Dipartimento di Ingegneria Civile, Informatica
e delle Tecnologie Aeronautiche

Corso di Laurea in Ingegneria Informatica

Aumentare le capacità di un chatbot usando Model Context Protocol (MCP)

Anno Accademico 2024/2025

Laureando

Del Prete Andrea

Matricola 589453

Relatore

Prof. Merialdo Paolo

Tutor

Dott. Di Nardo Giorgio

Indice

Introduzione

Siamo oramai in un'era in cui l'intelligenza artificiale è presente in ogni aspetto della nostra vita digitale, ancora di più dopo l'uscita di ChatGPT, che infranse il record di applicazione con la più rapida crescita nella storia di Internet, ben 100 milioni di utenti in soli 2 mesi [1]. Gli ultimi anni in particolare sono stati caratterizzati dall'intelligenza artificiale generativa, capace cioè di generare testi, audio, immagini o video a partire da un semplice "prompt" dell'utente, come ad esempio i modelli GPT alla base di ChatGPT stesso.

Di particolare interesse sono le IA capaci di generare testo, i cosiddetti Large Language Model (LLM), letteralmente modelli linguistici di grandi dimensioni, in grado di emulare con sorprendente accuratezza le capacità conversazionali di un essere umano. Un'interessante conseguenza del modo in cui questi algoritmi sono stati creati, ma soprattutto della mole di dati usata per addestrarli, è stato l'emergere di comportamenti che potremmo considerare intelligenti, con un livello quasi al pari di quello di un essere umano. Ad esempio, gli LLM sono in grado di ricordare informazioni, generare risposte articolate basate su un ampio contesto e ragionare su problemi complessi derivanti dalle branche più disparate, dalla meccanica quantistica alla biologia evolutiva.

Negli anni è perciò nato il desiderio di rendere ancora più sofisticate le abilità di questi modelli linguistici, finora relegate al "semplice" rispondere ai messaggi degli utenti. Ed è per questo che Anthropic, una delle maggiori aziende nel settore della ricerca sugli LLM e creatrice dei molto diffusi modelli Claude, ha ideato un meccanismo che facilita questo obiettivo: il Model Context Protocol (MCP). Questo

protocollo open-source definisce una procedura standard che consente alle applicazioni di fornire contesto ai modelli linguistici, permettendo loro di accedere ad ed interagire con una serie di strumenti esterni in modo sicuro e controllato. Grazie a ciò, gli LLM possono eseguire operazioni specifiche, recuperare informazioni o integrare funzionalità aggiuntive, migliorando l'efficacia e la personalizzazione delle risposte generate [2].

Obiettivo di questa tesi è illustrare un progetto volto a dimostrare nella pratica le funzionalità del Model Context Protocol. Il progetto in questione è un sistema di ticket management, che consente agli utenti di interfacciarsi con un chatbot che rende automatica la creazione di ticket delineanti i problemi riscontrarti e l'inoltro degli stessi a degli sviluppatori, che possono visionarli ed avviare delle procedure per risolverli.

1

Large Language Model - Cosa sono e come funzionano

1.1 Breve storia

All'inizio degli anni '90, i modelli statistici di IBM hanno aperto la strada alle tecniche di allineamento delle parole per la traduzione automatica. Durante gli anni 2000, con l'aumento dell'accesso diffuso a Internet, i ricercatori hanno iniziato a compilare enormi set di dati testuali dal web ("web as corpus" [3]) per addestrare questi modelli linguistici statistici [4][5], chiamati modelli n-gram [6].
Andando oltre i modelli n-gram, nel 2000 i ricercatori hanno iniziato a utilizzare le reti neurali per addestrare i modelli linguistici [7]. In seguito al successo delle reti neurali profonde ("Deep Neural Networks") nella classificazione delle immagini intorno al 2012 [8], architetture simili sono state adattate per compiti linguistici. Questo cambiamento è stato segnato dallo sviluppo di "word embeddings" (ad esempio, Word2Vec di Mikolov nel 2013) e modelli "seq2seq". Nel 2016, Google ha convertito il suo servizio di traduzione alla traduzione automatica neurale ("Neural Machine Translation"), sostituendo i modelli statistici basati su frasi con reti neurali profonde ricorrenti ("Deep Recurrent Neural Networks").

Alla conferenza NeurIPS del 2017, i ricercatori di Google hanno introdotto l'architettura del trasformatore in un articolo, divenuto fondamentale nel settore, chiamato "Attention Is All You Need" [9]. L'obiettivo di questo articolo era quello di migliorare la tecnologia seq2seq del 2014 basandosi su un meccanismo chiamato attenzione, sviluppato nel 2014 [10]. L'anno successivo, nel 2018, è stato introdotto BERT, che è rapidamente diventato "onnipresente" [11]. Sebbene il trasformatore originale abbia sia blocchi encoder che decoder, BERT è un modello solo encoder. L'uso accademico e di ricerca di BERT ha iniziato a diminuire nel 2023, a seguito di rapidi miglioramenti nelle capacità dei modelli solo decoder (come GPT) di risolvere compiti tramite prompt [12].
Sebbene GPT-1 ("Generative Pre-Trained Transformer"), modello solo decoder, sia stato introdotto nel 2018, è stato GPT-2 nel 2019 ad attirare l'attenzione generale, poichè OpenAI ha affermato di averlo inizialmente ritenuto troppo potente per essere rilasciato pubblicamente, per timore di un utilizzo dannoso [13]. Ma è stata l'applicazione ChatGPT del 2022, con la sua interfaccia utente che riprende il concetto di chatbot, a ricevere un'ampia copertura mediatica e l'attenzione del pubblico [14]. GPT-4 del 2023 è stato elogiato per la sua maggiore accuratezza e definito un "Sacro Graal" per le sue capacità multimodali (i.e.: la sua capacità di analizzare dati di diverso tipo, come testo, audio e immagini) [15]. Il rilascio di ChatGPT ha portato ad un aumento nell'utilizzo degli LLM in diversi sottocampi di ricerca dell'informatica, tra cui robotica ed ingegneria del software, oltre che ad un maggior riguardo per il loro impatto sociale [12]. Nel 2024 OpenAI ha rilasciato il modello di ragionamento OpenAI o1, che genera lunghe catene di pensiero prima di restituire una risposta finale [16]. Negli anni sono stati sviluppati molti LLM con dimensioni paragonabili a quelli della serie GPT di OpenAI [17].
Dal 2022, i modelli linguistici open-source hanno guadagnato popolarità, soprattutto con i primi modelli BLOOM e LLaMA di Meta e anche i modelli Mistral 7B e Mixtral 8x7b di Mistral AI. Nel gennaio 2025, DeepSeek ha rilasciato DeepSeek R1, un modello open-weight da 671 miliardi di parametri che offre prestazioni parago-

nabili a OpenAI o1, ma a un costo molto inferiore [18].
Dal 2023, molti LLM sono stati addestrati per essere multimodali, con la capacità di elaborare o generare anche altri tipi di dati, come immagini o audio. Questi LLM sono anche chiamati grandi modelli multimodali ("Large Multimodal Models") [19].
Ad oggi, i modelli più grandi e performanti sono tutti basati sull'architettura a trasformatore [20].

