UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica e dell'Automazione



TESI DI LAUREA

Progettazione e implementazione di case study relativi alla generazione di codice e di immagini mediante l'utilizzo di LLM differenti

Design and implementation of case studies related to code and image generation through the usage of different LLMs

Relatore

Prof. Domenico Ursino

Candidato

Tarek Naja

ANNO ACCADEMICO 2025-2026

Come raggiungere un traguardo?
Senza fretta ma senza sosta.

Goethe

Sommario

Negli ultimi anni, l'Intelligenza Artificiale Generativa ha assunto un'importanza sempre maggiore, grazie alla sua capacità di produrre contenuti complessi, in tempi estremamente brevi, partendo da richieste formulate in linguaggio naturale. Nella presente tesi vengono analizzati e confrontati quattro diversi LLM, valutandone le specifiche capacità operative nella generazione di codice e di immagini. In particolare, la valutazione viene condotta sottoponendo le medesime richieste in *input* alle diverse architetture, adattando, dove necessario, le istruzioni tramite *prompt* mirati per mantenere la coerenza tra gli *output* restituiti. Questo approccio ha garantito un confronto trasparente e oggettivo tra le prestazioni dei vari modelli, permettendo quindi di evidenziarne le principali differenze architetturali e i rispettivi punti di forza.

Keyword: Intelligenza Artificiale Generativa, Gemini, ChatGPT, Copilot, Claude, *Case study*, *Prompt*, Generazione di codice e immagini

Indice

Introduzione **1**

1 Introduzione all'Intelligenza Artificiale Generativa **3**
1.1 Definizione di Intelligenza Artificiale . . . 3
1.2 Dal *Machine Learning* ai Modelli Generativi . . . 3
1.2.1 *Machine Learning* . . . 4
1.2.2 *Deep Learning* . . . 4
1.2.3 IA Generativa . . . 5
1.3 Funzionamento dell'IA Generativa . . . 6
1.3.1 Addestramento . . . 6
1.3.2 Ottimizzazione . . . 6
1.3.3 Generazione, Valutazione e Ulteriore Messa a Punto . . . 7
1.4 Vantaggi dell'IA e Casi d'Uso dell'IA Generativa . . . 7
1.4.1 Principali Vantaggi dell'IA . . . 7
1.4.2 Casi d'Uso Reali dell'IA Generativa . . . 8
1.5 Rischi, Considerazioni Etiche e Responsabilità . . . 10
1.5.1 Rischi Relativi ai Dati e ai Modelli . . . 10
1.5.2 Considerazioni Etiche e Responsabilità Sociali dell'IA Generativa . . . 11
1.5.3 Sostenibilità e Impatto Ambientale . . . 11
1.6 Il Futuro dell'IA Generativa . . . 12

2 Presentazione degli LLM in Esame **13**
2.1 Definizione dei Parametri negli LLM . . . 13
2.1.1 Parametri Addestrabili: Pesi e *Bias* . . . 13
2.1.2 Gli Iperparametri . . . 14
2.2 Classificazione degli LLM: Modelli Proprietari e *Open Source* . . . 15
2.2.1 Modelli Proprietari . . . 15
2.2.2 Modelli *Open Source* . . . 15
2.2.3 Criteri di Valutazione e Selezione . . . 16
2.2.4 Strumenti di Valutazione . . . 16
2.3 Discussione dei Modelli in Esame . . . 17
2.3.1 Gemini . . . 17
2.3.2 ChatGPT . . . 19
2.3.3 Copilot . . . 21
2.3.4 Claude . . . 23

3 Un *Case Study* per Confrontare Diversi LLM nella Generazione di Codice 25
3.1 Definizione del *Case Study* . . . 25
3.2 Analisi dei Modelli . . . 26
3.2.1 Generazione con Gemini . . . 26
3.2.2 Generazione con ChatGPT . . . 37
3.2.3 Generazione con Copilot . . . 48
3.2.4 Generazione con Claude . . . 56
3.3 Valutazione dei Modelli . . . 65

4 Un *Case Study* per Confrontare Diversi LLM nella Generazione di Immagini 66
4.1 Definizione del *Case Study* . . . 66
4.2 Analisi dei Modelli . . . 66
4.2.1 Generazione con Gemini . . . 67
4.2.2 Generazione con ChatGPT . . . 71
4.2.3 Generazione con Copilot . . . 74
4.2.4 Generazione con Claude . . . 81
4.3 Valutazione dei Modelli . . . 83

5 Discussione 84
5.1 Discussione Relativa alla Generazione di Codice . . . 84
5.2 Discussione Relativa alla Generazione di Immagini . . . 84

Conclusioni 86

Bibliografia 87

Sitografia 88

Ringraziamenti 91

Elenco delle figure

1.1 Evoluzione Cronologica dell'Intelligenza Artificiale 4
1.2 Funzionamento Modello di Diffusione 5

2.1 Categorizzazione delle Competenze del Modello 16
2.2 Schermata Relativa a una Nuova Chat di Gemini 19
2.3 Schermata Relativa a una Nuova Chat di ChatGPT 21
2.4 Schermata Relativa alla Nuova Chat di Copilot 22
2.5 Schermata Relativa alla Nuova Chat di Claude 24

4.1 *Zoom* per evidenziare il fenomeno della degradazione testuale 68
4.2 *Zoom* per evidenziare il fenomeno della degradazione testuale 71
4.3 *Zoom* per evidenziare il fenomeno della degradazione testuale 73
4.4 *Zoom* per evidenziare il fenomeno della degradazione delle mani 78
4.5 Zoom per evidenziare il fenomeno della degradazione testuale 80

Elenco delle tabelle

3.1 Confronto delle caratteristiche dei modelli LLM 65

Introduzione

Negli ultimi anni l'Intelligenza Artificiale ha assunto ormai una posizione sempre più centrale nella nostra quotidianità. In tale contesto, i sistemi di IA Generativa hanno acquisito un'enorme popolarità, distinguendosi per versatilità e innovazione, oltre che per le capacità creative, la rapidità di esecuzione e le molteplici funzionalità offerte. Questi modelli consentono, infatti, di produrre un'ampia varietà di contenuti, tra cui, ad esempio, testo, immagini, musica e codice informatico, riuscendo così a rivoluzionare profondamente i vari settori professionali.

In questa tesi ci poniamo l'obiettivo di valutare tali capacità operative proponendo due *case study*, uno per la generazione di codice Python per un *quiz* interattivo e animato, curato nei dettagli grafici e funzionali, e l'altro focalizzato sulla creazione di immagini, su quattro diversi modelli: Gemini, ChatGPT, Copilot e Claude. Questo approccio permette non solo di individuare le criticità e i punti di forza di ciascun sistema, ma anche di realizzare un confronto strutturato tra gli LLM oggi più utilizzati. L'obiettivo finale è, dunque, comprendere come integrare efficacemente tali strumenti nelle attività lavorative e quotidiane, riconoscendo i loro limiti e dimostrando come la supervisione sia ancora un elemento fondamentale per garantire *output* di qualità. Tale supervisione si concretizza nel *Prompt Engineering*. Durante l'analisi verrà, infatti, evidenziato come la stesura di istruzioni chiare e contestualizzate sia lo strumento principale per guidare i modelli verso risposte affidabili. Nella maggior parte dei casi il primo *output* generato rappresenta soltanto la base di partenza e non la versione definitiva. Il processo richiede continue iterazioni in cui il *prompt* iniziale viene affinato progressivamente per correggere le criticità.

Questa tesi è strutturata come di seguito specificato:

- Nel Capitolo 1 verrà introdotto il concetto di Intelligenza Artificiale Generativa, partendo dalla sua definizione e dal suo sviluppo storico, per poi analizzarne il funzionamento tecnico, i principali vantaggi e i potenziali rischi.
- Nel Capitolo 2 verrà introdotta la definizione di parametro per gli LLM, procedendo, poi, con la classificazione tra i diversi modelli. Infine, verranno presentate nel dettaglio le architetture in esame.
- Nel Capitolo 3 verrà affrontato il *case study* relativo alla generazione di codice. Dopo un'iniziale presentazione dell'obiettivo della prova, saranno riportati e confrontati i risultati ottenuti dai quattro LLM.

- Nel Capitolo 4 verrà analizzato il *case study* relativo alla creazione di immagini. A seguito di una breve introduzione sugli obiettivi del *test*, verranno esaminati e confrontati gli output prodotti dai quattro modelli.
- Nel Capitolo 5 verranno discussi i risultati complessivi ottenuti dai *test* condotti, evidenziando i punti di forza e le principali limitazioni emerse per ciascun modello.

CAPITOLO 1

Introduzione all'Intelligenza Artificiale Generativa

Nel presente capitolo viene analizzata nel dettaglio la disciplina dell'Intelligenza Artificiale Generativa. L'analisi partirà dalla sua definizione concettuale e dallo sviluppo storico che ne ha caratterizzato l'evoluzione, per poi approfondirne il funzionamento. Successivamente, verranno descritti i principali vantaggi e i molteplici rischi legati all'impiego di tali sistemi, affrontando sia le criticità di sicurezza sia le questioni etiche e normative. Infine, il capitolo si concluderà esplorando le prospettive di sviluppo futuro e l'impatto che i nuovi modelli avranno sul piano sociale e lavorativo.

1.1 Definizione di Intelligenza Artificiale

> "L'Intelligenza Artificiale (IA) è l'abilità di una macchina di mostrare capacità umane quali il ragionamento, l'apprendimento, la pianificazione e la creatività."

Questa è la definizione che il Parlamento Europeo attribuisce a tale tecnologia. La peculiarità di questi sistemi risiede proprio nella capacità di operare in autonomia, riadattando il proprio comportamento in base all'analisi delle azioni passate. Le applicazioni e i dispositivi dotati di queste innovazioni sono in grado di comprendere e rispondere al linguaggio umano, oltre a imparare da nuove informazioni ed esperienze. L'IA, oramai, è parte integrante della vita di tutti i giorni. Basti pensare ai software di traduzione automatica che analizzano testi e audio per fornire conversioni linguistiche sempre migliori, o alla comodità dei sottotitoli generati in tempo reale per i contenuti multimediali. Inoltre, l'impatto di questi sistemi è evidente nel settore automobilistico, dove i sensori di bordo sono già in grado di riconoscere i pericoli alla guida e costituiscono la base per i nascenti veicoli a conduzione autonoma.

1.2 Dal *Machine Learning* ai Modelli Generativi

L'evoluzione dell'Intelligenza Artificiale può essere rappresentata dallo sviluppo di molteplici sottodiscipline nell'arco degli ultimi settanta anni. Nello specifico, questo percorso parte dai primi approcci al *Machine Learning*, fino ad arrivare al *Deep Learning* e alle attuali applicazioni di IA Generativa. Come mostrato nella Figura 1.1, questo processo evidenzia una crescente capacità di autonomia dei sistemi.

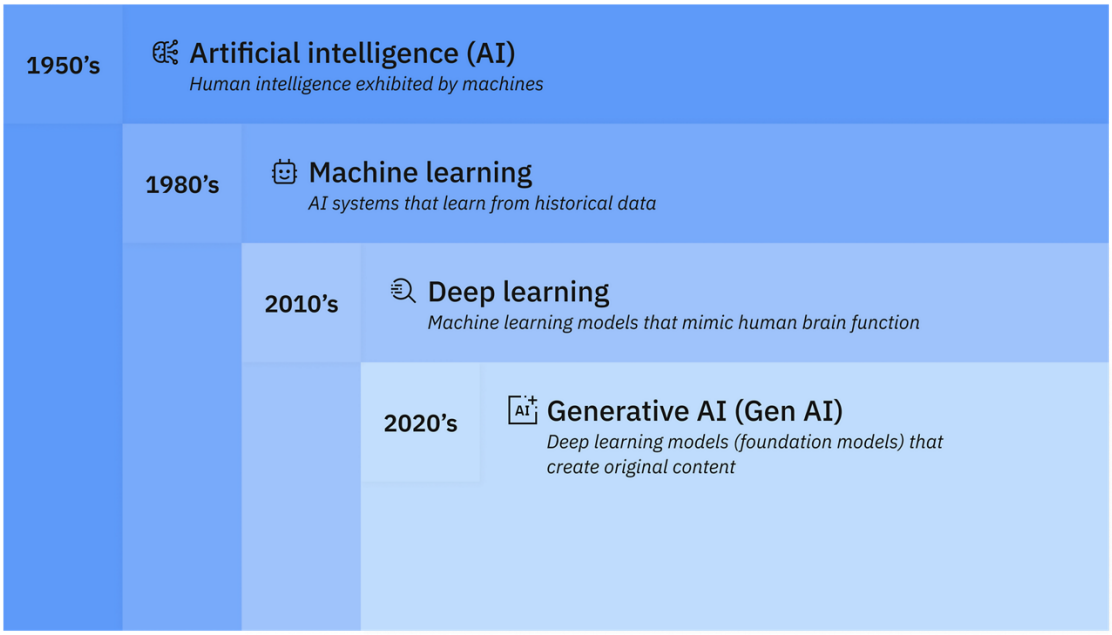


Figura 1.1: Evoluzione Cronologica dell'Intelligenza Artificiale

1.2.1 *Machine Learning*

La prima branca di specializzazione dell'Intelligenza Artificiale è rappresentata dal Machine Learning, disciplina che consente agli algoritmi di individuare schemi ricorrenti all'interno di set di dati. Questi modelli sono in grado di elaborare nuovi dati in ingresso per effettuare previsioni e restituire un output accurato, senza la necessità di una programmazione esplicita per il compito specifico da svolgere. Esistono innumerevoli algoritmi, ciascuno specificamente calibrato per affrontare diverse tipologie di problemi e di dataset. Nel corso del tempo sono state sviluppate svariate architetture, tra le quali si distinguono per efficacia e versatilità, le reti neurali artificiali. Queste ultime, essendo costituite da molteplici strati di nodi interconnessi che collaborano per elaborare informazioni strutturate, si rivelano particolarmente adatte per tutte quelle attività che richiedono l'identificazione di *pattern* e di relazioni non lineari all'interno di vaste moli di dati.

