione che deve svolgere. Nel contesto del nostro progetto, tale funzione è rappresentata dall'oggetto *Function* implementato.

Parameters

La classe *Parameters* include l'attributo *type*, che nella definizione della funzione sopra menzionata in figura 3.4 è impostato su "object". Inoltre, essa comprende l'attributo *properties*, il

quale è a sua volta una classe che specifica le proprietà o i campi che l'oggetto deve contenere. L'attributo *required*, invece, elenca le informazioni necessarie affinché l'assistente riconosca la necessità di applicare la funzione per quella determinata domanda.

Properties

Nel contesto del progetto, la classe Properties contiene solamente la proprietà *targa* che a sua volta è identificata dalla classe *Targa*

Targa

La classe *Targa* include gli attributi *type* e *description*, definiti per convenzione al fine di rappresentare la targa di un veicolo.

Automobile

```
public static Automobile auto1 = new Automobile
{
    proprietario = "Federica Santisi",
    targa = "BO142YZ",
    modello = "Mercedes",
    marca = "Classe A",
    descrizione_del_danno = "Si e' rotto il paraurti",
};
```

Figura 3.8: Istanza della classe Automobile

Questa rappresenta una delle definizioni di automobili fornite. Quando l'assistente identifica che la domanda corrisponde all'azione della funzione *recuperaInformazioni*, procede ad estrarre il parametro *targa* o *proprietario* dalla domanda, verifica a quale delle automobili definite corrisponde tale parametro e restituisce le informazioni pertinenti relative a quell'automobile.

Question

La classe *Question* possiede l'attributo *role*, che rappresenta l'entità da cui proviene la domanda, ovvero "user", e l'attributo *content*, che indica il contenuto della domanda stessa, ossia ciò che viene scritto dall'utente.

Tool Output, Tool Output Elements

La classe *ToolOutput* possiede un unico attributo, *tool_output*, che contiene tutti gli elementi *ToolOutputElements*, ovvero gli output generati dall'esecuzione della funzione.

L'oggetto *ToolOutputElement* contiene l'identificativo *tool_id* dell'azione riconosciuta come da eseguire con il tool *function* e l'attributo *output*, che rappresenta l'output effettivo della funzione eseguita.

3.4 Componenti del client

Le API (Application Programming Interface) sono state fondamentali per l'implementazione del progetto, consentendo l'interazione con servizi esterni e l'integrazione di funzionalità avanzate. Le API utilizzate per il progetto sono quelle fornite dalla piattaforma di OpenAI [5.2]. La scelta di utilizzare le API di OpenAI è stata motivata dalla loro capacità di fornire modelli di intelligenza artificiale avanzati, come GPT-3.5-turbo, che offrono prestazioni elevate in termini di comprensione e generazione del linguaggio naturale. Per comodità del progetto, tutte le API sono state inserite all'interno di un file chiamato *openAIApiClient*. Questo file centralizza tutte le chiamate API, facilitando la gestione e la manutenzione del codice.

3.4.1 Configurazione del client HTTP

```
1 public HttpClient createRequest()
2 {
3     _httpClient = new HttpClient();
4     _httpClient.DefaultRequestHeaders.Authorization = new System.Net.Http.
          Headers.AuthenticationHeaderValue("Bearer", this.settings.apiKey);
5     _httpClient.DefaultRequestHeaders.Add("OpenAI-Beta", this.settings.
          version);
6     _httpClient.DefaultRequestHeaders.Accept.Add(new System.Net.Http.Headers
          .MediaTypeWithQualityHeaderValue("application/json"));
7     return _httpClient;
8 }
```

Figura 3.9: Metodo createRequest()

Il metodo *createRequest()* crea e configura un client HTTP, pronto per inviare richieste a un'API remota per accedere a servizi come OpenAI. Quando il metodo *createRequest()* viene chiamato, si crea un'istanza di HttpClient, che è uno strumento utilizzato per gestire la comunicazione via HTTP. Successivamente, vengono aggiunte diverse intestazioni alla richiesta: una per l'autenticazione, utilizzando un token API nel formato Bearer, e un'altra intestazione personalizzata che indica una versione specifica dell'API in uso. Infine, il client viene configurato per accettare risposte in formato JSON, garantendo che i dati ricevuti dal server siano strutturati in un formato standard e facilmente elaborabile. Una volta configurato, questo client HTTP è pronto per inviare richieste al server e ricevere risposte, facilitando l'integrazione con servizi esterni.

3.4.2 Metodi preliminari

I tre metodi descritti successivamente (*createContent()*, *PostAsync()*, *GetAsync()*) sono stati implementati come metodi preliminari per facilitare l'esecuzione delle chiamate API all'interno del progetto. Questi metodi sono stati progettati per standardizzare e semplificare il processo di invio e ricezione delle richieste HTTP, evitando la necessità di riscrivere ripetutamente lo stesso codice per ogni chiamata API. In questo modo, è possibile richiamare semplicemente questi metodi per eseguire le operazioni desiderate, migliorando l'efficienza e la manutenibilità del codice.

- **Metodo createContent(object data):**

  Il metodo *createContent()* è utilizzato per creare il contenuto della richiesta POST verso il server. Questo processo inizia con la conversione dell'oggetto passato come parametro in una stringa formattata JSON, tramite il metodo *JsonConvert.SerializeObject()*. Una volta ottenuta questa stringa, essa viene incapsulata in un oggetto *StringContent*, che rappresenta il contenuto della richiesta HTTP. Questo oggetto è codificato in UTF-8 e contrassegnato con il tipo MIME "application/json", assicurando che il server possa interpretare correttamente i dati ricevuti.