1.2 Accenni del funzionamento

Fondamentalmente, un Large Language Model (LLM) è in grado di prevedere quale sia la parola più plausibile da inserire alla fine di un testo dato, un processo che in gergo tecnico si chiama inferenza.
Poiché gli algoritmi di Machine Learning possono elaborare esclusivamente informazioni numeriche, il testo in ingresso viene innanzitutto suddiviso in frammenti più piccoli, chiamati *token*. Un token non coincide necessariamente con una parola intera: può rappresentare una parola completa, una parte di essa o anche un singolo carattere, a seconda della suddivisione stabilita dal sistema di tokenizzazione.
Ogni token viene poi convertito in un vettore numerico, detto *embedding*. Ogni embedding rappresenta il token in uno spazio a molte dimensioni (spesso centinaia o migliaia), in cui la distanza e la direzione tra vettori riflettono relazioni semantiche e sintattiche tra parole. Ad esempio, gli embedding di "Italia" e "Francia" saranno vicini in quanto entrambe le parole rappresentano nazioni europee. In questo spazio vettoriale si trovano tutti gli embedding dei token che il modello ha incontrato durante la fase di addestramento. È proprio in questa fase che l'algoritmo "impara" a trasformare i token in embedding significativi, modificando gradualmente i propri parametri — talvolta dell'ordine di miliardi — in modo da ridurre l'errore nelle previsioni e aumentare la coerenza dei testi generati.
Come accennato prima, i moderni LLM si basano sull'architettura *transformer*, caratterizzata dall'uso del meccanismo dell'attenzione (*attention mechanism*). All'in-

terno di questa architettura, i vari embedding vengono manipolati matematicamente attraverso una serie di passaggi che includono moltiplicazioni di matrici, normalizzazioni e funzioni di attivazione, fino a produrre un vettore finale corrispondente alla previsione di un token specifico. Una volta generato questo token, viene aggiunto alla fine del testo e l'intero contenuto aggiornato viene nuovamente elaborato dal modello, ripetendo il ciclo fino a costruire un testo completo in risposta al prompt dell'utente.

Il meccanismo dell'attenzione è un passaggio cruciale: esso permette al modello di valutare, per ogni token in ingresso, quali altri token della sequenza siano più rilevanti per determinarne il significato. In altre parole, non si limita a interpretare un token in maniera isolata, ma lo considera nel contesto più ampio della frase o del paragrafo in cui si trova. Il termine "attenzione" è ispirato a un processo analogo nella cognizione umana, in cui focalizziamo la nostra concentrazione su determinate parti di un testo per comprenderne appieno il senso. Grazie a questa capacità, gli LLM riescono a mantenere coerenza logica anche su sequenze testuali lunghe, collegando correttamente informazioni distanti tra loro e producendo risposte che seguono un filo narrativo o argomentativo esteso.

L'introduzione di questo meccanismo ha segnato un punto di svolta nel settore. Tuttavia, la sua applicazione su larga scala è stata possibile solo grazie ai progressi nell'hardware, in particolare all'uso delle GPU (*Graphics Processing Unit*). Le GPU, inizialmente progettate per l'elaborazione grafica, sono straordinariamente efficienti nel calcolo parallelo di operazioni matriciali, proprio quelle su cui si basano il meccanismo dell'attenzione e l'intera architettura transformer. Questo connubio tra innovazione algoritmica e potenza di calcolo ha permesso di addestrare modelli con miliardi di parametri su dataset enormi, aprendo la strada alla generazione di testi complessi e coerenti su scala mai vista prima.

Durante l'addestramento iniziale, il modello viene esposto ad enormi quantità di testo provenienti da fonti eterogenee, imparando a prevedere il token successivo dato un contesto. Per ogni previsione calcola l'errore rispetto al token reale e regola i

propri miliardi di parametri tramite il processo di *backpropagation*, ottimizzandoli gradualmente per ridurre l'errore medio.
Un modello già addestrato può essere ulteriormente specializzato tramite fine-tuning, che consiste nel riaddestrarlo (in parte o per intero) su un set di dati più ristretto e specifico, ad esempio testi tecnici o conversazioni in uno stile particolare. Questa procedura permette di adattare l'LLM a compiti specifici — come assistenza clienti, generazione di codice o analisi di documenti giuridici — senza ripetere da zero il costoso addestramento iniziale.
Un'ulteriore tecnica molto diffusa è RLHF (*Reinforcement Learning from Human Feedback*), utilizzata per rendere le risposte del modello più utili, sicure e in linea con i valori umani. In questo approccio, dopo l'addestramento iniziale, il modello genera diverse possibili risposte ad un insieme di prompt; queste risposte vengono valutate da valutatori umani, che le classificano in base a criteri come accuratezza, cortesia o pertinenza. Con queste valutazioni, si addestra un modello a parte capace di sostituire, il più accuratamente possibile, il compito dei valutatori umani. L'LLM viene così ottimizzato per soddisfare al meglio la valutazione di questo modello.

2

Model Context Protocol

2.1 Funzionamento

MCP è un protocollo open-source che standardizza il modo in cui le applicazioni forniscono contesto agli LLM. Si può pensare a MCP come ad una porta USB-C per le applicazioni IA. Proprio come USB-C fornisce un modo standardizzato per collegare i dispositivi a varie periferiche e accessori, MCP fornisce un modo standardizzato per collegare i modelli IA a diverse fonti di dati e strumenti. MCP consente di creare agenti e flussi di lavoro complessi basati sugli LLM e connette i modelli con il mondo. [21]

Il protocollo MCP include i seguenti progetti:

- Specifiche MCP: una specifica di MCP che delinea i requisiti di implementazione per client e server.
- SDK MCP: *Software Development Kit* (SDK) per diversi linguaggi di programmazione che implementano MCP.
- Strumenti di sviluppo MCP.
- Implementazioni di server MCP di riferimento.