1.2.2 *Deep Learning*

Il Deep Learning costituisce un sottoinsieme del Machine Learning basato su reti neurali profonde, ciascuna delle quali esegue operazioni matematiche. Queste reti includono un livello di *input*, almeno tre livelli nascosti (ma solitamente anche centinaia) e un livello di *output*. Utilizzando il Machine Learning per regolare l'intensità delle connessioni tra le singole unità computazionali in livelli adiacenti, la rete può essere ottimizzata per produrre output più accurati. Le reti neurali sono definite come approssimatori universali, in quanto è stato dimostrato a livello teorico che è possibile riprodurre qualsiasi funzione tramite una determinata disposizione di reti neurali. I modelli di Deep Learning vengono solitamente addestrati tramite apprendimento non supervisionato. I livelli multipli delle reti neurali sviluppate nel Deep Learning consentono l'apprendimento non supervisionato automatizzando l'estrazione delle caratteristiche da set di dati di grandi dimensioni, non etichettati e non strutturati, facendo previsioni sul loro significato. Tale approccio è emerso come una modalità importante di addestramento delle reti neurali, in particolare per i *foundation model* dell'IA Generativa.

1.2.3 IA Generativa

L'IA Generativa si riferisce a modelli di Deep Learning che, differentemente dai sistemi tradizionali usati per classificazioni o previsioni, sono capaci di generare contenuti originali complessi sulla base di schemi appresi da enormi quantità di dati di addestramento. Tali algoritmi sono stati utilizzati per anni nel campo della statistica per analizzare i dati numerici. Inoltre, nell'ultimo decennio si sono evoluti per esaminare e generare tipologie di dati più complesse, come immagini dettagliate o stringhe di codice. Questo sviluppo coincide con l'emergere di alcuni sofisticati sistemi di Deep Learning.

Autoencoder Variazionali (VAE)

Gli autoencoder variazionali, introdotti nel 2013, hanno permesso la creazione di modelli capaci di generare molteplici versioni di contenuto in risposta a un *prompt* o a un'istruzione.

Modelli di Diffusione

I modelli di diffusione, proposti nel 2014, si basano su un meccanismo che altera le immagini aggiungendo progressivamente rumore gaussiano fino a renderle irriconoscibili. Attraverso un processo inverso, il sistema procede successivamente a rimuovere tale rumore, riuscendo, così, a generare immagini del tutto nuove in base agli *input* forniti dall'utente.

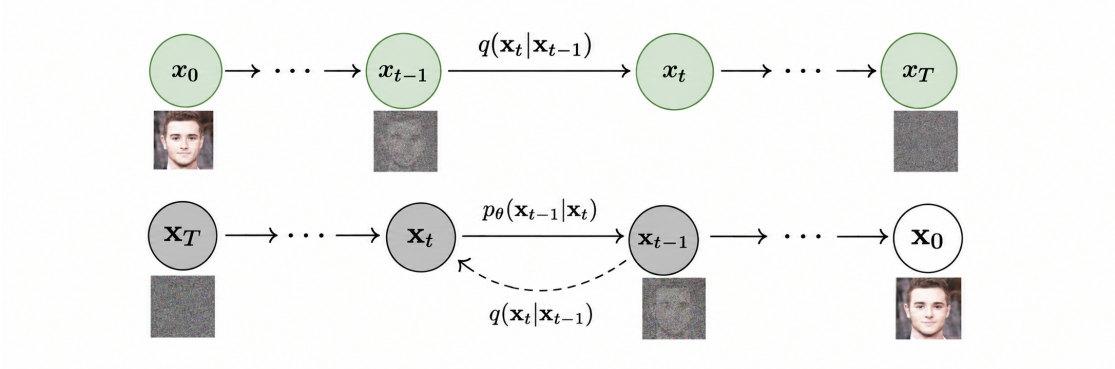


Figura 1.2: Funzionamento Modello di Diffusione

Reti Generative Avversarie (GAN)

Un'altra innovazione fondamentale del 2014 è rappresentata dalle Reti Generative Avversarie (GAN), introdotte da Ian Goodfellow con l'articolo "Generative Adversarial Nets". Una GAN è un'architettura di apprendimento automatico progettata per generare dati sintetici altamente realistici, imparando a replicare gli schemi ricorrenti presenti nei set di dati forniti durante la fase di addestramento. Tale sistema si basa su un framework di Deep Learning, in cui due reti neurali lavorano in opposizione: la prima che genera dati e la seconda deve valutare se questi siano reali o generati.

Transformer

L'architettura dei Transformer, introdotta nel 2017 con l'articolo "Attention is All You Need" di Vaswani *et al.*, ha segnato una vera e propria svolta nel Deep Learning. I Transformer vengono addestrati su dati sequenziali per generare, a loro volta, sequenze di contenuti, quali parole in un testo, forme in un'immagine e stringhe di codice. Questa architettura è alla

base della maggior parte degli attuali strumenti di IA Generativa, come ChatGPT e GPT-4. I modelli appartenenti alla famiglia dei *Large Language Model* (LLM), tra cui la serie GPT, spiccano nella comprensione e nella manipolazione di sequenze di dati, risultando, così, estremamente competenti sia con il linguaggio naturale sia con quello di programmazione. Un LLM elabora il *prompt* dell'utente formulando una risposta basata sulla sequenza più probabile di parole o token successivi.

1.3 Funzionamento dell'IA Generativa

Il ciclo di vita di un sistema di Intelligenza Artificiale generativa si sviluppa generalmente attorno a tre fasi distinte:

- *addestramento*: creazione del *foundation model* concepito come base generica per diverse applicazioni di IA di nuova generazione;
- *ottimizzazione*: adattamento del modello per eseguire compiti specifici;
- *generazione, valutazione e ulteriore messa a punto*: monitoraggio costante degli output e della loro accuratezza.

1.3.1 Addestramento

I *foundation model* sono modelli di Deep Learning progettati come punto di partenza per diversi tipi di applicazioni di IA generativa. Tra le soluzioni più diffuse spiccano gli LLM, specializzati nella generazione testuale. Tuttavia, esistono anche architetture per la generazione di immagini, audio o video, fino a modelli multimodali in grado di gestire simultaneamente molteplici tipologie di contenuti. Per realizzare un *foundation model*, gli sviluppatori coordinano l'addestramento di un algoritmo di Deep Learning su vaste quantità di dati grezzi, non strutturati e non etichettati, come terabyte o petabyte di testo, immagini o sequenze video raccolti dal web o da altre fonti di dati. Durante la fase di addestramento l'algoritmo esegue e valuta milioni di test, cercando di prevedere l'elemento successivo in una determinata sequenza, come la parola successiva in una frase o l'istruzione successiva in una riga di codice, ed effettua un'autoregolazione per tentare di colmare la distanza tra la previsione effettuata e il dato reale. Come risultato si ottiene una rete neurale di parametri intesi come rappresentazioni codificate di *pattern*, entità e relazioni nei dati, capace di generare contenuti in modo autonomo in risposta ai *prompt* ricevuti. Tale processo di formazione è caratterizzato da un'elevata intensità di calcolo, dovuta all'impiego massiccio di Unità di Elaborazione Grafica (GPU). Queste ultime vengono organizzate in *cluster* (reti di macchine interconnesse) che operano in parallelo per diverse settimane, rendendo l'intera fase estremamente onerosa in termini di tempo e costi. In alternativa, la disponibilità di modelli *open source*, come la famiglia Llama 2 sviluppata da Meta, permette agli sviluppatori di IA Generativa di aggirare il pre-addestramento e i relativi oneri economici.

1.3.2 Ottimizzazione

Il modello ottenuto al termine della fase di addestramento risulta generalista, pur possedendo conoscenze su molteplici contenuti, non è in grado di generare tipi specifici di output con la precisione e la qualità desiderate. Risulterà, pertanto, fondamentale lavorare sull'ottimizzazione di tale sistema focalizzando l'attività di generazione su contenuti specifici. Per la specializzazione si possono seguire diversi approcci:

- Messa a punto (*fine-tuning*): l'algoritmo viene istruito mediante l'importazione di dati etichettati specifici dell'applicazione, cioè di domande o *prompt* tipici che il sistema potrebbe ricevere, e delle relative risposte nel formato desiderato.
- Apprendimento per rinforzo con feedback umano (*RLHF*): il *Reinforcement Learning from Human Feedback* è una metodologia in cui i supervisori umani valutano e ordinano qualitativamente diverse versioni di output generate per uno stesso input. I dati così raccolti consentono di orientare l'IA verso risposte più accurate rispetto agli standard attesi.

1.3.3 Generazione, Valutazione e Ulteriore Messa a Punto

In quest'ultima fase, gli sviluppatori e gli utenti devono occuparsi della regolare valutazione della qualità degli *output* delle applicazioni di IA Generativa, adattando, di conseguenza, il modello. Un'ulteriore tecnica per ottimizzare le prestazioni di un'applicazione di IA è la *Retrieval-Augmented Generation* (RAG). Tale sistema permette l'estensione delle capacità del *foundation model* integrandolo con fonti pertinenti, al di fuori dei dati di addestramento, raggiungendo, così, un livello di precisione e pertinenza ancora più elevato. La RAG garantisce che un'applicazione di IA generativa abbia sempre accesso alle informazioni più aggiornate e contestualizzate. Inoltre, le fonti aggiuntive, a differenza del *dataset* del *foundation model* originale, sono trasparenti e verificabili.

1.4 Vantaggi dell'IA e Casi d'Uso dell'IA Generativa

Secondo il rapporto *The State of AI* di McKinsey, il 65% delle aziende intervistate utilizza regolarmente l'IA Generativa in almeno una funzione aziendale. Non si tratta più solo di test o sperimentazioni. Infatti, numerose organizzazioni registrano già ora benefici concreti, tra cui la diminuzione dei costi operativi e l'aumento dei ricavi nelle unità che la adottano. L'IA generativa risulta attualmente più diffusa in ambiti con impatti immediati, quali marketing, vendite e sviluppo dei prodotti. Simultaneamente, Gartner segnala la nuova evoluzione legata agli agenti di IA. Tali sistemi non si limitano alla generazione di contenuti o all'assistenza dei clienti, ma sono in grado di gestire processi complessi, come il rifornimento, la gestione degli ordini e il supporto alle vendite, operando in maniera autonoma per ridurre i tempi e i margini di errore. Secondo le stime di Gartner, entro il 2028 gli agenti di IA intermedieranno fino al 90% degli acquisti B2B, influenzando transazioni globali di migliaia di miliardi di dollari. Per trarne beneficio, le aziende devono integrare tale tecnologia nei processi strategici, sviluppando competenze interne.

1.4.1 Principali Vantaggi dell'IA

Lo sviluppo dell'IA ha apportato numerosi vantaggi in diversi settori e applicazioni. Alcuni tra i benefici più frequentemente citati includono:

- automazione delle attività ripetitive;
- informazioni più dettagliate e più rapide dai dati;
- miglioramento a livello di processi decisionali;
- meno errori umani;
- disponibilità 24 ore su 24, 7 giorni su 7;

- riduzione dei rischi fisici.

Nel seguito vediamo più in dettaglio ciascuno di questi benefici.

Automazione delle attività ripetitive

L'implementazione di sistemi di IA rende possibile l'automazione di attività ripetitive, affidando un lavoro, solitamente a carico di un operatore, a un software, il quale si occupa di svolgere tale mansione emulando l'essere umano. Lato software, ciò si traduce nell'esecuzione autonoma di attività, quali la raccolta, l'inserimento e l'elaborazione preventiva dei dati; lato hardware, ciò si traduce con l'impiego di *supply chain* auto-ottimizzanti che regolano automaticamente gli inventari. Questo tipo di automazione permette di concentrare l'attenzione su compiti di maggior valore.

Informazioni più dettagliate e più rapide dai dati

Sia che venga utilizzata come supporto sia che venga adottata per un processo decisionale completamente automatizzato, l'IA consente previsioni più rapide e accurate, favorendo decisioni affidabili e basate sui dati. Tale tecnologia permette alle aziende di cogliere le opportunità e rispondere alle crisi nel momento stesso in cui emergono, operando in tempo reale e senza la necessità di intervento umano.

Meno errori umani

L'IA può minimizzare gli errori umani attraverso diverse modalità: guidando gli operatori nelle fasi corrette di un processo, segnalando potenziali anomalie prima che si verifichino o automatizzando completamente le procedure senza alcun intervento umano. Inoltre, gli algoritmi di apprendimento automatico possono migliorare continuamente la loro precisione, riducendo ulteriormente il margine di errore man mano che elaborano nuovi dati e si perfezionano tramite l'esperienza acquisita.

Disponibilità 24 ore su 24, 7 giorni su 7

L'IA garantisce un'operatività continua, mantenendo prestazioni costanti nel tempo. Strumenti come i *chatbot* o gli assistenti virtuali permettono di gestire l'assistenza clienti in qualsiasi momento, riducendo il carico di lavoro assegnato al personale umano. In altri contesti applicativi, quali la lavorazione dei materiali o le linee di produzione, l'IA impiegata per l'esecuzione di attività ripetitive contribuisce a mantenere costanti gli standard qualitativi e i volumi di produzione.

Riduzione dei rischi fisici

Sebbene le relative tecnologie siano ancora in fase di perfezionamento, le auto a guida autonoma e altri mezzi automatizzati presentano il potenziale per ridurre significativamente il rischio di lesioni fisiche per i passeggeri.

1.4.2 Casi d'Uso Reali dell'IA Generativa

La tecnologia dell'IA Generativa presenta un'ampia gamma di applicazioni nel mondo reale, che spaziano dalla generazione di testi e immagini, fino allo sviluppo di software. Di seguito verranno analizzati alcuni dei casi d'uso attuali più diffusi.