- **Metodo PostAsync(string requestUri, StringContent content):**

  Il metodo *PostAsync* è progettato per inviare richieste POST al server in modo asincrono, permettendo all'applicazione di non bloccarsi mentre attende la risposta. Questo metodo prende come parametri l'*URL* della richiesta e il contenuto da inviare, creato in precedenza dal metodo *createContent()*. Una volta inviata la richiesta, il metodo si occupa

di ricevere la risposta dal server, leggendo il contenuto della risposta come una stringa JSON. Successivamente, la stringa viene convertita in un oggetto JSON (JObject) tramite il metodo *JsonConvert.DeserializeObject()*, in modo da facilitare la manipolazione dei dati all'interno dell'applicazione. Infine, il metodo restituisce l'oggetto JSON risultante, permettendo così di accedere ai dati restituiti dal server.

- **Metodo GetAsync(string requestUri):**

  Il metodo *GetAsync* gestisce le richieste GET verso il server, sempre in maniera asincrona per mantenere l'applicazione reattiva. Inizialmente, viene inviata la richiesta GET all'*URL* specificato come parametro, e il metodo attende la risposta dal server. Successivamente, il contenuto della risposta, inizialmente in formato JSON, viene de-serializzato in un oggetto JSON (JObject), che viene poi restituito per consentire un facile accesso ai dati ricevuti dal server.

3.4.3 Metodi principali

CheckStatusAndUpdateResponse(string threadId, string runId)

Il metodo *CheckStatusAndUpdateResponse* è progettato per monitorare lo stato di un "run" specifico all'interno di un thread, gestire diverse situazioni basate sullo stato del "run", e potenzialmente intraprendere azioni aggiuntive se richiesto. Restituisce true se il run viene completato con successo, altrimenti restituisce false.

Il metodo inizia chiamando il metodo *CheckStatus()*, che recupera lo stato attuale del "run" specificato tramite l'API. Questo stato è contenuto nell'oggetto *result* restituito dall'API. In base allo stato del run ottenuto, il metodo gestisce la prossima azione da eseguire.

- `"in_progress","queued"`:

  Se il *run* è in corso o in coda, il metodo attende per un breve intervallo di tempo, impostato a 10 secondi e poi richiama se stesso ricorsivamente per verificare nuovamente lo stato. Questo continua fino a quando lo stato non cambia.

```
if (result["status"].ToString() == RunStatusEnum.in_progress.ToString() ||
    result["status"].ToString() == RunStatusEnum.queued.ToString())
{
    TimeSpan retryInterval = TimeSpan.FromSeconds(10);
    await Task.Delay(retryInterval);
    return await CheckStatusAndUpdateResponse(threadId, runId);
}
```

Figura 3.10: Caso status "in_progress" o "queued"

- `"completed"`:

  Se il *run* è completato, il metodo restituisce *true*, segnalando che il processo è terminato con successo.

- `"cancelling","failed","expired"`:

  Se il *run* è in fase di annullamento, ha fallito o è scaduto, il metodo restituisce *false*, indicando che il processo non è riuscito a completarsi correttamente.

- `"requires_action`:

  Se il *run* richiede un'azione aggiuntiva, il metodo controlla se ci sono azioni specifiche da eseguire. Se esiste un'azione denominata *submit_tool_outputs*, che richiede l'invio di *tool_output* ovvero i risultati prodotti da operazioni specifiche eseguite durante il "run", il metodo itera su ogni *toolCall* contenuta in *requiredAction*. Per ciascuna di queste chiamate, il metodo chiama *handleRequiresAction()* per gestire l'azione richiesta, passando l'ID del thread di azione come parametro.

```
    foreach (JObject toolCall in toolCalls)
        {
            JToken actionThreadId = toolCall["id"];
            string action_threadId = actionThreadId.ToString();
            await this.handleRequiresAction(result, threadId, runId,
                action_threadId);
        }
```

Figura 3.11: Caso status "requires_action"

HandleRequiresAction(JObject result, string threadId, string runId, string action_threadId)

Il metodo *handleRequiresAction* è progettato per gestire azioni richieste durante l'esecuzione di un *run* specifico in un thread, in particolare quando l'azione richiesta riguarda l'invio di *toolOuput* (nel contesto del progetto sono informazioni su un automobile).

Il metodo esegue delle query sull'oggetto JSON restituito dall'API del metodo *CheckStatus()*, per ottenere l'array contenente i *toolCall*.

Per ogni *toolCall*, il metodo controlla se il nome della funzione contenuto è la funzione implementata *recuperaInformazioni*.

Se la funzione è quella cercata, il metodo estrae gli argomenti che contengono i dati necessari per eseguire l'azione, ovvero la targa e il proprietario dell'automobile.

Utilizzando la targa o il proprietario estratti, il metodo cerca l'automobile corrispondente all'interno di una lista (*ListaAutomobili*). Se trova l'automobile con quella targa o con quel proprietario, serializza le informazioni dell'automobile in formato JSON.

```
1 List<Automobile> listaAuto = Automobile.ListaAutomobili;
2
3 if(targa != null && targa != "" && targa != "non disponibile")
4 {
5     Automobile automobileCercata = listaAuto.Find(automobile => automobile.
          targa == targa);
6     string auto = JsonConvert.SerializeObject(automobileCercata);
7     targaOutput = new ToolOutputElement(action_threadId, auto);
8 }
```

Figura 3.12: Ricerca automobile per targa

Se la targa non è vuota o nulla, il metodo crea un oggetto *ToolOutputElement* che rappresenta il risultato da inviare tramite una richiesta POST all'endpoint:

(*"https://api.openai.com/v1/threads/threadId/runs/runId/submit_tool_outputs", content*)

RESPOND(string threadId)

Il metodo *RESPOND* ha lo scopo di recuperare l'ultima risposta generata dall'assistente all'ultima domanda inviata dall'utente in un thread specifico.

Il metodo inizia recuperando tutti i messaggi associati a un thread specifico chiamando il metodo *GetResponse(threadId)*, che restituisce un oggetto JSON contenente i dati del messaggio.