MCP si concentra esclusivamente sul protocollo per lo scambio di contesto, senza stabilire come le applicazioni IA utilizzino gli LLM o gestiscano il contesto fornito. [22]

2.1.1 Partecipanti

MCP segue un'architettura client-server in cui un host MCP, un'applicazione IA come Claude Code o Claude Desktop, stabilisce connessioni ad uno o più server MCP. L'host MCP realizza questo creando un client MCP per ogni server MCP. Ogni client MCP mantiene una connessione uno-a-uno dedicata con il suo server MCP corrispondente. I principali partecipanti all'architettura MCP sono:

- Host MCP: l'applicazione IA che coordina e gestisce uno o più client MCP.
- Client MCP: un componente che mantiene una connessione ad un server MCP ed ottiene, da un server MCP, il contesto che l'host MCP può utilizzare.
- Server MCP: un programma che fornisce contesto ai client MCP; possono essere eseguiti sia in locale che in remoto. [22]

2.1.2 Livelli

MCP è costituito da due livelli:

- Livello dati: definisce il protocollo, basato su JSON-RPC, per la comunicazione client-server, inclusa la gestione del ciclo di vita e le primitive principali, come strumenti, risorse, prompt e notifiche.
- Livello trasporto: definisce i meccanismi e i canali di comunicazione che consentono lo scambio di dati tra client e server, inclusi l'instaurazione di connessioni specifiche per il trasporto, il framing dei messaggi e l'autorizzazione.

Concettualmente, il livello dati è il livello interno, mentre il livello trasporto è il livello esterno. [22]

Livello dati

Il livello dati implementa un protocollo di scambio basato su JSON-RPC 2.0 che definisce la struttura e la semantica dei messaggi. Questo livello include:

- Gestione del ciclo di vita: gestisce l'inizializzazione della connessione, la negoziazione delle capacità e la terminazione della connessione tra client e server.
- Funzionalità del server: consente ai server di fornire funzionalità di base, inclusi strumenti per azioni IA, risorse per dati di contesto e richieste per schemi di interazione da e verso il client.
- Funzionalità del client: consente ai server di chiedere al client di campionare dall'LLM, ottenere input dall'utente e registrare messaggi al client.
- Funzionalità di utilità: supporta funzionalità aggiuntive come notifiche per aggiornamenti in tempo reale e monitoraggio dei progressi per operazioni di lunga durata. [22]

Livello di trasporto

Il livello di trasporto gestisce i canali di comunicazione e l'autenticazione tra client e server. Gestisce la creazione della connessione, il framing dei messaggi e la comunicazione sicura tra i partecipanti. MCP supporta due meccanismi di trasporto:

- *Stdio Transport*: utilizza flussi di input/output standard per la comunicazione diretta tra processi locali sulla stessa macchina, garantendo prestazioni ottimali senza sovraccarico di rete.
- *Streamable HTTP transport*: utilizza metodi HTTP POST per i messaggi client-server con eventi inviati dal server opzionali per le funzionalità di streaming. Questo trasporto consente la comunicazione con il server remoto e supporta metodi di autenticazione HTTP standard, inclusi *bearer token*, chiavi API e intestazioni personalizzate.

MCP consiglia di utilizzare OAuth per ottenere i token di autenticazione. Il livello di trasporto astrae i dettagli di comunicazione dal livello di protocollo, consentendo lo stesso formato di messaggio del protocollo JSON-RPC 2.0 su tutti i meccanismi di trasporto. [22]

2.1.3 Protocollo del livello dati

Una parte fondamentale di MCP è la definizione dello schema e della semantica tra client e server MCP. Il livello dati è la parte di MCP che definisce le modalità con cui gli sviluppatori possono condividere il contesto dai server MCP ai client MCP. MCP utilizza JSON-RPC 2.0 come protocollo *Remote Procedure Call* (RPC). Client e server si inviano richieste e rispondono di conseguenza. Le notifiche possono essere utilizzate quando non è richiesta alcuna risposta. [22]

Primitive

Le primitive MCP sono il concetto più importante all'interno di MCP. Definiscono ciò che client e server possono offrirsi reciprocamente. Queste primitive specificano i tipi di informazioni contestuali che possono essere condivise con le applicazioni IA e la gamma di azioni che possono essere eseguite. MCP definisce tre primitive principali che i server possono esporre:

- *Tools*: funzioni eseguibili che le applicazioni IA possono invocare per eseguire azioni (ad es.: operazioni su file, chiamate API, query di database).
- Risorse: fonti di dati che forniscono informazioni contestuali alle applicazioni IA (ad es.: contenuto di file, record di database, risposte API).
- Prompt: schemi riutilizzabili che aiutano a strutturare le interazioni con i modelli linguistici (ad es.: prompt di sistema, prompt *few-shot*).

Ogni tipo di primitiva ha metodi associati per la scoperta (`*/list`), il recupero (`*/get`) e, in alcuni casi, l'esecuzione (`tools/call`). I client MCP utilizzeranno

i metodi `*/list` per scoprire le primitive disponibili. Ad esempio, un client può prima elencare tutti gli strumenti disponibili (`tools/list`) e poi eseguirli. Questa progettazione consente di creare elenchi dinamici.
Come esempio concreto, si consideri un server MCP che fornisce contesto su un database. Può esporre strumenti per interrogare il database, una risorsa che contiene lo schema del database e un prompt che include esempi di interazione con gli strumenti.
MCP definisce anche le primitive che i client possono esporre. Queste primitive consentono agli autori del server MCP di creare interazioni più ricche.

- Campionamento: consente ai server di richiedere il completamento del modello linguistico dall'applicazione IA del client. Questa funzionalità è utile quando gli autori del server desiderano accedere ad un modello linguistico, ma vogliono rimanere indipendenti dal modello e non includere un SDK del modello linguistico nel proprio server MCP. Possono utilizzare il metodo `sampling/complete` per richiedere il completamento del modello linguistico dall'applicazione IA del client.

- Elicitazione: consente ai server di richiedere informazioni aggiuntive agli utenti. Questa funzionalità è utile quando gli autori del server desiderano ottenere maggiori informazioni dall'utente o chiedere la conferma di un'azione. Possono utilizzare il metodo `elicitation/request` per richiedere informazioni aggiuntive all'utente.