Generazione di Immagini

I modelli generativi per le immagini consentono di creare nuovi contenuti visivi originali, come fotografie fotorealistiche in risposta a *prompt* scritti. Tali sistemi trasformano un'istruzione di *input* in linguaggio naturale in una rappresentazione grafica, offrendo un'elevata flessibilità. Infatti, gli utenti possono anche fornire indicazioni generiche, lasciando ampio margine operativo all'algoritmo, oppure essere particolarmente dettagliati nelle richieste. Un'altra tecnica è quella del trasferimento di stile. In tale modello generativo il sistema riceve due *input* distinti, ovvero un'immagine di contenuto, che definisce il soggetto, e un'immagine di stile. L'output finale preserverà la struttura della prima, ma ne rielaborerà i tratti estetici e cromatici adattandoli alle caratteristiche visive della seconda.

Generazione di Testo

Sebbene uno dei casi d'uso dell'IA Generativa basata su testo sia costituito dai *chatbot*, la tecnologia trova sviluppi applicativi in molteplici ambiti lavorativi. I modelli linguistici possono, ad esempio, supportare la stesura di *e-mail* mantenendo un registro professionale. Un altro campo di applicazione è quello della creazione di documenti promozionali per un determinato prodotto. Mentre per tale realizzazione un operatore necessiterebbe di tempi lunghi per poter analizzare le specifiche del prodotto ed elaborare una relativa bozza, un sistema di IA Generativa, una volta ricevute le informazioni, impiega pochi secondi per creare contenuti già pronti all'uso. Tali capacità possono risultare utili anche in ambito cinematografico, fornendo sottotitoli e traduzioni dei contenuti in varie lingue.

Composizione Musicale

L'IA Generativa è in grado di comporre brani musicali inediti, in un genere specifico o emulando lo stile di specifici compositori. Analogamente ai generatori di immagini, i modelli operano prendendo come punto di partenza *prompt* testuali, che possono consistere in richieste dettagliate, specificando, ad esempio, la struttura metrica, il tempo e gli strumenti desiderati; si può anche permettere piena libertà creativa al sistema. È, inoltre, possibile chiedere al sistema di IA di applicare il trasferimento di stile, fondendo due brani diversi. Per fare ciò, si deve fornire all'algoritmo una melodia di base, dalla quale generare un nuovo remix, che riproduca l'impronta stilistica di un altro autore. Mentre alcuni modelli generano tracce partendo da basi esistenti, quelli multimodali più avanzati sono in grado di generare colonne sonore tramite il riconoscimento degli elementi presenti in un filmato.

Dati Sintetici

L'IA generativa è in grado di creare dati sintetici tramite l'utilizzo di algoritmi e simulazioni avanzate. Tali set di dati artificiali replicano fedelmente le proprietà matematiche dei dati reali, senza tuttavia contenere alcuna informazione effettiva. Poiché la raccolta di dati tradizionale può risultare onerosa in termini di tempo e costi, questa tecnologia consente alle organizzazioni di generare volumi scalabili di set di dati strutturati ed etichettati, ottimizzando, così, l'impiego delle risorse. Oltre all'impiego nella *data augmentation* per migliorare l'addestramento dei modelli di IA, i principali casi d'uso dei dati sintetici includono:

- *Ricerca e test*: i dati sintetici dell'IA generativa permettono di simulare ambienti reali complessi, come i mercati economici o le condizioni meteorologiche, per testare in sicurezza prodotti e modelli predittivi.

- *Tutela della privacy*: le organizzazioni che gestiscono dati sensibili, come quelle del settore sanitario e finanziario, possono generare dati sintetici privi di informazioni personali identificabili (PII), ma statisticamente equivalenti agli originali. Ciò garantisce un'analisi accurata dei dati, nel rispetto delle normative sulla protezione dei consumatori.
- *Riduzione dei bias*: i modelli di IA addestrati esclusivamente su dati reali tendono spesso a ereditare pregiudizi (*bias*) che producono risultati distorti. L'introduzione dei dati sintetici generati può contribuire a bilanciare il set di dati complessivo, garantendo che l'output rappresenti in modo più equo e accurato la realtà.

Generazione di Codice

Il processo di sviluppo *software* è semplificato dall'impiego dell'IA generativa, che fornisce suggerimenti di programmazione e completa automaticamente il codice sorgente in tempo reale. I programmatori inseriscono come *input* un *prompt* descrittivo delle funzioni che il codice dovrebbe eseguire e, successivamente, lo strumento di IA generativa elabora automaticamente lo *script* corrispondente. L'IA Generativa può, inoltre, modernizzare il codice *legacy* e tradurlo da un linguaggio di programmazione a un altro. Gli strumenti di generazione di codice basati sull'IA offrono supporto anche nella fase di *debugging*. Essi sono in grado di ottimizzare la leggibilità e la manutenibilità, oltre a individuare potenziali vulnerabilità di sicurezza.

1.5 Rischi, Considerazioni Etiche e Responsabilità

I modelli di IA non comprendono le domande che vengono loro poste, né le loro stesse risposte, in quanto sono solo macchine, come una calcolatrice o uno *smartphone*. Essi funzionano sulla base di sistemi statistici probabilistici addestrati a individuare gli elementi rilevanti di un contesto tramite l'architettura *Transformer*, calcolando matematicamente la continuazione testuale più verosimile.

1.5.1 Rischi Relativi ai Dati e ai Modelli

Rischi relativi ai dati

I sistemi di IA si basano su set di dati che potrebbero essere vulnerabili alla manomissione, alla distorsione dei dati o agli attacchi informatici che possono portare a violazioni dei dati. Le organizzazioni possono limitare tali rischi proteggendo l'integrità dei dati e implementando sicurezza e disponibilità in tutto il ciclo di vita dell'IA, dallo sviluppo all'addestramento, fino all'implementazione e alla post-implementazione.

Rischi Relativi al Modello

Gli autori di un attacco possono compromettere l'integrità di un modello manomettendo l'architettura, i pesi o i parametri, ovvero i componenti fondamentali che determinano il comportamento, la precisione e le prestazioni di quest'ultimo. È, quindi, fondamentale la supervisione umana sugli *output* generati. Tali risultati, pur apparendo formalmente corretti, possono, infatti, nascondere inesattezze logiche (le cosiddette allucinazioni), inefficienze architetturali o vulnerabilità di sicurezza intrinseche, mettendo a rischio la *privacy* degli utenti attraverso l'esposizione involontaria di dati sensibili. A tali criticità si aggiunge il rischio legato all'infrastruttura. Infatti l'impiego di strumenti di IA basati sul *cloud* richiede protocolli rigorosi per evitare l'esposizione accidentale di codice sorgente proprietario o informazioni aziendali riservate fornite al modello come *input*.

1.5.2 Considerazioni Etiche e Responsabilità Sociali dell'IA Generativa

In un incidente in cui è coinvolta un'auto a guida autonoma, i danni devono essere ripagati dal proprietario, dal costruttore o dal programmatore? Attualmente si fatica ancora a trovare una risposta in merito. Infatti, se il produttore fosse esonerato da ogni responsabilità, potrebbero non esserci incentivi sufficienti a fornire un prodotto sicuro ed efficiente e il pubblico potrebbe, di conseguenza, nutrire meno fiducia nella tecnologia. Allo stesso tempo, però, normative troppo rigide potrebbero soffocare i tentativi di innovazione.

Minacce dell'Intelligenza Artificiale ai Diritti Fondamentali e alla Democrazia

I risultati prodotti da un sistema di IA dipendono da come viene progettato e da quali dati vengono immessi. Tale processo può essere influenzato intenzionalmente o meno. Ad esempio, il sistema potrebbe ignorare parametri rilevanti per prendere una decisione equa, oppure basarsi su dati di addestramento che riflettano disuguaglianze strutturali già presenti nella società, i cosiddetti *bias*. Inoltre, l'uso dei dati e dei numeri per rappresentare una realtà complessa fa sembrare l'IA precisa e indipendente anche quando non lo è; tale fenomeno è conosciuto con il nome di "*math-washing*". Ad esempio, se non programmata correttamente, l'IA potrebbe condurre a decisioni relative alle offerte di lavoro, alle concessioni di prestiti e anche nei procedimenti penali, influenzate dall'etnia, dal genere, dall'età. L'IA può anche minacciare la protezione dei dati e il diritto alla vita privata. Può essere usata, ad esempio, in dispositivi per il riconoscimento facciale o per la profilazione *online*. Inoltre, è capace di integrare le informazioni che acquisisce su una persona senza che questa ne sia a conoscenza. La minaccia per la democrazia rappresentata dall'Intelligenza Artificiale passa per l'informazione. L'IA È già stata accusata di mostrare contenuti in base a ciò con cui l'utente ha interagito in passato, ostacolando la creazione di un ambiente aperto per un dibattito a più voci, inclusivo e accessibile. Essa può anche essere usata per creare immagini, video e audio falsi, ma estremamente realistici, noti come *deepfake*, che possono essere usati per truffare, rovinare la reputazione e mettere in dubbio la fiducia nei processi decisionali. L'Intelligenza Artificiale potrebbe anche minacciare la libertà di riunione e di protesta, perché potrebbe permettere di rintracciare e profilare individui legati a determinati gruppi o opinioni. Per evitare, quindi, ogni genere di pregiudizio, la normativa sull'IA richiede che i set di dati utilizzati per l'addestramento siano il più possibile completi e privi di errori. La legge sull'Intelligenza Artificiale ha anche introdotto l'obbligo di etichettare in modo chiaro e visibile le immagini, i contenuti audio o video creati o manipolati artificialmente.

I Problemi per la Trasparenza

Le disuguaglianze nell'accesso alle informazioni potrebbero essere sfruttate a discapito degli utenti. Ad esempio, sulla base del comportamento in rete di una persona o di altri dati utilizzati a sua insaputa, un fornitore di servizi può riuscire a prevedere quanto questa persona sia disposta a pagare per un servizio, oppure una campagna politica può sapere quale messaggio inviarle. Un'ulteriore criticità riguarda la trasparenza dell'interazione. Talvolta, infatti, l'utente potrebbe non riuscire a distinguere se il suo interlocutore sia un operatore umano o un sistema di Intelligenza Artificiale. Per rispondere a tali sfide, la normativa sull'IA regola l'uso della tecnologia per impedire la manipolazione del comportamento umano, l'inganno o lo sfruttamento delle vulnerabilità delle persone.

1.5.3 Sostenibilità e Impatto Ambientale

La crescente diffusione dei modelli di Intelligenza Artificiale solleva importanti interrogativi in merito al loro costo ambientale. Le fasi di addestramento e implementazione di

modelli avanzati, come ChatGPT o Google Gemini, richiedono, infatti, sia elevate quantità di energia elettrica per l'alimentazione dei *data center*, sia volumi significativi di acqua per i relativi sistemi di raffreddamento. Secondo i dati recentemente riportati dalla Banca Centrale Europea, si stima che l'elaborazione di una singola risposta da parte di un sistema di IA generativa consumi circa dieci volte più energia rispetto a una tradizionale ricerca *web*. Inoltre, in base alle stime dell'Öko-Institut, i *data center* dedicati all'Intelligenza Artificiale necessitano del doppio delle risorse idriche per funzionare rispetto alle infrastrutture informatiche tradizionali. Di pari passo con l'incremento delle capacità di calcolo e del volume delle richieste di *input* degli utenti, cresce anche il fabbisogno energetico di tali infrastrutture. Uno studio realizzato dall'Öko-Institut per Greenpeace stima che l'impiego globale di risorse idriche per il raffreddamento dei *data center* aumenterà notevolmente, passando da 175 miliardi di litri nel 2023 a 664 miliardi di litri entro il 2030. Il rapporto prevede, inoltre, nel medesimo lasso temporale, anche un aumento dei consumi di elettricità. L'apporto positivo dell'Intelligenza Artificiale alla società è, quindi, strettamente vincolato alla sua sostenibilità ambientale. Per evitare che l'elevato fabbisogno energetico aggravi la crisi climatica in atto, la diffusione di tali tecnologie deve procedere simultaneamente con lo sviluppo delle energie rinnovabili. Le principali aziende del settore devono, quindi, impegnarsi ad adottare un rifornimento energetico basato interamente su fonti pulite.

1.6 Il Futuro dell'IA Generativa

L'impiego dell'Intelligenza Artificiale potrebbe portare alla scomparsa di molti posti di lavoro. Tuttavia, emergeranno nuove opportunità professionali legate allo sviluppo e alla gestione di tali tecnologie, come la selezione dei modelli più adatti, la cura dei dati di addestramento e la validazione dei risultati generati. Risulta, quindi, fondamentale investire in adeguati percorsi di formazione e riqualificazione. Secondo le stime di Gartner, l'IA Generativa rappresenta un'innovazione tecnologica di portata rivoluzionaria, destinata a diventare «tecnologia di uso generale» per via delle sue innumerevoli potenziali applicazioni. Mentre le architetture e gli algoritmi diventano più avanzati, le organizzazioni dovranno riuscire a trovare il giusto equilibrio tra l'automazione dei processi e il coinvolgimento umano.

CAPITOLO 2

Presentazione degli LLM in Esame

Il presente capitolo analizza alcuni tra i principali modelli linguistici di grandi dimensioni (LLM). La prima sezione definisce il ruolo dei parametri all'interno delle reti neurali, chiarendo la distinzione fondamentale tra i parametri addestrabili e gli iperparametri e illustrandone le relative logiche di classificazione. Successivamente, si effettua un'analisi dettagliata di quattro sistemi: Gemini, ChatGPT, Copilot e Claude. Per ciascun modello vengono descritte l'evoluzione storica e le metodologie impiegate per la generazione di codice e di immagini, concludendo con una valutazione delle specifiche limitazioni intrinseche ad esso.

2.1 Definizione dei Parametri negli LLM

I parametri in un LLM rappresentano le variabili fondamentali che determinano il comportamento e la qualità degli *output* del sistema. Nello specifico, si distinguono due categorie principali:

- *Parametri addestrabili (pesi e bias)*: definiscono le connessioni interne della rete neurale. Questi valori non vengono impostati manualmente, ma sono appresi e progressivamente ottimizzati dal modello stesso durante l'analisi dei dati di addestramento.
- *Iperparametri*: costituiscono le configurazioni architetturali ed esterne definite dagli sviluppatori. Essi non derivano dai dati, ma servono a governare il processo di apprendimento, a definire la struttura del modello e a modulare le caratteristiche dell'*output* finale.