Successivamente, il metodo cerca l'ultimo messaggio inviato dall'utente nel thread. Questo viene fatto filtrando l'array di messaggi *result["data"]* per trovare l'elemento più recente in cui il "role" è *user*. Questa ricerca viene effettuata con il metodo *LastOrDefault*, che restituisce l'ultimo messaggio che soddisfa la condizione o *null* se non ne trova nessuno.

Se l'ultimo messaggio dell'utente viene trovato, il metodo cerca il primo messaggio di risposta dell'assistente che è stato creato dopo quel messaggio in cui il "role" è *assistant* e il timestamp di creazione del messaggio dell'assistente è successivo a quello del messaggio dell'utente.

```
if (lastUserMessage != null)
{
    var associatedResponse = result["data"].FirstOrDefault(msg =>
        msg["role"].ToString() == "assistant" &&
        msg["created_at"].ToObject<long>() > lastUserMessage["created_at"].
            ToObject<long>()
    );

    if (associatedResponse != null)
    {
        var responseText = associatedResponse["content"].FirstOrDefault()?["
            text"]?["value"].ToString();
        return responseText;
    }
}
```

Figura 3.13: Metodo RESPOND

Se viene trovata una risposta associata, il metodo estrae il testo della risposta e lo restituisce, altrimenti restituisce la stringa "Risposta non trovata!".

3.5 Componenti del server

L'architettura client-server adottata nel progetto ha richiesto l'implementazione di un hub, un controller e un singleton, che fungono da intermediari tra il client e il server. Questi componenti coordinano le richieste del client e gestiscono le risposte del server, ottimizzando la comunicazione e l'efficienza del sistema.

3.5.1 Singleton

Il design pattern Singleton è una soluzione utilizzata in programmazione per garantire che una classe abbia una sola istanza e fornisca un punto di accesso globale a essa.

Il ClientSingleton è progettato per centralizzare la gestione delle interazioni con le API OpenAI, garantendo che vi sia una sola istanza per ciascun elemento necessario alla conversazione con il chatbot.

```
1 public static OpenAIApiClient? _OpenAIApiClient;
2 public static OpenAIApiClient OpenAIApiClient
3 {
4     get
5     {
6         if (_OpenAIApiClient == null) _OpenAIApiClient = instanceClient();
7         return _OpenAIApiClient;
8     }
9 }
```

Figura 3.14: Istanziazione del Client

La variabile _OpenAIApiClient è definita come un campo privato statico all'interno della classe ClientSingleton. Questo campo rappresenta un'istanza del client OpenAI, che viene utilizzata per gestire le interazioni con le API di OpenAI. È importante notare che _OpenAIApiClient non è istanziato direttamente al momento della dichiarazione; invece, viene inizializzato all'interno del metodo OpenAIApiClient, il quale verifica se l'istanza è già presente e, in caso contrario, la crea tramite il metodo instanceClient.

```
public static async Task<string> instanceAssistant()
{
    if (assistantCreated == false)
    {
        Assistant assistant = new Assistant("Oxygen");
        JObject _result = await _OpenAIApiClient.CreateAssistant(assistant);
        assistantCreated = true;
        assistantId = _result["id"].ToString();
    }
    return assistantId;
}
```

Figura 3.15: Singleton Assistant

Un esempio della logica del Singleton è il metodo *instanceAssistant*. Questo metodo verifica se l'assistente è già stato creato. Se non lo è, procede a istanziato utilizzando il client OpenAI. Nello specifico, chiama il metodo CreateAssistant dell'istanza del client per creare un nuovo assistente, memorizzando il suo ID per un utilizzo successivo. Questo approccio è analogo a come gli altri metodi, come instanceThread e instanceRun, operano nel contesto della creazione di thread e run.

Anche l'invio, l'update del run e la ricezione della risposta sono gestite nel Singleton, sempre richiamando i metodi dell'istanza del client.

3.5.2 Hub

L'Hub SignalR funge da componente centrale per la gestione della comunicazione in tempo reale tra il client e il server. Implementato come una classe C# denominata *ChatHub*. L'Hub eredita dalla classe *Hub* di SignalR e contiene metodi che permettono ai client di inviare messaggi al server e viceversa.

ChatHub contiene due metodi principali: *askQuestion* e *retrieveHistory*.

askQuestion(QuestionElem question)

Il metodo principale dell'Hub è *askQuestion*, che viene invocato dall'interfaccia utente per inviare una domanda. Questo metodo integra e coordina tutti i metodi del client istanziati nel

Singleton per gestire l'intero processo, dalla creazione dell'assistant alle ricezione della risposta.

```
OpenAIThread? thread = null;
if (string.IsNullOrEmpty(question.threadId))
{ thread = (OpenAIThread?)Context.Items[THREADITEMSNAME];}
else
{ thread = new OpenAIThread(question.threadId);}

if (thread == null)
{ thread = await ClientSingleton.instanceThread();
    if (thread == null) throw new NotImplementedException();
    else
    { Context.Items.Add(THREADITEMSNAME, thread); }
}
```

Figura 3.16: Gestione dell'istanziazione del thread

Il codice gestisce l'assegnazione di un'istanza di OpenAIThread per la conversazione con l'assistente. Inizialmente, verifica se *question.threadId* è vuoto, ovvero se non è stato creato dal client. Se è così, cerca un thread esistente in Context.Items che contiene tutti i valori della sessione relativa alla conversazione. Se esiste, lo utilizza; altrimenti, crea un nuovo thread utilizzando ClientSingleton.instanceThread(). Se il thread non può essere creato, solleva un'eccezione altrimenti il valore dell'ID viene memorizzato in Context.Items per usi futuri. Questo processo assicura che ci sia sempre un thread attivo per gestire la conversazione. Recuperare il thread dalla sessione invece, permette all'assistente di mantenere il contesto della conversazione.