- Logging: consente ai server di inviare messaggi di log ai client a scopo di debug e monitoraggio. [22]

Notifiche

Il protocollo supporta notifiche in tempo reale per abilitare aggiornamenti dinamici tra server e client. Ad esempio, quando gli strumenti disponibili su un server cambiano, come quando vengono rese disponibili nuove funzionalità o vengono modificati

strumenti esistenti, il server può inviare notifiche di aggiornamento per informare i client connessi di tali modifiche. Le notifiche vengono inviate come messaggi di notifica JSON-RPC 2.0 (senza attendere una risposta) e consentono ai server MCP di fornire aggiornamenti in tempo reale ai client connessi. [22]

3

Ticket Management System

3.1 Introduzione

La soluzione *TicketManagement* è stata concepita come piattaforma modulare e scalabile per la gestione strutturata di ticket, ossia unità di lavoro che rappresentano richieste, incidenti o attività da tracciare all'interno di un'organizzazione. L'adozione di sistemi di questo tipo risponde a un'esigenza ormai diffusa in contesti eterogenei, che spaziano dal supporto tecnico informatico alla gestione di eventi fino al coordinamento di flussi operativi complessi. In tali scenari, l'obiettivo primario non è soltanto quello di centralizzare la registrazione delle richieste, ma anche di garantire un processo di lavorazione tracciabile, collaborativo ed efficiente, capace di ridurre i tempi di risoluzione e di migliorare la produttività complessiva dei team coinvolti.

Il sistema affronta in particolare una serie di problematiche tipiche nella gestione delle attività organizzative. Tra queste, si annoverano la necessità di centralizzare le richieste, evitando dispersioni di informazioni; la possibilità di monitorare lo stato e la cronologia di ciascun ticket, garantendo trasparenza e accountability; la gestione flessibile dell'assegnazione dei task, sia essa automatica o manuale; l'erogazione di notifiche in tempo reale agli utenti coinvolti, al fine di favorire la comunicazione

tempestiva; e, infine, la produzione di analisi e reportistica utili a valutare l'efficienza dei processi.

Per rispondere a tali esigenze, la *solution* offre un insieme articolato di funzionalità. Tra le principali si annoverano la creazione, la modifica e la chiusura dei ticket, la gestione degli utenti e dei relativi ruoli, l'implementazione di workflow di assegnazione ed escalation, nonché la possibilità di integrazione con sistemi esterni mediante API. Ulteriori caratteristiche qualificanti comprendono una dashboard interattiva con strumenti di reportistica e il supporto nativo al pattern *CQRS* (Command Query Responsibility Segregation), che favorisce una maggiore scalabilità e manutenibilità del sistema.

Dal punto di vista tecnologico, *TicketManagement* si fonda su un'infrastruttura moderna e consolidata. Il progetto utilizza *.NET 8/9* come framework principale per lo sviluppo backend e frontend, sfruttandone le capacità in termini di prestazioni, sicurezza e aderenza alle architetture contemporanee. La persistenza dei dati è gestita attraverso *Entity Framework Core*, che semplifica l'interazione con database relazionali mediante l'uso di un ORM (Object-Relational Mapping). La creazione di API RESTful e interfacce web è supportata da *ASP.NET Core*, mentre l'adozione di un framework *CQRS* dedicato consente di separare le operazioni di lettura e scrittura, incrementando la scalabilità e migliorando la manutenzione del codice. Completano l'ecosistema ulteriori librerie dedicate al logging, all'autenticazione e alla gestione delle notifiche, che contribuiscono a rendere la soluzione robusta, sicura e facilmente estendibile.

In sintesi, *TicketManagement* si configura come un sistema complesso ma al tempo stesso flessibile, capace di affrontare in maniera organica le principali sfide legate alla gestione dei ticket. La combinazione tra un'architettura modulare, un set di funzionalità avanzate e l'impiego di tecnologie all'avanguardia rende la *solution* idonea a diversi contesti applicativi e ne sottolinea il potenziale come strumento di supporto alla produttività e all'efficienza aziendale.

3.1.1 Struttura architetturale della solution

La *solution TicketManagement* è stata progettata secondo un approccio modulare, in cui le diverse responsabilità funzionali vengono distribuite tra più progetti indipendenti ma interconnessi. Tale organizzazione riflette i principi delle moderne architetture software a livelli, favorendo la separazione delle preoccupazioni (*Separation of Concerns*), la manutenibilità e la possibilità di estendere il sistema senza introdurre dipendenze circolari o accoppiamenti eccessivamente rigidi.
La composizione della solution prevede cinque progetti principali, ciascuno con un ruolo ben definito. Il progetto *TM.Shared*, realizzato come libreria di classi, rappresenta la base comune dell'intero ecosistema e contiene modelli, costanti e logica condivisa, utilizzati trasversalmente dagli altri moduli. *TM.Data*, anch'esso strutturato come libreria di classi, ha il compito di gestire la persistenza delle informazioni, fornendo l'accesso al database e interagendo strettamente con i modelli definiti in *TM.Shared*. Il progetto *TM.CQRS* introduce l'implementazione del pattern *Command Query Responsibility Segregation*, offrendo un ulteriore livello di astrazione e separazione tra operazioni di lettura e scrittura dei dati; esso dipende da *TM.Shared* e coopera sia con *TM.Data* sia con *TM.Function*.
Il cuore della logica applicativa è rappresentato da *TM.Function*, presumibilmente implementato come *Web API*. Esso svolge il ruolo di orchestratore, coordinando le operazioni tra i moduli di persistenza e quelli che applicano i pattern architetturali, ed esponendo al contempo servizi RESTful utilizzati dai client esterni. Infine, *TM.Client* costituisce l'interfaccia utente, che può essere realizzata come applicazione desktop o come front-end web. Questo progetto comunica con *TM.Function* tramite API e, come gli altri moduli, dipende da *TM.Shared* per il riuso di modelli e definizioni comuni.
Da questa rappresentazione emerge chiaramente la natura stratificata della solution: *TM.Shared* costituisce il nucleo comune, *TM.Data* e *TM.CQRS* estendono e specializzano la logica, *TM.Function* funge da coordinatore e punto di accesso esterno, mentre *TM.Client* fornisce la parte di interazione diretta con l'utente. Tale disposi-

zione rispetta un modello a livelli, assimilabile a una combinazione tra architettura *layered* e *service-oriented*, con una distinzione netta tra presentazione, business logic e accesso ai dati.
Questa scelta architetturale consente non solo di migliorare la chiarezza e la modularità del codice, ma anche di garantire una maggiore scalabilità, permettendo l'eventuale sostituzione o estensione di singoli moduli senza compromettere la stabilità dell'intera soluzione.

3.2 Descrizione dei progetti

3.2.1 TM.Client

Il progetto *TM.Client* costituisce il front-end della soluzione *TicketManagement* ed è dedicato alla gestione dell'interfaccia utente e delle interazioni con il backend. La sua funzione primaria è consentire agli utenti finali di creare, visualizzare, modificare ed eliminare entità del sistema (ticket, task, commenti, categorie e utenti), fungendo da punto di accesso principale alla piattaforma. In questo contesto, *TM.Client* svolge un ruolo di mediazione tra la presentazione grafica e la logica di business implementata nel progetto *TM.Function*.