2.1.1 Parametri Addestrabili: Pesi e *Bias*

I pesi sono valori numerici che definiscono l'importanza assegnata dall'LLM a un determinato *input*. Durante la generazione delle risposte, infatti, il modello di IA non tratta tutte le informazioni in modo uniforme. Un peso più elevato indica una maggiore rilevanza del dato in ingresso nella generazione del risultato. I valori dei parametri addestrabili, come i pesi, vengono configurati tramite un processo di ottimizzazione durante la fase di addestramento. Tale processo valuta le prestazioni del modello di *Machine Learning* tramite una funzione di perdita, il cui obiettivo è minimizzare il margine di errore ottimizzando i parametri del sistema. All'interno delle reti neurali, i pesi fungono da moltiplicatori che determinano la potenza del segnale trasmesso da un livello neuronale all'altro. I segnali devono soddisfare

la soglia di intensità della funzione di attivazione per avanzare attraverso la rete. Di conseguenza, i pesi condizionano la propagazione dei dati attraverso i vari strati dell'architettura. Infine, il meccanismo di retropropagazione viene impiegato per calcolare in che misura la modifica di tali pesi influisca sulle prestazioni complessive del modello, permettendone il perfezionamento continuo.

Analogamente ai pesi, anche i *bias* vengono configurati in modo autonomo durante l'addestramento dei modelli di IA. Essi consistono in valori costanti che vengono sommati al segnale proveniente dai *layer* precedenti. I modelli utilizzano questi parametri per consentire ai neuroni di attivarsi in condizioni in cui i pesi da soli potrebbero non essere sufficienti per superare la soglia della funzione di attivazione. Tali valori garantiscono all'architettura una maggiore flessibilità adattiva. I *bias* vengono regolati con la retropropagazione durante la fase di apprendimento per ottimizzare le prestazioni del modello e ridurre al minimo gli errori. Negli LLM, la combinazione dei parametri addestrabili arriva a costituire architetture composte da svariati miliardi di elementi. Infine, durante la fase di messa a punto (*fine-tuning*), in cui un modello pre-addestrato viene adattato per specifiche applicazioni, tali parametri vengono ulteriormente ottimizzati sfruttando dati del dominio di riferimento.

2.1.2 Gli Iperparametri

A differenza di pesi e *bias*, gli iperparametri sono configurazioni esterne definite a priori, che determinano l'architettura globale, le dimensioni, il comportamento e l'utilizzo delle risorse computazionali di un modello. Il processo di ottimizzazione di queste impostazioni impiega algoritmi specifici per individuare la combinazione ideale al fine di incrementare le prestazioni. Unitamente al *prompt engineering*, ovvero la regolazione delle istruzioni fornite in *input* per elevare la qualità della risposta, tale pratica rappresenta uno dei metodi fondamentali per la personalizzazione e il controllo degli LLM. A seconda della loro funzione specifica, gli iperparametri possono essere suddivisi in diverse categorie.

Iperparametri dell'Architettura

Gli iperparametri dell'architettura, come il numero di *layer* e la dimensione degli strati nascosti, configurano la struttura e l'ampiezza del modello.

Iperparametri di Addestramento

Gli iperparametri di addestramento, come il tasso di apprendimento e le dimensioni del *batch* (numero di campioni elaborati in una singola iterazione del processo), guidano l'intero processo di istruzione del modello. Tali configurazioni incidono profondamente sulle prestazioni finali, determinando la capacità del sistema di superare o meno i *benchmark LLM* richiesti. Questi ultimi sono *framework* di valutazione costituiti da campioni di dati, una serie di domande o attività per testare gli *LLM* su competenze specifiche, criteri di misurazione delle prestazioni e un meccanismo di punteggio.

Iperparametri di Inferenza

Gli iperparametri di inferenza, come la temperatura e il campionamento *top-p*, determinano le modalità con cui un modello di IA generativa produce il proprio *output*. Nello specifico:

- *La temperatura* regola il livello di prevedibilità del testo generato. Valori bassi rendono i risultati più deterministici, orientando il modello a selezionare sistematicamente i *token*

successivi con la probabilità più alta. Viceversa, un innalzamento della temperatura introduce un maggiore grado di casualità, favorendo *output* più diversificati e originali.

- *Il Top-p* è una tecnica di campionamento che agisce in sinergia con la temperatura per controllare quanto il modello debba attenersi a schemi rigidi. Mantenere tale parametro su valori contenuti assicura un *output* preciso e coerente, mentre un suo innalzamento favorisce la generazione di risposte più articolate.

Iperparametri di Memoria e Calcolo

Gli iperparametri di memoria e calcolo, tra cui la *context window* (l'intervallo massimo di dati in *input* che un modello può elaborare e consultare simultaneamente durante il funzionamento), il numero massimo di *token* in *output* e le sequenze di interruzione, hanno la funzione di bilanciare le prestazioni e le funzionalità del modello con i requisiti delle risorse.

Iperparametri della Qualità dell'Output

Gli iperparametri della qualità dell'*output* supportano gli LLM nel generare testi più diversificati. Tra i principali strumenti di controllo, entrambi configurabili con valori compresi tra -2,0 e 2,0, figurano i seguenti:

- *Presence penalty* (penalità di presenza): un valore positivo di tale parametro penalizza i nuovi *token* in base alla loro precedente apparizione nel testo generato. Questo parametro incrementa la probabilità che il modello introduca nuovi argomenti.
- *Frequency penalty* (penalità di frequenza): un valore positivo di tale parametro penalizza i nuovi *token* in stretta proporzione alla loro frequenza di comparsa nel testo elaborato fino a quel momento, riducendo la probabilità che il modello ripeta letteralmente le medesime sequenze.

2.2 Classificazione degli LLM: Modelli Proprietari e *Open Source*

Il campo degli LLM offre diverse opzioni. In generale, è possibile raggruppare gli LLM in due categorie, ovvero proprietari e open source.

2.2.1 Modelli Proprietari

I modelli proprietari sono sviluppati da aziende private e il loro utilizzo richiede generalmente specifiche licenze. L'esempio più rappresentativo è GPT-4o, il modello alla base di ChatGPT. Quest'ultimo mette a disposizione un'interfaccia intuitiva tramite la quale gli utenti possono inserire *prompt* e ottenere risposte pertinenti. Inoltre, gli sviluppatori hanno la possibilità di accedere alle API dedicate per integrare tale LLM in applicazioni e servizi terzi. Ulteriori esempi in questa categoria includono Gemini di Google, Claude di Anthropic e Copilot di Microsoft.

2.2.2 Modelli *Open Source*

Un'alternativa consiste nell'implementare un LLM in modo autonomo, sfruttando tipicamente modelli *open source* rilasciati per uso commerciale. Questo settore ha compiuto rapidi progressi, riuscendo ad allineare le proprie prestazioni a quelle delle architetture proprietarie. Tra i modelli più diffusi figurano Llama 4 di Meta e Mixtral 8x22B.

2.2.3 Criteri di Valutazione e Selezione

Le considerazioni fondamentali che guidano la scelta tra l'utilizzo di un'*API* chiusa di un fornitore terzo e l'*hosting* interno del proprio LLM *open source* si concentrano sulla sostenibilità futura, sulla gestione economica e sull'impiego strategico dei dati. I modelli proprietari possono diventare obsoleti e venire rimossi; i modelli *open source*, al contrario, garantiscono una disponibilità illimitata nel tempo. Questi ultimi, se ottimizzati, offrono maggiori opzioni di personalizzazione, consentendo un migliore compromesso tra prestazioni e costi. Pianificare l'ottimizzazione futura dei propri modelli permette di sfruttare i dati dell'organizzazione come vantaggio competitivo per costruire modelli migliori di quelli disponibili pubblicamente. Infine, i modelli proprietari possono sollevare problemi di *governance*, in quanto questi LLM *black box* consentono una minore supervisione dei processi di addestramento e dei pesi. L'*hosting* dei propri LLM *open source* richiede maggiore sforzo implementativo rispetto all'utilizzo di LLM proprietari. MLflow di Databricks rende più semplice, per chi ha esperienza con Python, estrarre qualsiasi modello *Transformer* e usarlo come oggetto del linguaggio stesso.

2.2.4 Strumenti di Valutazione

La valutazione degli LLM rappresenta un processo complesso e in continua evoluzione, soprattutto perché questi modelli dimostrano spesso capacità variabili in base al *task*. Un *LLM* potrebbe eccellere in uno specifico *benchmark*, ma subire una variazione nelle prestazioni a causa di minime modifiche nel *prompt* o nel problema posto. Tra i principali strumenti e *framework* di riferimento impiegati per misurarne l'efficacia figurano:

- *MLflow*: fornisce una serie di strumenti *LLMOps* per la valutazione dei modelli. Con *Large Language Model Operations* (*LLMOps*) ci si riferisce a un insieme di metodologie operative utilizzate per supervisionare i modelli linguistici di grandi dimensioni. Questo approccio permette di gestire e automatizzare il ciclo di vita degli LLM in tutte le sue fasi, dal *fine-tuning* alla manutenzione, costituendo, quindi, un valido supporto per team aziendali e sviluppatori.
- *Mosaic Model Gauntlet*: un approccio di valutazione composto che, come mostrato nella Figura 1.3, suddivide le competenze del modello in sei vasti ambiti anziché fornire un'unica misura complessiva.

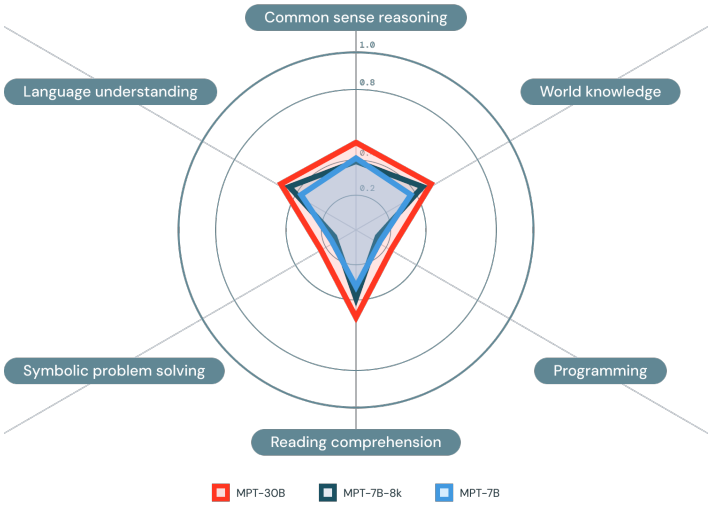


Figura 2.1: Categorizzazione delle Competenze del Modello

- *Hugging Face*: ospita centinaia di migliaia di modelli condivisi dagli sviluppatori di LLM *open source*.
- *BIG-bench (Beyond the Imitation Game Benchmark)* : un *framework* di *benchmarking* dinamico che attualmente include oltre 200 *task*, con l'obiettivo di adattarsi alle future capacità degli LLM.
- *EleutherAI LM Evaluation Harness*: un *framework* olistico, un'architettura in cui è possibile trattare un ambiente tecnologico multisfaccettato come un'unica entità che testa i modelli su oltre 200 *task*, integrando valutazioni come *BIG-bench* al fine di promuovere la riproducibilità e la comparabilità.

2.3 Discussione dei Modelli in Esame

Per l'analisi dei casi di studio presentata nei prossimi capitoli, sono stati selezionati quattro *Large Language Model*: *Gemini*, *ChatGPT*, *Copilot* e *Claude*. In particolare, si è deciso di prendere in esame due modelli utilizzati nella loro versione di base (*ChatGPT* e *Claude*) e altrettanti impiegati in versione avanzata (*Gemini* e *Copilot*, resi disponibili tramite licenze per studenti). Questa struttura permette di valutare le prestazioni non solo tra sviluppatori diversi, ma anche tra versioni gratuite e a pagamento. Di seguito si riportano le versioni impiegate durante i test:

- Gemini: Pro 3.1;
- ChatGPT: 5.6 Luna;
- Copilot: basato su GPT-4 (impostato su ThinkDeeper);
- Claude: Sonnet 5 (impostato su un livello di elaborazione medio).

Una caratteristica comune dei modelli valutati è l'utilizzo dei *token*, ovvero unità con cui il sistema elabora e produce il linguaggio. Nello specifico, essi si dividono in:

- *token* di *input*: il *prompt* fornito dall'utente;
- *token* di *output*: il codice o l'immagine generati dal modello.

2.3.1 Gemini

Gemini è l'LLM sviluppato da Google. Progettato con un'architettura nativamente multimodale, il sistema è in grado di comprendere, elaborare e combinare simultaneamente diverse modalità, quali testo, immagini, audio e video.

Evoluzione Storica

Il suo sviluppo si basa sulla ricerca all'avanguardia di Google nel campo degli LLM, iniziata con l'articolo "Efficient Estimation of Word Representations in Vector Space" su Word2Vec del 2013, il quale proponeva nuovi modelli in grado di trattare le parole come concetti matematici, a cui è seguita l'introduzione di un modello conversazionale neurale nel 2015. Tale *framework* ha dimostrato che i modelli potevano prevedere la frase successiva in una conversazione basandosi sulle affermazioni precedenti, generando interazioni più naturali. Questi progressi sono stati consolidati nel 2017 dalla pubblicazione dell'articolo sull'architettura dei *Transformer* e, nel 2020, dall'implementazione delle funzionalità di *chat*

a più turni, portando a sviluppi ancora più rilevanti nel campo del linguaggio generativo. Gemini è stato inizialmente rilasciato nel 2023 e, da allora, ha trovato diversi impieghi, quali il supporto alla stesura di testi, il *debugging* del codice, l'ideazione di eventi futuri e l'apprendimento di concetti complessi. Il modello è inoltre soggetto a regolari aggiunte di nuove funzionalità e innovazioni.

Generazione di Codice e Immagini

Il modello può apprendere in modo iterativo dai risultati dei vari *input* fino a raggiungere un *output* finale. Gemini è in grado di eseguire codice solo in Python; tuttavia può ricevere richieste per generare codice in un altro linguaggio di programmazione.