Questo tipo di ragionamento, nell'Hub viene adottato anche per l'*Assistant*. Per il *run*, invece, non è necessario perché come spiegato in precedenza, il run è un'"azione" eseguita per la singola domanda.

retrieveHistory(string threadId)

Questo metodo, è stato implementato per permettere all'*Assistant*, nel caso in cui fosse già stato istanziato, di recuperare tutti i "messaggi" relativi al *thread* salvato in sessione per quella conversazione.

```
JObject messages = await apiClient.retrieveMessages(threadID);
return JsonConvert.SerializeObject(messages);
```

Figura 3.17: Recupero messaggi dal thread in sessione

Questo metodo consente all'assistente di riprendere una conversazione, mantenendo il contesto e permettendo di porre domande relative a messaggi precedentemente inviati (come chat GPT).

3.5.3 Controller

Nel progetto sono presenti due controller: *OpenAIController* e *HomeController*.

OpenAIController

È responsabile della gestione delle operazioni relative ai thread di conversazione. Questo controller include metodi per impostare e recuperare i thread dalla sessione, utilizzando il contesto HTTP per memorizzare e recuperare i dati del thread. In particolare, il metodo *threadSet* consente di memorizzare un thread nella sessione.

```
[HttpPost]
public async Task<JsonResult> threadSet(OpenAIThread thread)
{
    if (HttpContext != null)
    { DataSession.setThread(HttpContext, thread); return Json(thread); }
    else return Json(null);
}
```

Figura 3.18: Metodo threadSet

Il metodo *threadGet* invece, permette di recuperare il thread corrente dalla sessione.

```
[HttpGet]
public JsonResult threadGet()
{
    if (HttpContext != null)
    { string threadID = DataSession.getThread(HttpContext); return Json(
       threadID); }
    else return Json(null);
}
```

Figura 3.19: Metodo threadGet

Questi metodi sono essenziali per mantenere lo stato della conversazione tra le diverse richieste dell'utente, garantendo una continuità nell'interazione con il chatbot.

HomeController

Gestisce le operazioni generali dell'applicazione, come la visualizzazione della pagina principale e della pagina della privacy. Questo controller include anche un metodo per gestire gli errori, restituendo una vista con le informazioni sull'errore corrente. Sebbene *HomeController* non interagisca direttamente con le funzionalità specifiche del chatbot, fornisce una struttura di base per l'applicazione e gestisce le operazioni comuni.

3.6 Sviluppo dell'interfaccia con SignaLR e Kendo

L'interfaccia utente del progetto è stata sviluppata utilizzando SignalR e Kendo UI, con l'obiettivo di creare un'esperienza interattiva e reattiva per l'utente. SignalR è stato utilizzato per gestire la comunicazione in tempo reale tra il client e il server, mentre Kendo UI è stato impiegato per costruire i componenti dell'interfaccia utente.

Il codice dell'interfaccia è organizzato in un oggetto JavaScript denominato *ochatbox*, che contiene vari metodi per gestire la sessione, la connessione SignalR e l'interazione con l'utente. Di seguito sono elencati i principali metodi utilizzati per gestire la chat nell'interfaccia del browser.

3.6.1 Gestione della sessione

Il metodo *session_setThread* viene chiamato quando l'utente inizia una nuova conversazione con l'assistente. In questo modo viene istanziato un thread, che verrà memorizzato nella sessione della conversazione.

```
session_setThread: function (threadID) {
    $.ajax({
        async: true,
        url: "/OpenAI/threadSet",
        type: "POST",
        dataType: "json",
        data: { threadId: threadID },
    }).done(function (data) {
        console.log("Thread settato con successo:", data);
    }).fail(function (xhr, status, error) {
        console.error("Errore durante il set del thread:", error);
    });
}
```

Figura 3.20: Metodo *session_getThread*

Il metodo *session_getThread* invece, invia una richiesta AJAX per recuperare il thread corrente dalla sessione. Se il thread esiste, viene utilizzato per recuperare la cronologia delle conversazioni tramite il metodo *chatbox_retrieveHistory*. In caso contrario, viene inizializzata una nuova sessione di messaggi con il metodo *chatbox_messagesInit*.

3.6.2 Recupero cronologia della conversazione

Il metodo *chatbox_retrieveHistory* utilizza la connessione SignalR per invocare il metodo *retrieveHistory* sull'Hub, recuperando la cronologia delle conversazioni per il thread specificato. I messaggi recuperati vengono poi visualizzati nell'interfaccia utente.

```
this.connection.invoke("retrieveHistory", threadId).then(function (response)
    {
    if (response == null) {
        console.log("errore");
    } else {
        this.flag == false;
        let messages = JSON.parse(response);
        ChatAgent.ChatAgent_postMessageHistory(ochatbox);

    }
}.bind(this)).catch(function (err) {
    console.error(err.toString());
});
```

Figura 3.21: Metodo *_retriveHistory()*

3.6.3 Gestione della visualizzazione della Chat

L'oggetto *ChatAgent* contiene metodi per visualizzare i messaggi nell'interfaccia utente, inclusi i messaggi iniziali, i messaggi di cronologia e le risposte del server.

```
ChatAgent_postMessageInit: function (ochatbox) {
    ochatbox.chat.renderMessage({
        type: "text",
        text: "Ciao! Sonol'assistente virtuale di Oxygen. Come posso
           aiutarti oggi?",
        timestamp: new Date()
    }, {
        name: "Oxygen"
    });
}
```

Figura 3.22: Esempio messaggio dell'interfaccia

Capitolo 4

Risultati ottenuti

Durante questo progetto, sono state implementate molteplici funzionalità per migliorare l'interazione degli utenti con il sistema di chat implementato. I risultati ottenuti e le ottimizzazioni effettuate per garantire l'efficienza del chatbot indicano che il progetto, in un futuro prossimo, potrebbe essere integrato nel software in modo ottimale, diventando una risorsa preziosa per gli utenti.