3.2.1.1 Namespace principali

Il progetto è strutturato in namespace distinti che favoriscono la separazione delle responsabilità:

- *TM.Client*: contiene la logica complessiva dell'interfaccia utente e i servizi di comunicazione con il backend.
- *TM.Client.Services*: gestisce le chiamate verso le API esposte da *TM.Function*, occupandosi dell'integrazione dei dati e delle operazioni CRUD.

- *TM.Client.ViewModels*: implementa la logica di presentazione secondo il pattern *MVVM* (Model-View-ViewModel), garantendo il binding tra i dati e i componenti grafici.
- *TM.Client.Views*: definisce le pagine, finestre o componenti visuali che costituiscono l'esperienza utente.

3.2.1.2 Classi rilevanti

Tra le classi più significative si distinguono:

- *MainWindow / App*: rappresentano il punto di ingresso dell'applicazione, inizializzando il ciclo di vita del client.
- *TicketViewModel*, *UserViewModel*, *TaskViewModel*, *CategoryViewModel*, *CommentViewModel*: ciascuna di queste classi incapsula lo stato e le operazioni relative a specifiche entità del dominio, offrendo un'interfaccia coerente alla UI.
- *ApiService*, *TicketService*, *UserService*: gestiscono la comunicazione con il backend, incapsulando la logica di invio e ricezione delle richieste HTTP verso *TM.Function*.
- *DTOs* (Data Transfer Objects): modelli utilizzati per il trasferimento dei dati tra client e backend, spesso derivati da *TM.Shared* per garantire uniformità e coerenza semantica.

3.2.1.3 Dipendenze

TM.Client dipende direttamente da *TM.Shared*, da cui eredita modelli e costanti, e da *TM.Function*, con cui comunica tramite API RESTful. A livello tecnico, utilizza librerie per la costruzione dell'interfaccia (ad esempio *WPF*, *Blazor* o equivalenti) e strumenti di supporto come *Newtonsoft.Json* o *System.Text.Json* per la serializzazione e deserializzazione dei dati.

3.2.1.4 Ruolo nella solution

In sintesi, *TM.Client* rappresenta l'anello di congiunzione tra l'utente finale e la logica di business. Grazie all'organizzazione in *Views*, *ViewModels* e *Services*, garantisce una netta separazione delle responsabilità e contribuisce a un'architettura modulare e manutenibile. La sua progettazione orientata all'utente lo rende il componente fondamentale per l'accesso e l'interazione con le funzionalità offerte dalla piattaforma *TicketManagement*.

3.2.2 TM.CQRS

Il progetto *TM.CQRS* rappresenta il nucleo della logica di orchestrazione della soluzione *TicketManagement*, implementando il pattern *CQRS* (Command Query Responsibility Segregation). Tale approccio consente di separare in maniera chiara le operazioni di scrittura, gestite dai comandi (*commands*), da quelle di lettura, gestite dalle query, garantendo una maggiore scalabilità, manutenibilità e testabilità del sistema. *TM.CQRS* si configura quindi come un modulo fondamentale per la gestione delle operazioni di business, fornendo strutture astratte e riutilizzabili per l'implementazione di handler specifici e coerenti con l'architettura complessiva della solution.

3.2.2.1 Namespace principali

Il progetto è organizzato in namespace distinti, ciascuno con una responsabilità precisa:

- *TM.CQRS._Base.Commands*: definisce le classi astratte di base per i comandi, quali *CommandBase* e *CommandBase<T>*, e le interfacce *ICommand* e *ICommand<T>*. Queste classi forniscono identificatori univoci e contratti comuni per la gestione delle operazioni di scrittura.

- *TM.CQRS._Base.Queries*: include le classi base per le query, come *QueryBase<T>*, e l'interfaccia *IQuery<T>*, garantendo uniformità e coerenza nelle operazioni di lettura dei dati.

- *TM.CQRS._Base.Handlers*: contiene le classi astratte per gli handler di comandi e query, tra cui *CommandHandlerBase<TCommand, TResponse>* e *QueryHandlerBase<TQuery, TResponse>*. Tali classi astratte implementano l'interfaccia *IRequestHandler* di *MediatR*, standardizzando la logica di gestione delle richieste.

- Namespace specifici di dominio: eventuali altri namespace possono essere dedicati all'implementazione concreta dei comandi, query e handler specifici per ciascun ambito funzionale della solution.

3.2.2.2 Classi rilevanti

Le principali classi che caratterizzano *TM.CQRS* includono:

- *CommandBase* / *CommandBase<T>*: classi astratte che rappresentano i comandi, fornendo identificatori univoci e struttura di base per l'estensione.

- *QueryBase<T>*: classe astratta per le query, analogamente ai comandi, garantendo identificatori univoci e contratti comuni.

- *CommandHandlerBase<TCommand, TResponse>* e *QueryHandlerBase<TQuery, TResponse>*: classi base per la gestione delle richieste di scrittura e lettura, integrando *MediatR* per orchestrare l'invocazione degli handler.

- *ICommand*, *ICommand<T>*, *IQuery<T>*: interfacce che tipizzano le operazioni, assicurando una gestione coerente dei comandi e delle query all'interno del sistema.

3.2.2.3 Dipendenze

TM.CQRS si integra con diverse dipendenze esterne e interne alla solution:

- *MediatR*: libreria che supporta il pattern *Mediator*, utilizzata per la gestione asincrona e orchestrata dei comandi e delle query.
- *TM.Shared*: fornisce modelli e DTO condivisi, fondamentali per garantire coerenza e riusabilità tra i vari progetti della solution.
- Eventuali riferimenti a *TM.Data* o *TM.Function*, che consentono agli handler di interagire con la persistenza dei dati e con la logica applicativa.

3.2.2.4 Ruolo nella solution

TM.CQRS ha un ruolo centrale all'interno della solution *TicketManagement*, in quanto definisce la struttura per l'implementazione dei comandi e delle query, favorendo la separazione netta delle responsabilità. Grazie a questo modulo, *TM.Function* può orchestrare le operazioni di business in modo chiaro e modulare, mentre altri progetti possono fare affidamento su una gestione consistente delle richieste, senza introdurre accoppiamenti indesiderati. La sua architettura consente inoltre di implementare handler specifici per ogni tipo di operazione, migliorando la scalabilità del codice e la qualità complessiva della soluzione.

3.2.2.5 Sintesi architetturale

TM.CQRS si configura come il cuore della logica di orchestrazione della solution:

- abilita una gestione chiara e separata di comandi e query,
- si integra con *MediatR* e con i modelli condivisi di *TM.Shared*,
- permette l'implementazione di handler specifici per ciascuna operazione, migliorando manutenibilità, testabilità e qualità complessiva del codice.