Per quanto riguarda la generazione di immagini, il modello Nano Banana 2 ne permette la creazione a partire da *prompt* testuali. Tra le sue funzionalità principali spiccano la coerenza dei soggetti, che consente di mantenere inalterato l'aspetto di un personaggio nel corso di più generazioni o di fondere scene diverse, e le modifiche mirate, utili per intervenire rapidamente su porzioni specifiche dell'immagine. Il sistema garantisce la comprensione di istruzioni complesse e la produzione di *output* ad alta risoluzione. Vengono proposti diversi modelli in base alle esigenze dell'utente:

- *Nano Banana 2 Lite*: utilizzato quando il modello è impostato su *Flash-Lite*. Essendo ottimizzato per la velocità, risulta ideale per la generazione rapida di immagini; tuttavia, non è progettato per gestire più immagini o per effettuare più modifiche.
- *Nano Banana 2*: selezionato quando si esegue una richiesta di immagine con il modello impostato su *Flash* o *Pro*. Combina velocità e qualità superiore dell'immagine. Supporta, inoltre, più immagini di riferimento.
- *Nano Banana Pro*: utilizzato quando il modello è impostato su *Pro* e si vuole rielaborare un'immagine integrandovi ulteriori dettagli. È il modello con il più alto livello di comprensione generale e offre un controllo creativo più accurato.

Limitazioni

Nonostante le elevate capacità generative, l'architettura di Gemini presenta alcune limitazioni. Tra le principali si trovano:

- *accuratezza*: gli *output* generati potrebbero presentare inesattezze, soprattutto nell'elaborazione di argomenti altamente specializzati;
- *bias*: le risposte potrebbero riflettere *bias* presenti nei dati di addestramento;
- *personalità*: le risposte del modello potrebbero erroneamente suggerire che quest'ultimo possieda opinioni o sentimenti propri;
- *falsi positivi e falsi negativi*: Gemini potrebbe rifiutarsi di elaborare alcuni *prompt* del tutto legittimi o, al contrario, generare risposte a richieste inappropriate;

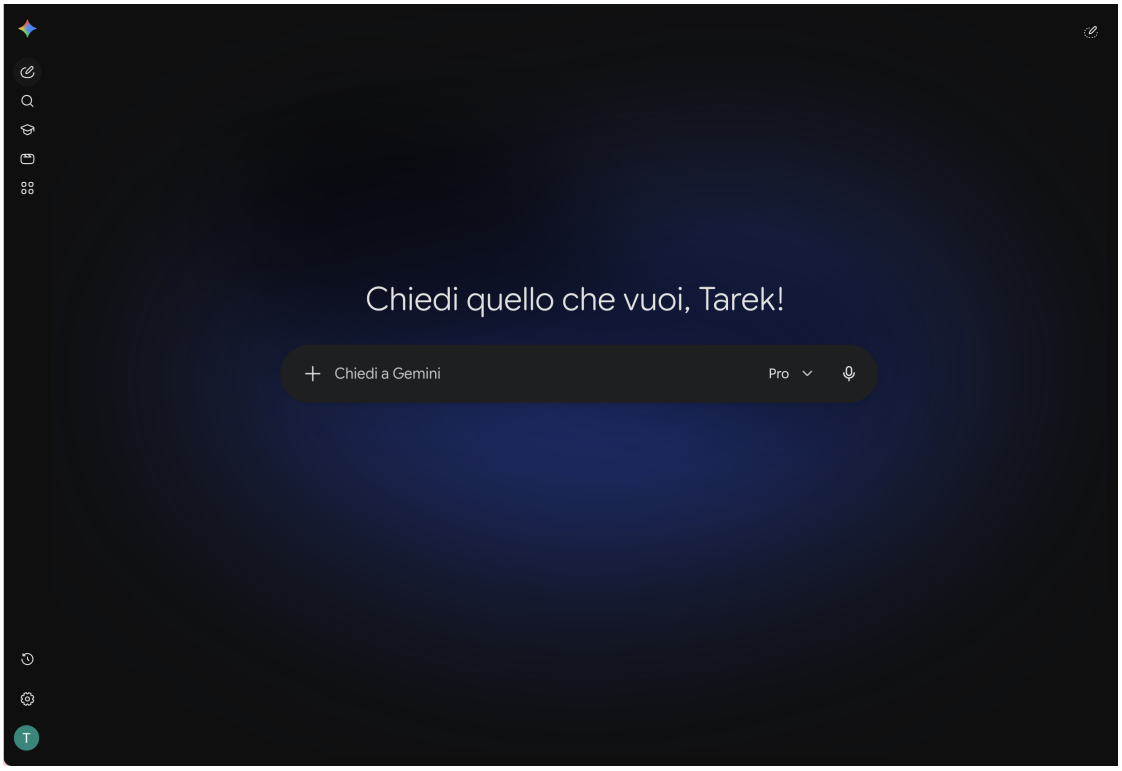


Figura 2.2: Schermata Relativa a una Nuova Chat di Gemini

2.3.2 ChatGPT

ChatGPT è un *chatbot* capace di generare testi coerenti con il linguaggio umano ed eseguire attività sulla base di *prompt* testuali. L'acronimo GPT sta per "Generative Pre-trained Transformer", un LLM che utilizza il Deep Learning per interagire in linguaggio naturale.

Evoluzione Storica

Il percorso evolutivo ha avuto inizio nel giugno 2018 con il lancio di GPT-1, il primo modello linguistico sviluppato da OpenAI a sfruttare l'architettura *Transformer*. GPT-1, con i suoi 117 milioni di parametri, figurava tra i modelli linguistici più rilevanti dell'epoca. Il sistema utilizzava libri come dati di addestramento ed era in grado di svolgere diverse attività, tra cui la comprensione del testo e il ragionamento basato sul senso comune. Nel febbraio 2019 è stato introdotto GPT-2, il quale contava 1,5 miliardi di parametri ed era stato addestrato utilizzando informazioni provenienti da Internet. Quest'ultimo era in grado di svolgere più compiti senza la necessità di un addestramento specifico per ciascuno di essi. Il rilascio di GPT-3 nel 2020 ha rappresentato un importante miglioramento in termini di dimensioni e capacità del modello, raggiungendo i 175 miliardi di parametri e prestazioni decisamente superiori rispetto ai modelli precedenti. Nel gennaio 2022 è stato lanciato InstructGPT, una versione di GPT-3 sottoposta a *fine-tuning*. L'obiettivo principale di OpenAI con questo modello era ridurre il linguaggio offensivo e la disinformazione, fornendo al contempo risposte ritenute utili dagli utenti. Questo passaggio ha posto le basi per GPT-3.5 (modello alla base di ChatGPT), una versione ottimizzata di GPT-3 in grado di comprendere e generare linguaggio naturale e codice. ChatGPT è stato reso disponibile al pubblico nel novembre 2022. Le capacità tecniche di InstructGPT e ChatGPT sono pressoché identiche, poiché entrambi i modelli sono stati addestrati utilizzando il metodo RLHF (*Reinforcement Learning from Human Feedback*). Tuttavia, l'aggiunta di dati di addestramento di tipo conversazionale e l'ottimizzazione del

processo di addestramento stesso hanno reso ChatGPT più intuitivo e capace di comprendere le preferenze degli utenti. In seguito, nel marzo 2023, è stato rilasciato GPT-4. La differenza più significativa rispetto a GPT-3.5 risiede nell'ampliamento della finestra di contesto. Questo aggiornamento ha, inoltre, segnato il primo passo verso la multimodalità, introducendo la capacità di accettare immagini come *input*, sebbene la generazione dell'*output* rimanesse limitata al solo formato testuale. Infine, è stato introdotto GPT-5 nel 2025, segnando un'altra tappa fondamentale nell'evoluzione degli LLM, con prestazioni superiori nell'elaborazione di codice, nella scrittura, nella ricerca, nella percezione visiva e nei compiti di ragionamento complesso.

Generazione di Codice e Immagini

Introdotto nel luglio 2023, Code Interpreter ha migliorato la capacità di ChatGPT di gestire file, eseguire codice e supportare attività complesse di analisi dei dati. In particolare, è in grado di elaborare *input* e generare *output* in molteplici formati come righe di codice e immagini, perfezionando la sua abilità di processare le informazioni e produrre i risultati desiderati. Lo strumento consente ai modelli di scrivere ed eseguire codice Python in un ambiente isolato (*sandbox*). Questa operazione può avvenire in modo iterativo per risolvere problemi: ad esempio, se il codice scritto da un modello non viene eseguito correttamente, può continuare a riscriverlo ed eseguirlo finché l'operazione non ha successo.

Quando si include lo strumento *image_generation* nella richiesta, il sistema può decidere quando e come generare immagini nel corso della conversazione, utilizzando il *prompt* ed eventuali file forniti in *input*. È possibile modificare le immagini, perfezionandole durante la conversazione, facendo riferimento alle risposte precedenti o ai loro ID.

Limitazioni

Nonostante le notevoli capacità generative, l'impiego operativo di ChatGPT è soggetto ad alcune criticità intrinseche, tra cui emergono le seguenti:

- *Vulnerabilità al jailbreaking*: pratica volta a eludere i vincoli di sicurezza per indurre il modello a fornire informazioni soggette a restrizioni. Per contrastare tale fenomeno, vengono impiegate tecniche di addestramento avversario (come l'utilizzo di reti GAN), i cui *output* vengono riutilizzati come dati di apprendimento.

- *Accuratezza fattuale*: la qualità e la veridicità delle risposte dipendono completamente dai dati di addestramento. L'impossibilità di esercitare un controllo totale su volumi di dati così vasti rende l'accuratezza dei fatti un problema strutturale, imponendo all'utente di non considerare infallibile la tecnologia.

- *Presenza di bias*: deriva dall'impossibilità tecnica di avere dati neutri. Il modello tenderà a riflettere costantemente i pregiudizi e le distorsioni intrinseche nei dati utilizzati per il suo addestramento. Pertanto è fondamentale non affidarsi completamente ai risultati prodotti dagli strumenti di IA, valutando con attenzione come e quando avvalersene.

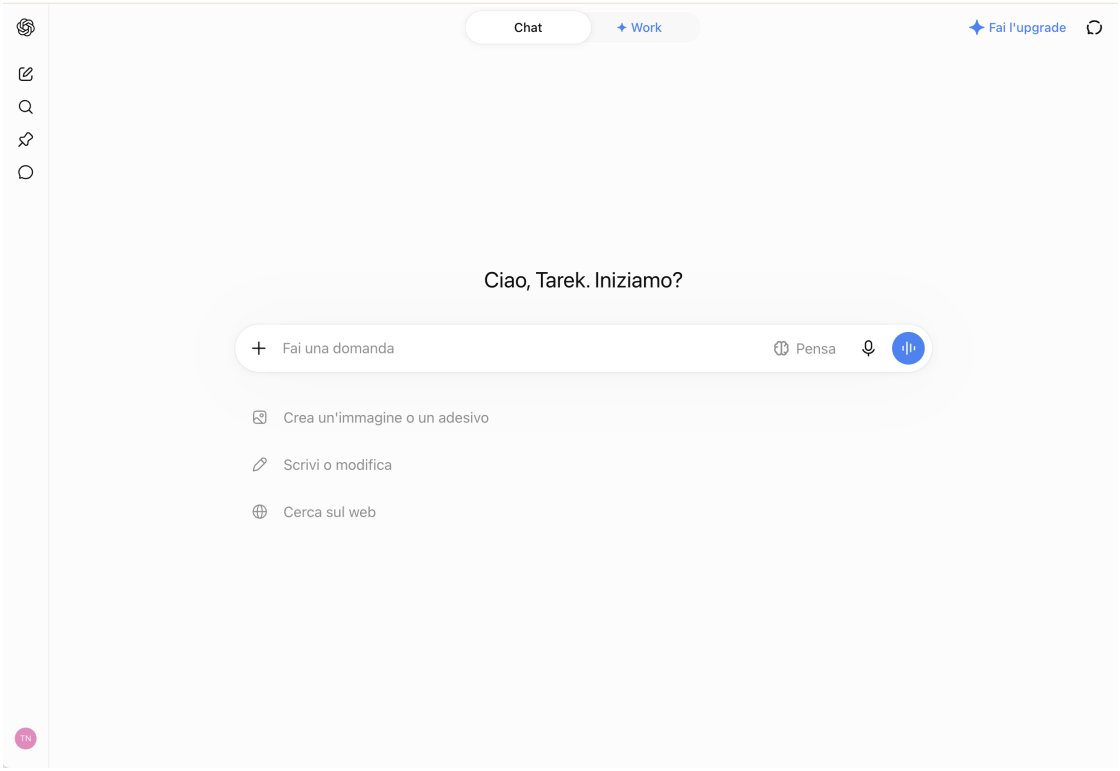


Figura 2.3: Schermata Relativa a una Nuova Chat di ChatGPT

2.3.3 Copilot

Copilot è un assistente basato sull'IA, sviluppato da Microsoft in collaborazione con OpenAI. Nato inizialmente come strumento di supporto alla programmazione, si è evoluto in una risorsa versatile, capace di potenziare la produttività in svariati ambiti.

Evoluzione Storica

Il percorso evolutivo del progetto è iniziato nel giugno 2021 con il lancio in anteprima tecnica di GitHub Copilot, uno strumento su misura per gli sviluppatori basato sui modelli di elaborazione del linguaggio naturale (NLP) di OpenAI. Integrato direttamente nella piattaforma, l'assistente consentiva di ottimizzare i tempi di programmazione tramite il completamento automatico di *snippet* (frammenti di codice) e di ridurre gli errori attraverso suggerimenti basati sul contesto. Nel 2023, l'ecosistema è stato ampliato con l'introduzione di Microsoft 365 Copilot, che ha esteso l'IA a strumenti di produttività quali Word, Excel e Teams, permettendo la generazione automatica di report e offrendo suggerimenti in tempo reale. Successivamente, per ottimizzare precisione e versatilità, Microsoft ha integrato GPT-4: aggiornamento che ha consentito al modello di comprendere *prompt* più complessi, generando *output* più coerenti.