4.1 Funzionalità retrieval

Per quanto riguarda la funzionalità di *retrieval*, il chatbot è in grado di prelevare le informazioni corrette per rispondere alle domande in modo accurato e completo. Sebbene il file di prova rappresenti solo una parte dei diversi manuali disponibili per il software, è possibile considerare l'aggiunta di ulteriori informazioni per aumentare la "conoscenza" dell'assistente, permettendogli di fornire risposte sempre più complete, pertinenti e utili per l'utente.

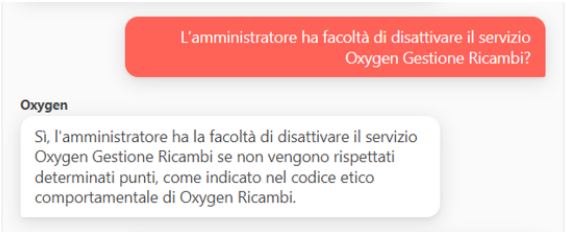

Figura 4.1: Chat con l'assistente

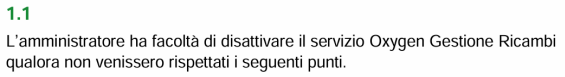

Figura 4.2: Manuale gestione ricambi

Come mostrato in 4.1, l'*Assistant*, risponde correttamente alla domanda relativa alla disattivazione del servizio Gestione Ricambi, in riferimento alla direttiva del manuale in figura 4.2. Questo dimostra che l'assistente è in grado di individuare le informazioni chiave nella domanda, consentendogli di recuperare i dati rilevanti dal manuale consultato in background.

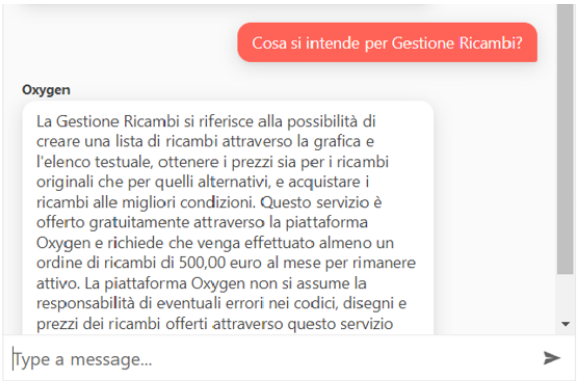

Figura 4.3: Chat con l'assistente 2

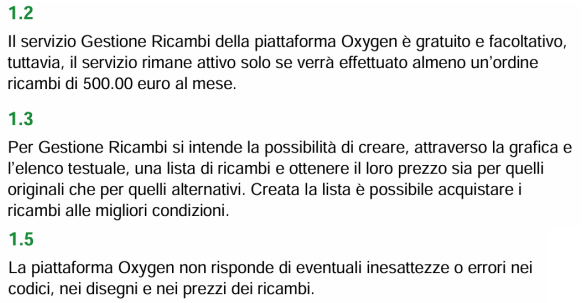

Figura 4.4: Manuale gestione ricambi 2

Nell'esempio 4.3 l'*Assistant*, oltre a rispondere bene alla domanda, fornisce una risposta più ampia integrando diverse risorse prelevate dal manuale, combinando le diverse direttive di figura 4.4 per fornire una risposta più completa e precisa.

Il testo generato dall'assistente risulta molto simile a quello presente nel manuale di riferimento. Questo perché il modello GPT, come *gpt-3.5-turbo*, utilizza l'auto-attenzione per analizzare il contesto delle parole circostanti e generare risposte. Tuttavia, poiché il modello non ha conoscenza pregressa del materiale specifico caricato (che è stato inserito tramite l'assistente), quando si trova di fronte a termini o concetti nuovi che non appartengono al suo addestramento originale, può avere difficoltà a costruire un contesto autonomo per generare una risposta. In questi casi, il modello si basa direttamente sulle sequenze di parole presenti nel manuale per formulare risposte coerenti, poiché non può fare affidamento sulla propria conoscenza pre-addestrata.

4.2 Funzionalità function

Nella versione corrente, si può affermare che la funzionalità *function* dell'assistente, funzioni correttamente: l'assistente riconosce in modo accurato quando deve eseguire la "function" e la esegue effettivamente in maniera corretta, restituendo le informazioni appropriate. Questa modalità presenta un grande potenziale poiché permetterebbe all'assistente di svolgere numerose

funzionalità. Ad esempio, collegando l'assistente al Pubblico Registro Automobilistico (PRA), sarebbe possibile ottenere informazioni di vario tipo relative a tutti i veicoli a motore immatricolati. Inoltre, utilizzando la *function*, si potrebbe provare a definire un preventivo relativo a un danno, fornendo le informazioni necessarie all'assistente tramite la chat. Le funzionalità che si possono implementare con questa modalità sono numerose e, grazie a queste, il chatbot potrebbe non solo rispondere alle domande, ma anche svolgere vere e proprie attività.