3.2.3 TM.Data

Il progetto *TM.Data* rappresenta il layer di accesso ai dati della soluzione *TicketManagement* ed è responsabile della gestione della persistenza delle entità del dominio, nonché dell'interazione con il database attraverso *Entity Framework Core* (EF Core). *TM.Data* fornisce contesti dedicati alle operazioni di lettura e scrittura, repository generici e classi di supporto per migrazioni e snapshot del modello dati, consentendo una gestione efficiente, coerente e scalabile dei dati. Il progetto costituisce quindi un elemento chiave della soluzione, fungendo da base per l'implementazione della logica di business e per l'integrazione con il pattern *CQRS* implementato in *TM.CQRS*.

3.2.3.1 Namespace principali

La struttura di *TM.Data* è organizzata in namespace distinti, che separano chiaramente le responsabilità:

- *TM.Data.Contexts*: include i contesti EF Core utilizzati per gestire la persistenza delle entità. In particolare, *WriteDbContext* è dedicato alle operazioni di scrittura e gestisce le relazioni tra le entità principali del sistema (*User*, *Ticket*, *Task*, *Category*, *Comment*), mentre *ReadDbContext* è ottimizzato per le operazioni di lettura, con configurazioni che disabilitano il tracking e il lazy loading per migliorare le performance.

- *TM.Data._Base*: definisce interfacce generiche e classi base per i repository, come *IRepositoryWrite<T>*, fornendo un contratto standard per le operazioni CRUD e garantendo coerenza e riusabilità del codice di accesso ai dati.

- *TM.Data.Migrations*: gestisce le migrazioni EF Core e la snapshot del modello, permettendo di mantenere allineato lo schema del database con l'evoluzione del modello di dominio.

3.2.3.2 Classi rilevanti

Le principali classi che caratterizzano *TM.Data* includono:

- *WriteDbContext*: contesto EF Core per operazioni di scrittura, gestisce le entità principali del dominio e le relazioni tra di esse. Supporta transazioni e configurazioni per garantire integrità dei dati durante le modifiche.
- *ReadDbContext*: contesto ottimizzato per le operazioni di lettura, progettato per massimizzare le prestazioni mediante il disabilitamento del tracking e l'uso di query read-only.
- *IRepositoryWrite<T>*: interfaccia generica per le operazioni CRUD su entità, che fornisce una base astratta e standardizzata per l'implementazione di repository concreti.
- Migrazioni e *ModelSnapshot*: classi generate automaticamente da EF Core, essenziali per la gestione dello schema del database e per il controllo delle modifiche nel tempo.

3.2.3.3 Dipendenze

TM.Data integra diverse dipendenze chiave:

- *Entity Framework Core*: ORM utilizzato per modellare le entità, gestire le relazioni e orchestrare le operazioni di lettura e scrittura sul database.
- *TM.Shared*: utilizzato per condividere modelli e DTO tra i vari progetti della solution, garantendo coerenza tra layer.
- *TM.CQRS*: referenziato per l'integrazione con la logica CQRS, permettendo agli handler di leggere e scrivere i dati tramite i contesti e i repository definiti.

3.2.3.4 Ruolo nella solution

Il progetto *TM.Data* costituisce la componente centrale per la gestione della persistenza e dell'accesso ai dati. Fornisce strutture e strumenti per separare in maniera efficiente le operazioni di lettura da quelle di scrittura, ottimizzando le performance complessive del sistema. Funziona come base per la logica di business implementata in *TM.Function* e per l'orchestrazione delle operazioni mediante il pattern *CQRS* in *TM.CQRS*. La sua architettura modulare consente di garantire coerenza e integrità dei dati, facilitando al contempo l'estensione e la manutenzione della solution.

3.2.3.5 Sintesi architetturale

TM.Data centralizza la gestione della persistenza tramite contesti EF Core e repository generici, esponendo le entità e le relazioni del dominio *TicketManagement*. Grazie a questa organizzazione, ogni operazione che richiede accesso al database può essere realizzata in modo coerente e scalabile. *TM.Data* costituisce quindi un componente imprescindibile per tutte le funzionalità della solution che interagiscono con i dati, assicurando robustezza, manutenibilità e supporto a pattern architetturali avanzati come *CQRS*.

3.2.4 TM.Function

Il progetto *TM.Function* costituisce il backend serverless della soluzione *TicketManagement*, orchestrando la logica di business tramite l'utilizzo di *Azure Functions* e strumenti di messaggistica asincrona. La sua funzione principale è gestire le operazioni CRUD e altre funzionalità relative alle entità del dominio (ticket, task, commenti, categorie e utenti), fungendo da ponte tra il front-end (*TM.Client*) e il layer di persistenza dei dati (*TM.Data*). La progettazione modulare e l'integrazione con il pattern *CQRS* ne garantiscono scalabilità, manutenibilità e possibilità di estensione in ambienti cloud.

3.2.4.1 Namespace principali

TM.Function organizza le proprie responsabilità in namespace distinti:

- *TM.Function*: contiene le *Azure Functions* e i tool *MCP*, fornendo implementazioni concrete per la gestione delle entità del dominio. Questo namespace definisce endpoint serverless pronti per essere consumati dal client o da altri servizi.
- *TM.Function._Base*: include classi base e interfacce generiche, come *FunctionBase*, *CreateEntityFunctionBase* e *IFunction*, che stabiliscono contratti di esecuzione e forniscono strumenti comuni per la gestione delle funzioni. Questo namespace promuove la riusabilità e l'astrazione della logica di business.

3.2.4.2 Classi rilevanti

Tra le classi principali si distinguono:

- *TicketManagementTools*: espone funzioni *MCP* per la gestione delle entità principali (*Ticket*, *Task*, *Comment*, *Category*, *User*), utilizzando store in-memory a scopo dimostrativo.
- *FunctionBase* / *FunctionBase<T>*: classi base astratte per tutte le funzioni, definiscono identificatori univoci e contratti standard di esecuzione, consentendo uniformità nella gestione delle operazioni.
- *IFunction* / *IFunction<T>*: interfacce per la definizione di funzioni generiche e tipizzate, che supportano la coerenza tra implementazioni concrete.
- *CreateEntityFunctionBase<TEntity, TDtoCreate>*: classe base specializzata per la creazione di entità, che gestisce la deserializzazione dei dati, la validazione, l'invio dei comandi tramite *MediatR* e la comunicazione asincrona tramite *Service Bus*.