Generazione di Codice e Immagini

All'interno di Microsoft Copilot Studio, lo strumento di interpretazione del codice e il creatore di *prompt* permettono di sviluppare agenti intelligenti capaci di eseguire script in Python per l'analisi dei dati, la generazione di visualizzazioni e l'elaborazione di documenti (Word, Excel, PowerPoint e PDF).

Microsoft Copilot ha una funzionalità nativa per creare immagini a partire da richieste testuali. L'architettura permette, inoltre, al modello di analizzare e trasformare un'immagine caricata precedentemente. Più preciso è il *prompt*, più è possibile generare immagini con stili artistici specifici o basate su descrizioni dettagliate di personaggi, stati d'animo e ambientazioni.

Limitazioni

Il modello presenta diverse limitazioni, tra cui:

- *Propagazione dei bias*: i dati di pubblico dominio, impiegati per addestrare modelli su larga scala, contengono pregiudizi sociali che si riflettono inevitabilmente nella distribuzione di parole, espressioni e strutture sintattiche.
- *Inadeguatezza per contenuti aperti*: gli scenari in cui gli utenti possono generare contenuti su qualsiasi argomento senza vincoli presentano il rischio di produrre testi offensivi o dannosi, specialmente nelle generazioni di testo più estese.
- *Limiti temporali e fattuali*: il servizio non dispone di informazioni su eventi verificatisi dopo la conclusione del suo addestramento. Risultando privo di conoscenze su fatti recenti, non è adatto a scenari che richiedono informazioni strettamente aggiornate o di assoluta correttezza fattuale.
- *Inapplicabilità in ambiti critici*: occorre escludere l'impiego del sistema in contesti che incidono direttamente sulla salute o sui diritti fondamentali, come nel caso di diagnosi cliniche o sentenze legali.

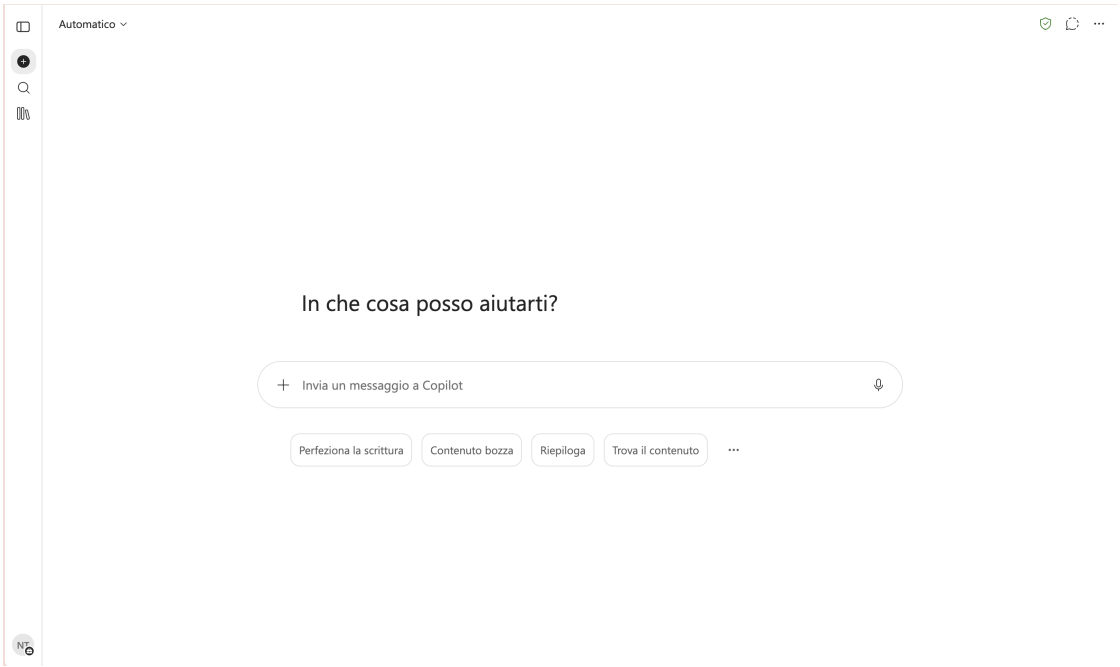


Figura 2.4: Schermata Relativa alla Nuova Chat di Copilot

2.3.4 Claude

Claude è una piattaforma di IA sviluppata da Anthropic. L'architettura si distingue per le elevate capacità in molteplici ambiti cognitivi, tra cui l'elaborazione del linguaggio naturale, il ragionamento logico, l'analisi documentale e la generazione di codice.

Evoluzione Storica

Reso disponibile al pubblico per la prima volta a marzo 2023, il modello offriva inizialmente due versioni: Claude e Claude Instant. La prima garantiva alte prestazioni, mentre la seconda rappresentava un'opzione più leggera, economica e nettamente più rapida. Gli sviluppatori, al rilascio, hanno comunicato che avrebbero continuato a lavorare per renderli più utili, onesti e innocui, basandosi sui risultati delle ricerche sulla sicurezza e sull'impiego pratico da parte degli utenti. A luglio dello stesso anno è stata sviluppata l'evoluzione del primo modello: Claude 2. Questa nuova versione presentava prestazioni migliorate e risposte più estese. Gli utenti hanno elogiato la facilità di conversare con Claude, sottolineando la sua capacità di argomentare le risposte, il minor rischio di produrre testi dannosi e la maggiore capacità di memoria. Il sistema ha mostrato miglioramenti rispetto ai modelli precedenti in ambiti quali la programmazione, la matematica e il ragionamento. A marzo 2024 è stata annunciata la famiglia Claude 3, la quale comprende tre varianti in ordine crescente di capacità: Claude 3 Haiku, Claude 3 Sonnet e Claude 3 Opus. Ciascuna di esse offre prestazioni sempre più elevate, consentendo agli utenti di scegliere il giusto equilibrio tra intelligenza, velocità e costi in base alla propria specifica applicazione.

Generazione di Codice e Immagini

Lo strumento di esecuzione del codice permette al sistema di lanciare comandi *Bash* e di manipolare *file*, inclusa la scrittura di codice, all'interno di un ambiente sicuro e isolato (*sandbox*). Tale architettura alimenta anche il filtraggio dinamico per le funzioni di *web search* e *web fetch*: il modello filtra i risultati all'interno dell'ambiente di esecuzione del codice prima che raggiungano la finestra di contesto (*context window*). Quando il filtraggio dinamico è attivo, l'API configura in automatico l'ambiente di esecuzione del codice necessario per la richiesta; quindi, non si deve aggiungere manualmente tale strumento per questo scopo.

Claude non genera foto o illustrazioni come fanno gli strumenti di generazione di immagini. Può, tuttavia, creare diagrammi, grafici e elementi visivi interattivi direttamente nella conversazione per aiutare nella comprensione. Questi vengono generati utilizzando HTML e SVG; quindi, è possibile chiedere di modificarli mentre si procede. Il modello permette anche la visualizzazione e l'analisi delle immagini che vengono caricate.

Limitazioni

L'addestramento dell'infrastruttura si fonda sul principio di rendere il modello utile, affidabile e sicuro. Tale obiettivo è perseguito fornendo al sistema una "Costituzione", ovvero un impianto di principi etici e comportamentali utilizzato per orientare gli *output* e vincolare l'architettura a non assistere l'utente in attività illecite o non etiche. Difatti, il sistema presenta le seguenti limitazioni:

- *Onestà e accuratezza fattuale*: la valutazione del modello si basa sulla capacità di fornire risposte precise e fedeli ai materiali di riferimento. In condizioni di incertezza, l'architettura è istruita a esplicitare i propri limiti, esprimendo dubbi o ammettendo la carenza di informazioni sufficienti per formulare una risposta definitiva.

- *Inapplicabilità autonoma in scenari ad alto rischio*: è escluso l'impiego indipendente del modello in contesti critici in cui un errore comporterebbe danni significativi. Sebbene possa fornire supporto in domini complessi come quello legale o medico, il sistema non può operare in sostituzione dei professionisti, rendendo obbligatoria la revisione umana dei risultati.
- *Vincoli informativi e temporali*: le risposte si basano sui dati immagazzinati durante la fase di addestramento. In assenza di moduli esterni per la ricerca *web*, il modello non è in grado di recuperare informazioni in tempo reale e vincola le proprie conoscenze alla data di conclusione del suo addestramento (collocata tra la seconda metà del 2023 e la primavera del 2024).
- *Disomogeneità multilingue*: sebbene l'architettura possieda avanzate capacità di comprensione, il modello registra un calo di efficacia e accuratezza quando opera su lingue a scarse risorse (*low-resource languages*), un limite tecnico dovuto alla ridotta mole di dati per l'addestramento.

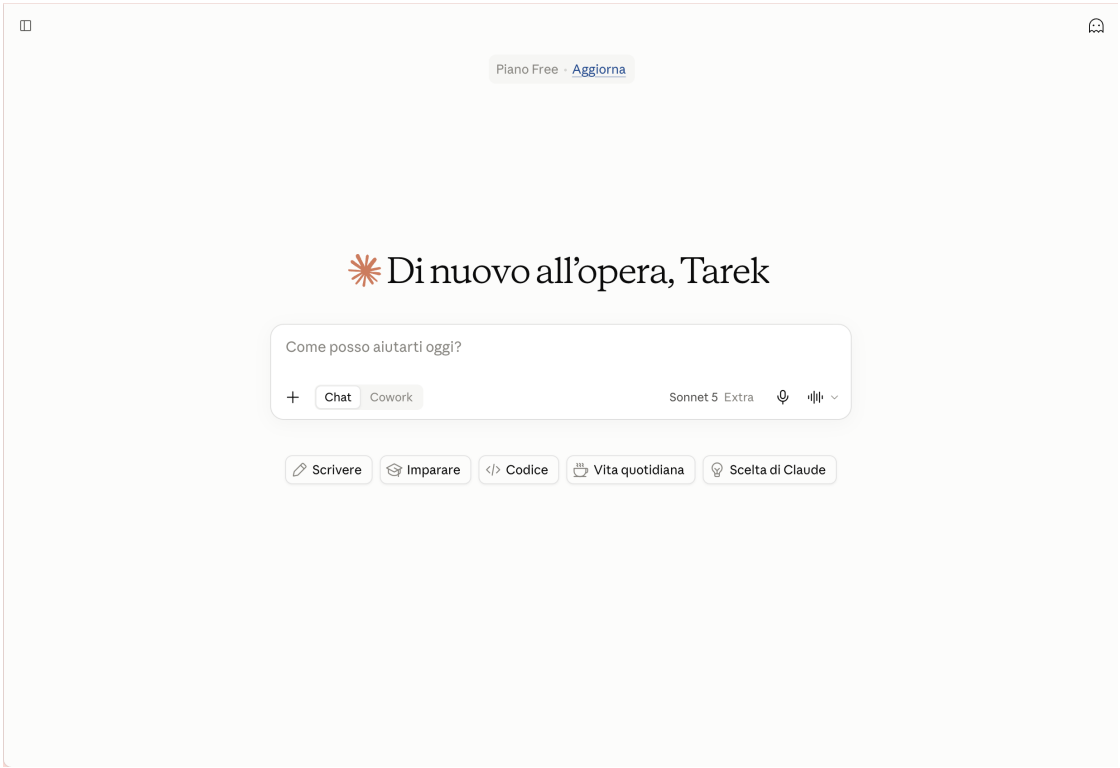


Figura 2.5: Schermata Relativa alla Nuova Chat di Claude

CAPITOLO 3

Un *Case Study* per Confrontare Diversi LLM nella Generazione di Codice

Questo capitolo presenta gli obiettivi del case study per la generazione di codice, delineando i requisiti e le dinamiche del progetto da realizzare. L'analisi prosegue esaminando nel dettaglio il processo di sviluppo condotto con i quattro modelli linguistici selezionati, riportando i principali prompt utilizzati, le risposte generate e gli snippet più rilevanti. La sezione finale propone un confronto comparativo mirato a far emergere le differenze architetturali delle soluzioni proposte.

3.1 Definizione del *Case Study*

La valutazione riguardante la capacità di generazione di codice degli LLM è stata condotta richiedendo ai vari sistemi la stesura di uno *script* in Python, finalizzato alla realizzazione di un *quiz* interattivo a tema calcistico. L'ambiente di gioco presenta un rigorista e un portiere, inquadrati attraverso una prospettiva alle spalle del tiratore. All'avvio del programma, ha inizio la rincorsa del calciatore verso la palla. Raggiunta la posizione di tiro, l'animazione si interrompe per far comparire sullo schermo il quesito, mostrando in contemporanea un *timer* di dieci secondi per la scelta della risposta. Per rappresentare l'esito del tiro, si gestiscono tre diversi scenari:

- se l'utente risponde in modo corretto, il calciatore segna e il contatore dei gol si incrementa;
- se l'utente seleziona l'opzione errata, il portiere avversario para il tiro;
- se l'utente non effettua alcuna scelta entro la scadenza del *timer*, il rigorista calcia fuori dallo specchio della porta.

L'ambiente grafico è stato ulteriormente migliorato introducendo l'abbigliamento per i due calciatori e integrando i tifosi sullo sfondo, dietro la linea di porta. Il sistema dispone, inoltre, di un calcolatore interno che, analizzando il numero di risposte corrette ed errate, restituisce sullo schermo il messaggio di vittoria o sconfitta. Per garantire un confronto equo, a tutti gli LLM in esame è stato sottoposto lo stesso *prompt* iniziale. I sistemi valutati in seguito sono stati guidati tramite *input* specifici, così da allinearne l'*output* al traguardo raggiunto durante il primo *test*.

3.2 Analisi dei Modelli

L'analisi sperimentale si articola nello studio individuale delle prestazioni restituite dai quattro sistemi di IA. Di seguito verranno documentate le fasi di sviluppo condotte con Gemini, ChatGPT, Copilot e Claude. Per ciascun modello verranno mostrati *snippet* di codice e schermate dell'interfaccia grafica generata durante le varie iterazioni.