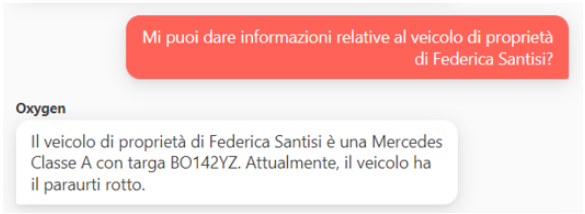

Figura 4.5: Chat con l'assistente 3

```
public static Automobile auto1 =
    new Automobile
{
    proprietario = "Federica
        Santisi",
    targa = "BO142YZ",
    modello = "Mercedes",
    marca = "Classe A",
    descrizione_del_danno = "Si e'
        rotto il paraurti",
};
```

Figura 4.6: Istanza della classe Automobile 1

```
public static Automobile auto2 =
    new Automobile
{
    proprietario = "Stefano Rossi"
        ,
    targa = "RT435CV",
    modello = "Toyota",
    marca = "Yaris Ibrid",
    descrizione_del_danno = "Si e'
        rotto lo stereo",
};
```

Figura 4.7: Istanza della classe Automobile 2

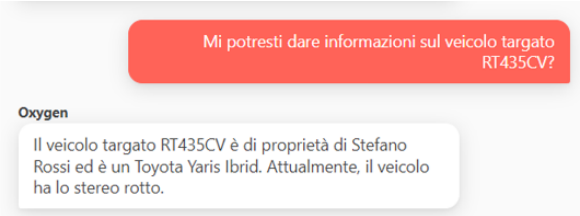

Figura 4.8: Chat con l'assistente 4

Come si può notare, le informazioni rilevanti nella domanda [1]vengono riconosciute nel modo corretto e scatenano l'attivazione della funzione. La funzione gestisce nel modo corretto le

[1]Le informazioni sono inventate

informazioni e va a recuperare nella lista, l'automobile inerente ai dati raccolti nella domanda.

4.3 Recupero dei messaggi dalla sessione

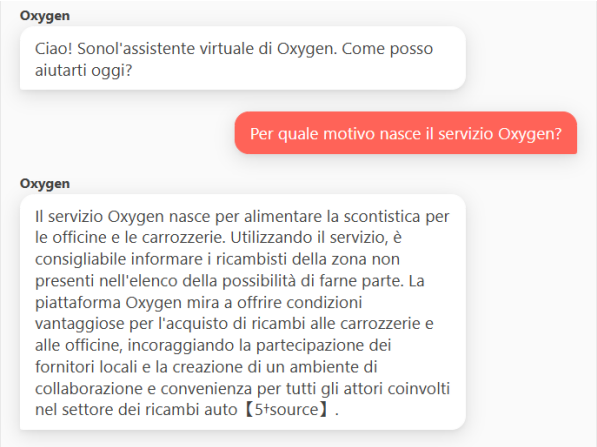

Figura 4.9: Chat con l'assistente 5

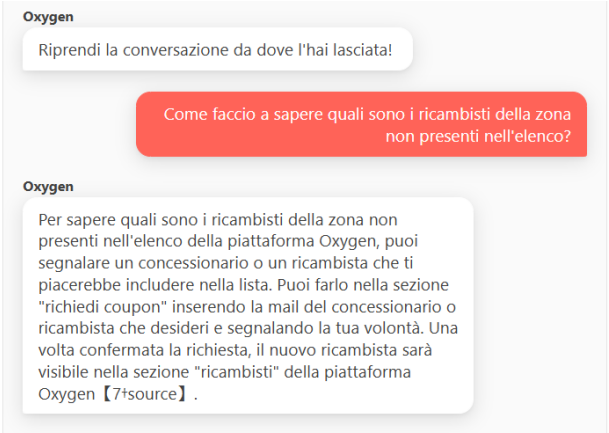

Figura 4.10: Chat con l'assistente 6

Grazie alla possibilità di salvare il thread nella sessione e recuperare i messaggi precedenti, è possibile fare domande che fanno riferimento a conversazioni passate, anche se i messaggi non sono più visibili nella chat. L'assistente è in grado di mantenere un contesto coerente, rispondendo correttamente anche a domande basate su contenuti che non sono più presenti nella chat attuale ma che appartengono alla stessa conversazione [2]. Il messaggio iniziale *"Riprendi la conversazione da dove l'hai lasciata"* conferma che i messaggi sono stati recuperati correttamente. Come mostrato in figura 4.10, viene posta una domanda senza riferimenti espliciti al contesto precedente. Tuttavia, la risposta risulta comunque pertinente e coerente, poiché il contesto della conversazione è stato preservato dal thread creato in seguito alla domanda mostrata in figura 4.9.

[2]Derivante dalla condivisione del thread

Capitolo 5

Conclusioni

Il progetto sviluppato utilizzando le API di OpenAI ha portato alla creazione di un assistente virtuale avanzato, progettato per rispondere efficacemente alle richieste degli utenti del servizio Oxygen Carrozzerie. Dalla definizione del client, con l'implementazione di tutti i metodi necessari per l'interazione con le API, alla gestione delle richieste e delle risposte lato server, fino alla creazione di un'interfaccia utente semplice e reattiva, ogni fase è stata progettata per migliorare l'esperienza dell'utente, rendendo la navigazione nel software più veloce e intuitiva.

L'obiettivo principale del chatbot era consentire agli utenti di sfruttare al meglio le numerose funzionalità del software, facilitando l'interazione e riducendo il tempo necessario per ottenere risposte o completare operazioni. Grazie a una comprensione avanzata delle esigenze degli utenti, l'assistente è in grado di fornire risposte precise e rapide, integrandosi perfettamente con servizi esterni e personalizzando l'interazione in base al contesto.

L'integrazione dell'assistente all'interno di Oxygen Carrozzerie rappresenta un importante passo verso una gestione più automatizzata e intelligente delle richieste degli utenti. Questa evoluzione apre la strada a ulteriori sviluppi futuri, con l'obiettivo di migliorare continuamente l'efficienza e l'efficacia del servizio offerto.

5.1 Sviluppi futuri

Il modello GPT utilizzato è in continua evoluzione, aprendo numerose possibilità per gli sviluppi futuri di questo progetto. Di seguito alcune possibili evoluzioni di questo progetto.

5.1.1 Espansione della conoscenza di base per il recupero dati

Il manuale attualmente caricato sulla piattaforma di OpenAI è limitato nelle dimensioni ed è specificamente dedicato al Regolamento Ricambi del servizio Oxygen. Tuttavia, come spiegato in precedenza, la piattaforma offre numerose funzionalità, supportate da diversi manuali che spiegano in dettaglio come utilizzare il servizio.