3.2.4.3 Dipendenze

TM.Function si integra con numerose librerie e progetti della solution:

- *TM.Shared*: per modelli, DTO e schemi condivisi tra i vari progetti, garantendo consistenza dei dati.
- *TM.CQRS*: per la gestione decoupled dei comandi e delle query, implementando il pattern *CQRS*.
- *Azure Functions Worker*: framework per la definizione e l'hosting delle *Azure Functions*, che consente l'esecuzione serverless.
- *MediatR*: utilizzato per l'orchestrazione dei comandi e delle query in modo modulare e disaccoppiato.
- *Azure.Messaging.ServiceBus*: per la comunicazione asincrona tra funzioni e servizi.
- *Castle.Core.Logging*: per la gestione del logging e del tracciamento delle operazioni.

3.2.4.4 Ruolo nella solution

TM.Function rappresenta il punto centrale per la logica di business della soluzione *TicketManagement*. Consente l'integrazione tra il front-end e il layer di persistenza, orchestrando le operazioni tramite *Azure Functions*, implementando *CQRS* e garantendo un'architettura modulare e scalabile. Le funzionalità serverless e la possibilità di estendere il sistema con nuovi endpoint e tool *MCP* rendono *TM.Function* un componente fondamentale per la gestione delle operazioni automatizzate della piattaforma.

3.2.4.5 Sintesi architetturale

In sintesi, *TM.Function* centralizza l'orchestrazione delle operazioni e la logica di business, integrandosi con:

- *TM.Shared*, per modelli e DTO;
- *TM.CQRS*, per la gestione dei comandi e delle query;
- *TM.Data*, per la persistenza dei dati.

La separazione tra logica di presentazione, persistenza e orchestrazione serverless garantisce modularità, manutenibilità e scalabilità, rendendo *TM.Function* un elemento cruciale della soluzione *TicketManagement*.

3.2.5 TM.Shared

Il progetto *TM.Shared* costituisce il modello di dominio centrale della soluzione *TicketManagement*, fornendo le entità, le enumerazioni e le classi base condivise da tutti gli altri progetti della solution. La sua funzione principale è definire le strutture dati e i contratti comuni utilizzati per garantire coerenza, integrità e riusabilità dei dati tra persistenza, logica di business, API e interfaccia utente. *TM.Shared* non dipende da librerie esterne, essendo una libreria di classi standalone, e rappresenta pertanto la base fondamentale su cui si innestano tutti i layer applicativi.

3.2.5.1 Namespace principali

La struttura di *TM.Shared* è organizzata in namespace distinti:

- *TM.Shared.Models*: include le entità di dominio principali, quali *User*, *Ticket*, *Task*, *Comment* e *Category*, insieme alle enumerazioni *TicketStatus*, *TicketPriority* e *TaskStatus*. Questo namespace definisce le proprietà, le relazioni e le regole principali del dominio.

- *TM.Shared._Base.Models* (presunto): contiene classi base, come *EntityBase*, da cui derivano le entità principali, promuovendo uniformità e coerenza nella definizione dei modelli.

3.2.5.2 Classi rilevanti

Tra le classi più significative di *TM.Shared* si evidenziano:

- *User*: rappresenta un utente del sistema, definendo proprietà identificative e relazioni con i ticket, i task e i commenti creati o assegnati.
- *Ticket*: modella una richiesta o un incidente, con dettagli, stato, priorità e relazioni verso utente creatore, categoria, task e commenti associati.
- *Task*: rappresenta un'attività collegata a un ticket, assegnata a un utente, con proprietà che ne definiscono lo stato e la descrizione.
- *Comment*: gestisce i commenti associati ai ticket, definendo autore, contenuto e data di creazione.
- *Category*: definisce la categoria di un ticket, con nome, descrizione e collezione di ticket associati.
- Enumerazioni: *TicketStatus* (Open, InProgress, Closed), *TicketPriority* (Low, Medium, High) e *TaskStatus* (Pending, InProgress, Completed), fondamentali per modellare lo stato e la priorità delle entità.

3.2.5.3 Dipendenze

TM.Shared non presenta dipendenze esterne, essendo progettato come libreria autonoma. Viene utilizzato come base comune da *TM.Data*, *TM.Function*, *TM.CQRS* e *TM.Client*, garantendo uniformità nella definizione dei modelli e integrità dei dati tra i vari layer della solution.

3.2.5.4 Ruolo nella solution

TM.Shared funge da contratto comune per tutti i progetti della solution *TicketManagement*. Garantisce coerenza semantica e riusabilità dei dati, rappresentando il punto di riferimento per la definizione delle entità e delle loro relazioni. Tutti i layer applicativi, dalla persistenza alla logica di business, dalle API al client, si basano sui modelli definiti in *TM.Shared*, rendendo il progetto imprescindibile per la manutenzione e l'estensione della piattaforma.

3.2.5.5 Sintesi architetturale

In sintesi, *TM.Shared* costituisce la base dati e il contratto di dominio della solution *TicketManagement*:

- espone entità e enumerazioni fondamentali,
- non richiede dipendenze esterne,
- viene referenziato da *TM.Data*, *TM.Function*, *TM.CQRS* e *TM.Client* per garantire coerenza e integrità dei dati in tutto il sistema.

3.3 Conclusione

Il capitolo ha illustrato in dettaglio la composizione della solution *TicketManagement*, evidenziando i singoli progetti che ne costituiscono l'architettura modulare e le rispettive responsabilità. La descrizione dei progetti *TM.Shared*, *TM.Data*, *TM.CQRS*, *TM.Function* e *TM.Client* ha messo in luce come ogni componente contribuisca in maniera specifica al funzionamento complessivo del sistema, rispettando i principi di modularità, coesione e separazione delle responsabilità.
In particolare, *TM.Shared* definisce il modello di dominio comune, garantendo coerenza tra tutti i layer; *TM.Data* centralizza la gestione della persistenza e delle operazioni sul database; *TM.CQRS* implementa il pattern *Command Query Responsibility Segregation*, separando logicamente le operazioni di lettura da quelle di

scrittura; *TM.Function* orchestrando la logica di business tramite *Azure Functions*, funge da ponte tra il front-end e il backend; infine, *TM.Client* offre l'interfaccia utente, consentendo l'interazione con le funzionalità del sistema.
L'analisi dei namespace, delle classi principali e delle dipendenze interne ed esterne ha permesso di comprendere non solo il ruolo di ciascun progetto, ma anche le interazioni e le relazioni tra essi, che costituiscono una rete coerente e scalabile. Tale organizzazione favorisce la manutenibilità, l'estendibilità e la qualità del codice, elementi fondamentali per sistemi complessi come quelli destinati alla gestione di ticket, workflow e processi aziendali.
In sintesi, la soluzione *TicketManagement* rappresenta un esempio di architettura modulare e stratificata, in cui ogni progetto contribuisce a un ecosistema integrato, garantendo uniformità dei dati, orchestrazione delle operazioni, separazione dei compiti e scalabilità complessiva del sistema. Questa struttura architetturale costituirà la base per l'analisi delle funzionalità e dei pattern implementati nei capitoli successivi.