3.2.1 Generazione con Gemini

Generazione del Codice di Base

Il processo di sviluppo è stato avviato sottoponendo al modello il *prompt* iniziale definito per i *test*, con l'obiettivo di valutare la sua capacità di tradurre la richiesta, formulata in linguaggio naturale, in un codice Python eseguibile. L'istruzione di partenza fornita al sistema è stata la seguente:

Utente

Dovrei creare un codice Python per la realizzazione di un quiz calcistico. Per la realizzazione avevo immaginato 2 calciatori (stickman), 1 rigorista ed un portiere, dove al momento dell'impatto con il pallone del rigorista compare la domanda alla quale l'utente dovrà rispondere, se la risposta sarà corretta il calciatore segnerà, altrimenti il pallone verrà parato dal portiere.

L'obiettivo della prima richiesta era quello di generare le funzionalità di base del programma, rimandando ai *prompt* successivi la cura dei dettagli. Di seguito si riporta il risultato fornito dal modello:

Gemini

Per realizzare questo progetto in modo che sia interattivo, visivamente chiaro e senza il bisogno di scaricare immagini esterne, la libreria ideale è `Pygame`. Assicurati di avere `Pygame` installato sul tuo computer. Puoi installarlo aprendo il terminale (o prompt dei comandi) e digitando: *pip install pygame*

A seguito della richiesta, il modello ha prodotto il codice Python fornendo le istruzioni di base per l'esecuzione. Di seguito si analizzano gli *snippet* principali per illustrarne il funzionamento.

Il primo frammento illustra la logica architetturale scelta dall'LLM, caratterizzata dalla memorizzazione dei *quiz* in una lista, dall'uso di una macchina a stati per la gestione delle fasi operative e dall'utilizzo diretto delle variabili `Xa` e `Xb` per il conteggio del punteggio (Listato 3.1).

```
# Variabili di gioco (Punteggi)
Xa = 0  # Punteggio Utente (Gol)
Xb = 0  # Punteggio Portiere (Parate)

# Lista delle domande (Struttura a dizionario)
domande = [
   {"testo": "Chi ha vinto i Mondiali del 2006?",
    "risposte": ["1. Italia", "2. Francia"], "corretta": "1"},
   # ... altre domande omesse per brevita ...
]

# Macchina a stati
stato_gioco = "corsa"  # Stati: "corsa", "domanda", "tiro", "risultato"
esito_tiro = None      # "gol" o "parata"
```

Listato 3.1: Gemini - variabili e struttura dati

La validazione dell'*input* avviene intercettando gli eventi da tastiera tramite il modulo di `Pygame`. Il tasto premuto viene confrontato con la soluzione memorizzata nel dizionario della domanda attiva; in caso di corrispondenza, il sistema incrementa la variabile `Xa` (Gol), altrimenti aggiorna `Xb` (Parate), forzando, in entrambi i casi, la transizione allo stato successivo per avviare l'animazione, come mostrato nel seguente *snippet* (Listato 3.2)

```
if stato_gioco == "domanda":
    # Controlla la risposta (Tasti 1 o 2)
    domanda_corrente = domande[indice_domanda]
    risposta_data = None

    if event.key == pygame.K_1:
        risposta_data = "1"
    elif event.key == pygame.K_2:
        risposta_data = "2"

    if risposta_data:
        if risposta_data == domanda_corrente["corretta"]:
            esito_tiro = "gol"
            Xa += 1
        else:
            esito_tiro = "parata"
            Xb += 1
        stato_gioco = "tiro"
```

Listato 3.2: Gemini - Logica di *input* e validazione

L'esito del tiro viene rappresentato graficamente tramite la traslazione lineare delle coordinate degli oggetti. Se la risposta risulta corretta, il pallone avanza verso la rete lungo l'asse X mentre il portiere devia sull'asse Y; in caso di errore, le coordinate del portiere e del pallone convergono per simulare l'intercettazione fisica, prima di passare alla schermata di risultato, come evidenziato di seguito (Listato 3.3).

```
elif stato_gioco == "tiro":
    # Animazione del pallone
    if esito_tiro == "gol":
        x_pallone += 5
        # Il portiere si tuffa dalla parte sbagliata
        y_portiere += 2
        if x_pallone >= 710:
            stato_gioco = "risultato"
    else: # parata
        x_pallone += 4
        # Il portiere va incontro al pallone
        x_portiere -= 2
        if x_pallone >= x_portiere - 20:
            stato_gioco = "risultato"
```

Listato 3.3: Gemini - Animazione del tiro ed esito

Gemini



Gestione dei Quesiti

Dall'analisi del primo *output* si è evidenziato un margine di miglioramento relativo al contenuto e alla varietà dei quesiti, i quali risultavano troppo generici e soggetti a ripetizioni cicliche. Si è, pertanto, proceduto a perfezionare il sistema tramite un *prompt* specifico, finalizzato a restringere la sfera di interesse ai soli Mondiali di calcio e a implementare un meccanismo per garantire che ogni domanda venga posta al massimo una volta. Di seguito viene riportata la richiesta di perfezionamento:

Utente

Fai in modo che le domande riguardino solo i Mondiali (sia di quest'anno che passati), e che non si ripetano tra loro.

Attraverso questa direttiva si è richiesto all'LLM di specializzare i contenuti della lista e di implementare una logica in grado di esaurire le domande evitando ripetizioni o *loop* infiniti. Il modello ha generato la seguente risposta:

Gemini

Ecco il codice aggiornato. Ho inserito una serie di domande dedicate esclusivamente alla storia dei Mondiali e all'edizione di quest'anno (2026). Per assicurarmi che le domande non si ripetano, ho importato il modulo random per mescolare la lista all'avvio. Una volta terminate tutte le domande, il gioco passerà a una nuova schermata di "fine gioco" dove verranno mostrati i risultati finali accumulati nelle variabili Xa e Xb.

Il modello ha, quindi, apportato due modifiche precise. La prima riguarda la lista dei quesiti, aggiornata esclusivamente al tema dei Mondiali di calcio. La seconda, per soddisfare il vincolo di non ripetitività, interviene sulla logica di scorrimento: l'indice dell'*array*, anziché resettarsi a zero una volta raggiunta la sua lunghezza massima, forza il passaggio a un nuovo stato terminale, impedendo, di fatto, il *loop* delle domande. Si riporta il frammento di codice relativo (Listato 3.4).

```
        elif stato_gioco == "risultato":
            indice_domanda += 1
            if indice_domanda >= len(domande):
                stato_gioco = "fine" # Tutte le domande sono terminate
            else:
                # Resetta le posizioni per il prossimo tiro
```

```
                x_rigorista = 150
                x_pallone = 250
                x_portiere = 650
                y_portiere = 350
                stato_gioco = "corsa"
```

Listato 3.4: Gemini - Gestione del termine del *quiz*

Modifiche Grafiche

La successiva iterazione si è concentrata sull'ottimizzazione dell'interfaccia. Al modello è stato fornito un *prompt* multiplo per migliorare l'esperienza d'uso:

Utente

Vorrei apportare delle modifiche grafiche:

1. rimuovi Xa e Xb in alto a destra e sinistra vicino gol e parate;
2. metti il campo non di vista laterale, ma frontale (come se vedessi il rigore da dietro il tiratore);
3. metti rete e linee di campo.

Tramite questa istruzione multipla si è richiesto di apportare diverse modifiche all'architettura visiva, testando così la flessibilità del modello nella transizione da un'inquadratura laterale a una prospettiva frontale. Di seguito la risposta dell'LLM:

Gemini

Ho riscritto la logica grafica e geometrica per creare una prospettiva frontale (3D simulato), posizionando la telecamera dietro al rigorista. Ho aggiunto le linee del campo (area di rigore, area piccola, lunetta, dischetto e linea di fondo), la rete della porta e la fascia in alto in stile "banner" per il testo, rimuovendo le scritte "(Xa)" e "(Xb)" dallo schermo.

L'implementazione della prospettiva frontale e dei dettagli ambientali è gestita da una nuova funzione di *rendering statico*. Il codice traccia le linee dell'area di rigore e genera la porta iterando il disegno di linee orizzontali e verticali per simulare le maglie della rete (Listato 3.5).

```
def disegna_campo():
    # Spalti e Prato
    pygame.draw.rect(SCHERMO, GRIGIO, (0, 80, 800, 120))
    pygame.draw.rect(SCHERMO, VERDE_CAMPO, (0, 200, 800, 400))

    # Area di rigore e Area di porta
    pygame.draw.lines(SCHERMO, BIANCO, False, [(150, 200), (150, 480), (650, 480), (650,
     200)], 3)
    pygame.draw.lines(SCHERMO, BIANCO, False, [(300, 200), (300, 280), (500, 280), (500,
     200)], 3)

    # === PORTA E RETE ===
    x_sx, x_dx, y_alto, y_basso = 260, 540, 80, 200

    # Linee della rete (Dietro i pali)
    for i in range(x_sx, x_dx + 1, 15): # Verticali
        pygame.draw.line(SCHERMO, BIANCO, (i, y_alto), (i, y_basso), 1)
    for i in range(y_alto, y_basso + 1, 15): # Orizzontali
        pygame.draw.line(SCHERMO, BIANCO, (x_sx, i), (x_dx, i), 1)
```

Listato 3.5: Gemini - Disegno del campo e della rete

Il passaggio alla visuale in terza persona (alle spalle del tiratore) richiede una logica di traslazione sull'asse Y e l'introduzione di modificatori di scala per la profondità. Il raggio del

pallone diminuisce matematicamente man mano che la sua coordinata Y si riduce (simulandone l'allontanamento), mentre agli *stickman* viene assegnato un parametro di scala fisso in base alla loro posizione sullo schermo (Listato 3.6).

```
        # Disegna Portiere (scala piu piccola perche e lontano)
        braccia_su_portiere = (stato_gioco in ["domanda", "corsa"] or (stato_gioco == "
    tiro" and esito_tiro == "parata"))
        inclinazione_portiere = -20 if (stato_gioco == "tiro" and esito_tiro == "gol")
    else 0
        disegna_stickman(SCHERMO, BLU, x_portiere, y_portiere, scala=0.8, braccia_su=
    braccia_su_portiere, inclinazione=inclinazione_portiere)

        # Disegna Pallone (cambia dimensione per prospettiva)
        raggio_pallone = max(4, int(10 * (y_pallone / 380)))
        pygame.draw.circle(SCHERMO, BIANCO, (int(x_pallone), int(y_pallone)),
    raggio_pallone)

        # Disegna Rigorista (scala piu grande perche e vicino)
        disegna_stickman(SCHERMO, NERO, x_rigorista, y_rigorista, scala=1.5)
```

Listato 3.6: Gemini - Gestione della prospettiva e della profondità

L'interfaccia utente è stata incapsulata in una funzione dedicata, eliminando le sovrapposizioni visive sullo schermo. Gli elementi testuali sono stati aggiornati raggruppando i contatori e i testi delle domande all'interno di un *banner* superiore predefinito (Listato 3.7).

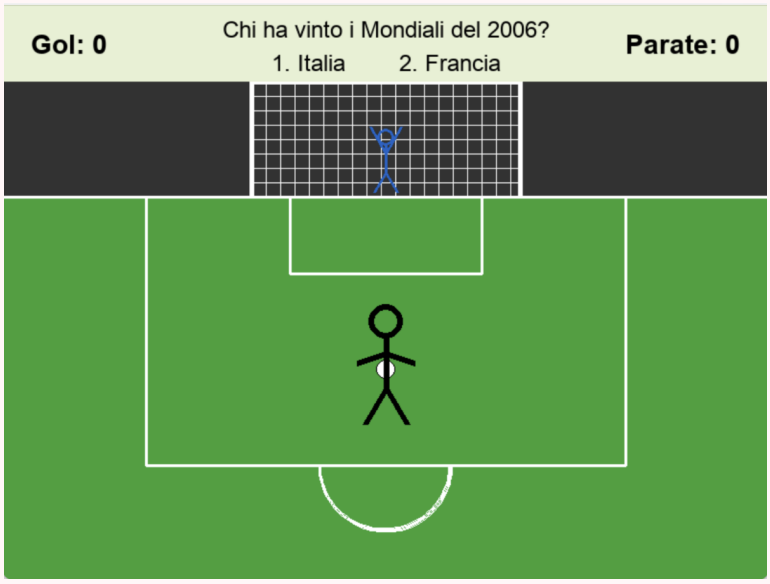
```
def disegna_ui():
    """Disegna la barra in alto stile immagine con i punteggi e le domande."""
    pygame.draw.rect(SCHERMO, BEIGE_UI, (0, 0, 800, 80))

    # Punteggi (senza l'esposizione diretta delle variabili Xa e Xb)
    img_gol = font_ui.render(f"Gol: {Xa}", True, NERO)
    img_parate = font_ui.render(f"Parate: {Xb}", True, NERO)
    SCHERMO.blit(img_gol, (30, 25))
    SCHERMO.blit(img_parate, (650, 25))

    # Se c'e una domanda, mostrala nel banner
    if stato_gioco == "domanda":
        domanda = domande[indice_domanda]
        disegna_testo_centrato(domanda["testo"], font_domanda, NERO, 400, 25)
```

Listato 3.7: Gemini - *Refactoring* dell'interfaccia utente

Gemini



Aggiunta dei Tifosi e Correzioni Grafiche

Con questo *prompt* si è richiesto al modello di arricchire lo sfondo, aggiungendo la presenza di tifosi dietro la porta, e di effettuare alcune correzioni grafiche relative alle dimensioni e al comportamento del pallone:

Utente

Vorrei effettuare ulteriori modifiche:

1. aggiunta di tifosi dietro la porta;
2. leggero incremento delle dimensioni della palla;
3. quando il calciatore segna fai in modo che la palla si fermi quando incontra la rete.