Poiché la capacità di retrieval dell'assistente non presenta limitazioni significative riguardo all'aggiunta di dati, è possibile espandere la conoscenza del chatbot integrando tutte queste informazioni aggiuntive. Questo consentirebbe all'assistente di rispondere in modo più preciso e completo anche a domande più specifiche, migliorando l'esperienza utente.

5.1.2 Integrazione dell'assistenza vocale

Come avviene per alcuni servizi ben noti, come Siri, anche il modello *GPT-4o* offre ora la possibilità di integrare API vocali, consentendo agli utenti di interagire con il chatbot tramite comandi vocali e ricevere risposte in forma audio. Questa funzionalità rappresenta una significativa innovazione per il chatbot implementato, poiché rende l'esperienza di utilizzo molto più intuitiva e accessibile, specialmente per chi preferisce o necessita di interfacce vocali rispetto a quelle testuali.

L'integrazione dei comandi vocali non solo migliora l'accessibilità per un pubblico più ampio, inclusi utenti con disabilità visive o chi desidera utilizzare il servizio in modalità hands-free, ma arricchisce anche l'interattività, permettendo una comunicazione più naturale e fluida. La possibilità di porre domande e ricevere risposte in tempo reale attraverso la voce avvicina l'assistente virtuale a un'esperienza simile a una conversazione umana, aumentando il livello di coinvolgimento e rendendo il chatbot uno strumento ancora più potente e versatile.

Questa evoluzione tecnologica, quindi, non solo eleva la qualità del servizio offerto, ma apre nuove possibilità per l'uso del chatbot in contesti dinamici, come l'assistenza in movimento o l'integrazione con dispositivi smart.

5.1.3 Generazione automatica di preventivi

Inoltre, l'assistente potrebbe essere progettato per guidare gli utenti attraverso un processo step-by-step, facilitando la generazione di preventivi in modo automatico. Riconoscendo l'intento dell'utente, l'assistente sarebbe in grado di porre una serie di domande mirate, raccogliendo in

modo progressivo tutte le informazioni necessarie per elaborare il preventivo, come i dettagli sul veicolo, il tipo di danno, i ricambi richiesti e altri dati rilevanti.

Questa funzionalità renderebbe il processo altamente automatizzato, riducendo al minimo l'intervento manuale da parte del carrozziere e garantendo al contempo una maggiore precisione nelle informazioni fornite. L'assistente potrebbe, ad esempio, verificare che i dati siano completi e corretti, segnalando eventuali informazioni mancanti o incongruenze, e quindi generare il preventivo finale in modo efficiente.

5.1.4 Caricamento di immagini

Un'altra implementazione particolarmente innovativa potrebbe essere la possibilità di caricare immagini dei pezzi danneggiati direttamente nell'assistente. In questo scenario, l'assistente non solo sarebbe in grado di elaborare le immagini ricevute, ma potrebbe anche collegarsi a servizi esterni specializzati per identificare automaticamente i componenti necessari per la riparazione. Questo approccio andrebbe oltre una semplice ricerca visiva, come quella offerta da servizi come Google Immagini, integrando capacità avanzate di riconoscimento degli oggetti con database di ricambi specifici per il settore automobilistico.

Grazie alla connessione con servizi esterni e fornitori specializzati, l'assistente sarebbe in grado di verificare la disponibilità dei pezzi di ricambio, confrontare le opzioni in base a parametri come prezzo, qualità e tempi di consegna, e persino suggerire soluzioni alternative in caso di indisponibilità immediata. Questa funzionalità aumenterebbe significativamente la precisione e l'efficienza del processo di preventivazione e riparazione, riducendo il margine di errore e i tempi di attesa per gli utenti.

5.2 Conclusione

Queste innovazioni non solo renderebbero l'interazione con l'assistente più intuitiva, ma contribuirebbero anche a creare un supporto altamente personalizzato e informativo, capace di rispondere in modo efficace alle esigenze specifiche degli utenti. La caratteristica distintiva di questo progetto è che il settore dell'assistenza virtuale è in costante evoluzione, con l'obiettivo di rendere l'esperienza utente con software, siti web e altre piattaforme sempre più semplice e immediata. Grazie a queste innovazioni, il progetto presenta numerose prospettive di sviluppo innovative, alcune delle quali potrebbero non essere ancora completamente immaginabili.

Riferimenti bibliografici

[1] VINCENZO BISCEGLIA. Chatbot. linee guida alla progettazione per le aziende. il caso wind. 2016.

[2] Suprita Das and Ela Kumar. Determining accuracy of chatbot by applying algorithm design and defined process. In *2018 4th International Conference on Computing Communication and Automation (ICCCA)*, pages 1–6. IEEE, 2018.

[3] Vidya Deshmukh, Ayush Shetty, Gaurav Singh, and Rajesh Parale. Breaking language barriers: Transformer based sentence translation. *JOURNAL OF TECHNICAL EDUCATION*, page 200, 2023.

[4] Adi Haviv, Ori Ram, Ofir Press, Peter Izsak, and Omer Levy. Transformer language models without positional encodings still learn positional information. *arXiv preprint arXiv:2203.16634*, 2022.

[5] Alessandro Iannella. "ok google, vorrei parlare con la poetessa saffo": intelligenza artificiale, assistenti virtuali e didattica della letteratura. *Thamyris, nova series: Revista de Didáctica de Cultura Clásica, Griego y Latín*, (10):81–104, 2019.

[6] Yiheng Liu, Hao He, Tianle Han, Xu Zhang, Mengyuan Liu, Jiaming Tian, Yutong Zhang, Jiaqi Wang, Xiaohui Gao, Tianyang Zhong, et al. Understanding llms: A comprehensive overview from training to inference. *arXiv preprint arXiv:2401.02038*, 2024.