Conclusioni

Ringraziamenti

Desidero esprimere la mia più sincera gratitudine ai docenti, per la dedizione e la passione con cui hanno trasmesso le loro conoscenze, contribuendo in modo fondamentale alla mia crescita accademica; ai colleghi in azienda, per la professionalità e disponibilità con la quale mi hanno accolto, guidato e supportato durante il mio percorso di tirocinio, permettendomi di mettere in pratica quanto appreso e di acquisire nuove competenze; ai miei amici e colleghi di università, per il costante sostegno, la collaborazione e la condivisione di momenti indimenticabili, che hanno reso questo percorso più ricco e stimolante; ed infine alla mia famiglia, per il supporto incondizionato che mi hanno sempre dimostrato e senza la quale nulla di tutto questo sarebbe stato possibile.

Bibliografia

[1] UBS. Latest house view daily, 2023. URL: https://www.ubs.com/global/en/wealthmanagement/insights/chief-investment-office/house-view/daily/2023/latest-25052023.html. Citato a pagina 3.

[2] Model Context Protocol. Getting started - introduction, 2023. URL: https://modelcontextprotocol.io/docs/getting-started/intro. Citato a pagina 4.

[3] Adam Kilgarriff and Gregory Grefenstette. Introduction to the special issue on the web as corpus. *Computational Linguistics*, 29(3):333–347, 2003. doi:10.1162/089120103322711569. Citato a pagina 5.

[4] Michele Banko and Eric Brill. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 26–33. Association for Computational Linguistics, 2001. doi:10.3115/1073012.1073017. Citato a pagina 5.

[5] Philip Resnik and Noah A. Smith. The web as a parallel corpus. *Computational Linguistics*, 29(3):349–380, 2003. Archived from the original on 2024-06-07. Retrieved 2024-06-07. doi:10.1162/089120103322711578. Citato a pagina 5.

[6] Joshua Goodman. A bit of progress in language modeling. *arXiv preprint*, 2001. URL: https://arxiv.org/abs/cs/0108005, arXiv:cs/0108005. Citato a pagina 5.

[7] Wei Xu and Alex Rudnicky. Can artificial neural networks learn language models? In *6th International Conference on Spoken Language Processing (ICSLP 2000)*, volume 1. ISCA, 2000. doi:10.21437/icslp.2000-50. Citato a pagina 5.

[8] Leiyu Chen, Shaobo Li, Qiang Bai, Jing Yang, Sanlong Jiang, and Yanming Miao. Review of image classification algorithms based on convolutional neural networks. *Remote Sensing*, 13(22):4712, 2021. doi:10.3390/rs13224712. Citato a pagina 5.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. Archived (PDF) from the original on 2024-02-21. Retrieved 2024-01-21. URL: https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf. Citato a pagina 6.

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint*, 2014. URL: https://arxiv.org/abs/1409.0473, arXiv:1409.0473. Citato a pagina 6.

[11] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics*, 8:842–866, 2020. Archived from the original on 2022-04-03. Retrieved 2024-01-21. arXiv:2002.12327, doi:10.1162/tacl_a_00349. Citato a pagina 6.

[12] Rajiv Movva, Sidhika Balachandar, Kenny Peng, Gabriel Agostini, Nikhil Garg, and Emma Pierson. Topics, authors, and institutions in large language model research: Trends from 17k arxiv papers. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Com-*

putational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 1223–1243, 2024. Retrieved 2024-12-08. arXiv:2307.10700, doi:10.18653/v1/2024.naacl-long.67. Citato a pagina 6.

[13] Alex Hern. New ai fake text generator may be too dangerous to release, say creators. *The Guardian*, February 2019. Archived from the original on 14 February 2019. Retrieved 20 January 2024. URL: https://web.archive.org/web/20190214173112/https://www.theguardian.com/technology/2019/feb/14/elon-musk-backed-ai-writes-convincing-news-fiction. Citato a pagina 6.

[14] Euronews. Chatgpt a year on: 3 ways the ai chatbot has completely changed the world in 12 months. *Euronews*, November 2023. Archived from the original on January 14, 2024. Retrieved January 20, 2024. URL: https://www.euronews.com/next/2023/11/30/chatgpt-a-year-on-3-ways-the-ai-chatbot-has-completely-changed-the-world-in-12-months. Citato a pagina 6.

[15] Will Heaven. Gpt-4 is bigger and better than chatgpt—but openai won't say why. *MIT Technology Review*, March 2023. Archived from the original on March 17, 2023. Retrieved January 20, 2024. URL: https://web.archive.org/web/20230317224201/https://www.technologyreview.com/2023/03/14/1069823/gpt-4-is-bigger-and-better-chatgpt-openai/. Citato a pagina 6.

[16] Cade Metz. Openai unveils new chatgpt that can reason through math and science. *The New York Times*, September 2024. Retrieved September 12, 2024. URL: https://www.nytimes.com/2024/09/12/technology/openai-chatgpt-math.html. Citato a pagina 6.

[17] Parameters in notable artificial intelligence systems, November 2023. Retrieved January 20, 2024. URL: https://ourworldindata.org/grapher/artificia

l-intelligence-parameter-count?time=2017-09-05..latest. Citato a pagina 6.

[18] Shubham Sharma. Open-source deepseek-r1 uses pure reinforcement learning to match openai o1 — at 95% less cost. *VentureBeat*, January 2025. Retrieved 2025-01-26. URL: https://venturebeat.com/ai/open-source-deepseek-r1-uses-pure-reinforcement-learning-to-match-openai-o1-at-95-less-cost/. Citato a pagina 7.

[19] Dr Tehseen Zia. Unveiling of large multimodal models: Shaping the landscape of language models in 2024. *Unite.AI*, January 2024. Retrieved 2024-12-28. URL: https://www.unite.ai/unveiling-of-large-multimodal-models-shaping-the-landscape-of-language-models-in-2024/. Citato a pagina 7.

[20] Rick Merritt. What is a transformer model? NVIDIA Blog, March 2022. Archived from the original on 2023-11-17. Retrieved 2023-07-25. URL: https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/. Citato a pagina 7.

[21] Model Context Protocol. Getting started - introduction, 2023. Accessed: 2025-08-16. URL: https://modelcontextprotocol.io/docs/getting-started/intro. Citato a pagina 10.

[22] Model Context Protocol. Learn - architecture, 2023. Accessed: 2025-08-16. URL: https://modelcontextprotocol.io/docs/learn/architecture. Citato alle pagine 11, 12, 13, 14, and 15.