[7] OpenAI. Openai platform documentation: Overview. `https://platform.openai.com/docs/overview`, 2023. Accessed: 2023-10-03.

[8] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[9] Gokul Yenduri, M Ramalingam, G Chemmalar Selvi, Y Supriya, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, G Deepti Raj, Rutvij H Jhaveri, B Prabadevi, Weizheng Wang, et al. Gpt (generative pre-trained transformer)–a comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions. *IEEE Access*, 2024.

Appendices

Appendice A

Elenco API utilizzate

A.1 Richieste POST

- **CreateAssistant(Assistant assistant)**

  URL: (*"https://api.openai.com/v1/assistants"*, content)

  Il metodo *createAssistant* converte l'oggetto *assistant*, passato come parametro, in un oggetto JSON utilizzando il metodo preliminare *createContent*. Successivamente, esegue una richiesta POST asincrona al server tramite l'URL specificato. Il risultato di questa richiesta POST è la creazione dell'assistente, con conseguente rilascio di un *JObject* contenente un *id* univoco associato all'assistente stesso.

- **sendQuestion(Question question, string_threadId)**

  URL: (*"https://api.openai.com/v1/threads/_threadId/messages"*, content)

  Il metodo *sendQuestion* è utilizzato per inviare una domanda a un thread specifico. Esso inizia trasformando l'oggetto *Question* in una stringa JSON utilizzando il metodo *createContent*. Successivamente, il metodo invia una richiesta POST all'endpoint dell'API, costruendo dinamicamente l'URL con l'identificatore del thread (_threadId).

- **CreateRun(Run run, string _threadId)**

  URL: (*"https://api.openai.com/v1/threads/_threadId/runs",*, content)

Il metodo *createRun()* è progettato per creare un nuovo "run" all'interno di un thread specifico. Esso inizia trasformando l'oggetto *Run* in una stringa JSON utilizzando il metodo *createContent*. Successsivamente, il metodo invia una richiesta POST all'endpoint dell'API, costruendo l'URL con l'identificatore del *_threadId* e aggiungendo */runs* al percorso per specificare l'azione desiderata. Il risultato di questa richiesta POST è la creazione del *run*, con conseguente rilascio di un *JObject* contenente un *id* univoco associato al run stesso.

- **CreateThread()**

  URL: (*"https://api.openai.com/v1/threads"*,, null)

  Il metodo *createThread()* è utilizzato per creare un nuovo thread. Esso invia una richiesta POST all'endpoint dell'API specifico per i thread. Poiché non è necessario inviare alcun contenuto nel corpo della richiesta, il secondo parametro è impostato a *null*. Il risultato di questa richiesta POST è la creazione del *thread*, con conseguente rilascio di un *JObject* contenente un *id* univoco associato al thread stesso.

- **cancelRun(string threadId, string runId)**

  URL:(*"https://api.openai.com/v1/threads/threadId/runs/runId/cancel"*, null)

  Il metodo *cancelRun()* è utilizzato per annullare un "run" specifico all'interno di un thread. Esso invia una richiesta POST all'endopoint specificato, costruendo l'URL con l'identificatore *threadId* e l'identificatore *runId*, aggiungendo il percorso */cancel*. Poiché non è necessario inviare alcun contenuto nel corpo della richiesta, il secondo parametro è impostato a *null*. La risposta del server, che conferma l'annullamento del "run", viene de serializzata in un oggetto JSON e restituita dal metodo, permettendo di verificarne l'effettivo annullamento.

A.2 Richieste GET

- **GetResponse(string _threadId)**

  URL: *("https://api.openai.com/v1/threads/_threadId/messages")*

  Il metodo *GetResponse()* recupera i messaggi associati a un thread specifico tramite l'endpoint specificato. Esso invia una richiesta GET, utilizzando l'identificatore del thread che

"punta" ai messaggi di quel thread. La risposta del server, contenente i messaggi del thread, viene deserializzata in un oggetto JSON e restituita dal metodo, permettendo di accedere ai messaggi associati a quel thread specifico.

- **CheckStatus(string _threadId, string _runId)**

  URL: *("https://api.openai.com/v1/threads/_threadId/runs/_runId")*

  Il metodo *checkStatus()* è progettato per verificare lo stato di un "run" specifico all'interno di un thread tramite l'API di OpenAI. Esso invia una richiesta GET all'endpoint specificato, costruendo l'URL con l'identificatore *_threadId* e l'identificatore *_runId*. La risposta del server, contenente informazioni sullo stato del "run", viene de serializzata in un oggetto JSON e restituita dal metodo, consentendo di accedere ai dettagli del "run e monitorandone lo stato".

- **retrieveMessages(string threadId)**

  URL: *("https://api.openai.com/v1/threads/threadId/messages")*

  Il metodo *retrieveMessages()* è utilizzato per recuperare i messaggi associati a un thread. Esso invia una richiesta GET all'endopoint specificato, costruendo l'URL con l'identificatore *_threadId* per accedere ai messaggi di quel thread specifico. La risposta del server, contenente i messaggi del thread specificato, viene de serializzata in un oggetto JSON e restituita dal metodo permettendo di accedere e utilizzare i messaggi ottenuti.

- **retrieveRun(string threadId)**

  URL: *("https://api.openai.com/v1/threads/threadId/runs")*

  Il metodo *retrieveRun()* è utilizzato per recuperare i run associati a un thread. Esso invia una richiesta GET all'endopoint specificato, costruendo l'URL con l'identificatore *_threadId* per accedere ai run di quel thread specifico. La risposta del server, contenente i run del thread specificato, viene de serializzata in un oggetto JSON e restituita dal metodo permettendo di accedere agli identificatori dei run ottenuti.