Come nell'iterazione precedente, il messaggio combina una richiesta per introdurre un nuovo elemento ambientale visivo con due interventi mirati a perfezionare la fisica e le proporzioni degli oggetti già presenti. Di seguito la risposta del modello:

Gemini

Ho generato i tifosi sugli spalti in modo statico all'avvio (per evitare che sfarfallino), ho aumentato le dimensioni del pallone modificandone la formula del raggio dinamico e ho regolato la logica del tiro affinché la palla si arresti contro il retro della rete (assegnando un limite preciso sull'asse Y) quando la variabile Xa del gol viene incrementata.

La generazione dei tifosi è gestita pre-calcolando una lista di coordinate e colori prima del *loop* principale. Per simulare la disposizione irregolare della folla sugli spalti, il modello opta per una griglia con un fattore di casualità per le posizioni e una probabilità di *spawn* del 70%, disegnandola poi con semplici elementi grafici (Listato 3.8).

```
# Generazione dei tifosi (pallini colorati sugli spalti)
tifosi = []
colori_tifosi = [ROSSO, BLU, BIANCO, (255, 215, 0), (255, 140, 0), (135, 206, 250)]
for row in range(85, 190, 12):
    for col in range(10, 790, 15):
        if random.random() > 0.3: # 70% di probabilita che ci sia un tifoso
            tifosi.append((col + random.randint(-3, 3), row + random.randint(-3, 3),
    random.choice(colori_tifosi)))

def disegna_campo():
    # Spalti
    pygame.draw.rect(SCHERMO, GRIGIO, (0, 80, 800, 120))
    # Disegna i tifosi
    for t in tifosi:
        pygame.draw.circle(SCHERMO, t[2], (t[0], t[1]), 3)
```

Listato 3.8: Gemini - Generazione e *rendering* dei tifosi

L'incremento dimensionale del pallone è stato implementato ritoccando i parametri della formula che ne calcola il raggio. La proporzione legata alla profondità (asse Y) è mantenuta, ma il valore minimo e il moltiplicatore base sono stati aumentati (Listato 3.9).

```
        # Palla ingrandita leggermente
        raggio_pallone = max(8, int(18 * (y_pallone / 380)))
        pygame.draw.circle(SCHERMO, BIANCO, (int(x_pallone), int(y_pallone)),
    raggio_pallone)
        pygame.draw.circle(SCHERMO, NERO, (int(x_pallone), int(y_pallone)),
    raggio_pallone, 1)
```

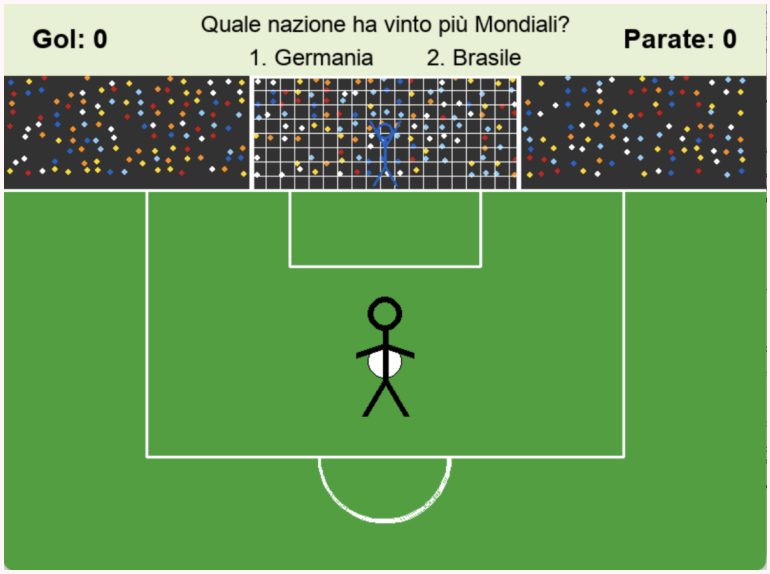
Listato 3.9: Gemini - Aggiornamento delle dimensioni del pallone

La collisione con la rete per fermare il pallone è stata risolta introducendo un blocco condizionale sull'animazione del gol. Invece di far traslare la sfera all'infinito fino all'uscita dallo schermo, il codice aggiorna le coordinate solo finché il pallone non supera la coordinata spaziale Y corrispondente al fondo della porta, forzando poi il passaggio di stato (Listato 3.10).

```
    elif stato_gioco == "tiro":
        if esito_tiro == "gol":
            if y_pallone > 110: # La palla prosegue finche non tocca la rete (y=110)
                x_pallone += 2.5
                y_pallone -= 4
                x_portiere -= 2
                y_portiere += 1
            else:
                # La palla si e incastrata nella rete, il gioco passa al risultato
                stato_gioco = "risultato"
```

Listato 3.10: Gemini - Interruzione dell'animazione sulla rete

Gemini



Divise per i Calciatori e Correzione dell'Animazione del Pallone

La successiva istruzione riguarda l'aggiunta grafica delle divise per i due calciatori. Inoltre, poiché nell'iterazione precedente l'LLM non era riuscito a implementare correttamente tutte le direttive, si è richiesto nuovamente di gestire l'arresto della sfera al momento del contatto con la rete:

Utente

Rendi il portiere dello stesso colore del tiratore e, successivamente, differenziali per il colore della maglietta (verde il portiere e bianca l'attaccante), e completa la divisa del portiere aggiungendogli dei guanti. Inoltre è ancora presente il problema della palla che non si ferma quando incontra la rete successivamente alla risposta corretta.

Il modello ha riconosciuto l'errore logico alla base del difetto visivo e ha restituito il seguente *output*:

Gemini

Hai perfettamente ragione, c'era un difetto nel calcolo della traiettoria! La velocità orizzontale della palla era troppo alta: questo la faceva viaggiare diagonalmente fino a farla "uscire" visivamente fuori dal palo destro (oltre la coordinata della rete) prima di fermarsi, dando l'impressione che non si fermasse affatto e volasse via nel vuoto. Ho ricalcolato la fisica del tiro in modo che il pallone finisca la sua corsa esattamente sotto l'incrocio dei pali, insaccandosi dentro la rete (fermandosi visivamente lì al momento del gol). Ho anche rielaborato completamente la funzione che disegna i personaggi per aggiungere spessore e dettagli: ora la base del corpo è nera per entrambi, ma possono indossare una maglietta colorata (aggiunta sopra le spalle) e, nel caso del portiere, dei guanti visibili. Ho anche ingrandito leggermente la scala del portiere per renderlo più presente in porta.

L'aggiornamento estetico dei personaggi richiede una riprogettazione della funzione di disegno. Vengono introdotti parametri opzionali per la renderizzazione stratificata degli indumenti (maglia e guanti), sovrascrivendo segmenti specifici del corpo e calcolando dinamicamente spessori e posizioni in base alla scala e alla postura dello *stickman* (Listato 3.11).

```
def disegna_stickman(superficie, colore_base, x, y, scala=1.0, braccia_su=False, inclinazione=0, colore_maglia=None, colore_guanti=None):
    # ... (calcolo coordinate base del corpo omesse per brevita) ...

    # Disegno Maglietta
    if colore_maglia:
        spessore_maglia = max(6, int(14 * scala))
        # Tronco della maglietta
        pygame.draw.line(superficie, colore_maglia, (int(x_testa), y_collo), (int(x), int(y - 8 * scala)), spessore_maglia)
        # Maniche (arrivano a meta braccio)
        meta_braccio_sx = (int((x_testa + fine_braccio_sx[0])/2), int((y_spalle + fine_braccio_sx[1])/2))
        meta_braccio_dx = (int((x_testa + fine_braccio_dx[0])/2), int((y_spalle + fine_braccio_dx[1])/2))
        pygame.draw.line(superficie, colore_maglia, (int(x_testa), y_spalle), meta_braccio_sx, spessore_maglia - 2)
        pygame.draw.line(superficie, colore_maglia, (int(x_testa), y_spalle), meta_braccio_dx, spessore_maglia - 2)

    # Disegno Guanti
    if colore_guanti:
        raggio_guanto = max(4, int(7 * scala))
        pygame.draw.circle(superficie, colore_guanti, fine_braccio_sx, raggio_guanto)
        pygame.draw.circle(superficie, colore_guanti, fine_braccio_dx, raggio_guanto)
```

Listato 3.11: Gemini - Implementazione degli indumenti e dei guanti

L'istanza degli *sprite* nel *loop* principale viene uniformata assegnando a entrambi i modelli il nero come colore base per la pelle. La differenziazione avviene passando alla funzione costanti cromatiche dedicate per la divisa, risolvendo la richiesta visiva (Listato 3.12).

```
        # Disegna Portiere (scala aumentata a 1.1 per migliore visibilita, con maglia verde e guanti gialli)
        disegna_stickman(SCHERMO, NERO, x_portiere, y_portiere, scala=1.1, braccia_su=braccia_su_portiere, inclinazione=inclinazione_portiere, colore_maglia=VERDE_MAGLIA,
         colore_guanti=GIALLO_GUANTI)

        # Disegna Rigorista (scala 1.5, con maglietta bianca, senza guanti)
        disegna_stickman(SCHERMO, NERO, x_rigorista, y_rigorista, scala=1.5, colore_maglia=BIANCO)
```

Listato 3.12: Gemini - Differenziazione cromatica dei giocatori

Per correggere definitivamente il *bug* della collisione con la rete, il modello riduce il moltiplicatore di traslazione sull'asse X. In questo modo il pallone non oltrepassa la coordinata

del palo destro prima di raggiungere il limite di stallo sull'asse Y, rimanendo visivamente confinato all'interno dell'area della porta (Listato 3.13).

```
    elif stato_gioco == "tiro":
        if esito_tiro == "gol":
            # La palla viaggia fino a Y=110 (Il fondo della rete)
            if y_pallone > 110:
                # Coordinata X ridotta a 1.5. Cosi rimane dentro al palo destro (che sta
 a x=540)
                x_pallone += 1.5
                y_pallone -= 4
                x_portiere -= 2.5
                y_portiere += 1
            else:
                # La palla tocca la rete e l'animazione termina, "incastrandola" li.
                stato_gioco = "risultato"
```

Listato 3.13: Gemini - Ricalibrazione della traiettoria del pallone

Utente

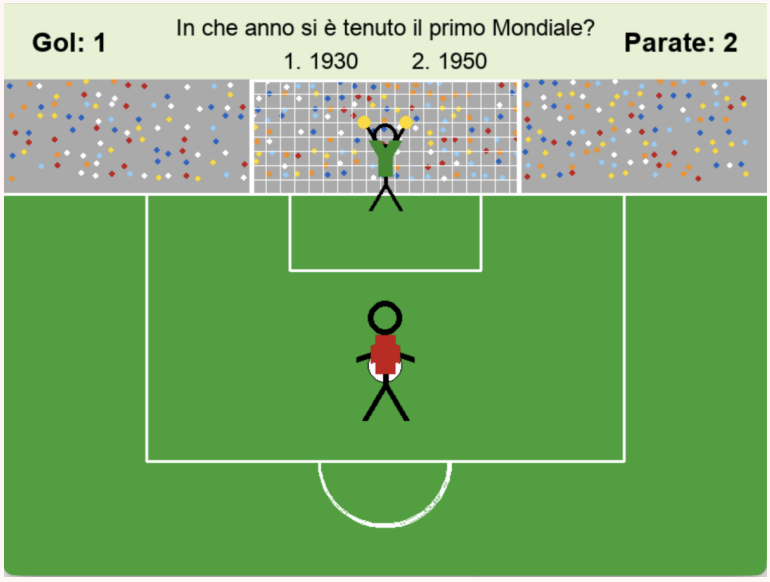
Per renderlo più visibile metterei la maglia del tiratore di rosso, e renderei il colore della gradinata di un grigio più chiaro, così da far risaltare maggiormente il portiere.

Sebbene il modello avesse rispettato le direttive precedenti, la resa cromatica dell'*output* non garantiva un contrasto visivo ottimale. Con questa successiva istruzione si è quindi richiesto di applicare due correzioni mirate: assegnare il rosso alla maglia del tiratore e schiarire le gradinate sullo sfondo, al fine di far risaltare maggiormente la figura del portiere:

Gemini

Ho apportato le modifiche per migliorare il contrasto visivo e l'impatto grafico della scena:

1. Maglia del tiratore: Ho assegnato il colore rosso (colore_maglia=ROSSO) allo stickman che batte il rigore, mantenendo la maglia verde e i guanti gialli per il portiere.
2. Spalti e contrasto: Ho modificato il colore della base degli spalti, passando da un grigio scuro a un grigio molto più chiaro (GRIGIO = (170, 170, 170)). In questo modo, lo sfondo dietro la porta risulta molto più luminoso, facendo risaltare in maniera evidente la sagoma scura del portiere e la sua maglia verde.



Aggiunta *Timer* per Risposte

Come ultima specifica per rendere l'esperienza d'uso dell'applicazione il più coinvolgente possibile, è stata richiesta l'aggiunta di un *timer* di dieci secondi entro il quale l'utente deve fornire la risposta:

Utente

Vorrei aggiungere anche un timer di 10 secondi entro cui dare la risposta. Se l'utente non inserisce alcuna risposta, allora il pallone tirato dal calciatore verrà mandato fuori dallo specchio della porta.

Il modello ha implementato la nuova logica richiesta, gestendo il caso limite della mancata selezione di una risposta:

Gemini

Ecco l'aggiornamento del codice con l'implementazione del timer di 10 secondi e della meccanica del tiro "fuori". Ho introdotto le seguenti novità:

- *Timer a schermo*: Quando il giocatore raggiunge il pallone, parte un conto alla rovescia di 10 secondi visibile in rosso sotto la barra delle domande.
- *Gestione del tempo scaduto*: Se il tempo scade senza che tu prema 1 o 2, il gioco registra automaticamente l'errore (incrementando la variabile Xb) e imposta l'esito del tiro su "fuori".
- *Animazione del tiro fuori*: Invece di fermarsi tra le braccia del portiere, il pallone schizzerà in alto, superando abbondantemente la traversa, mentre il portiere rimarrà fermo a guardare.

L'introduzione del *countdown* richiede l'impiego del modulo *time* di `Pygame` per calcolare i millisecondi trascorsi. Il *timer* viene inizializzato al termine della corsa, e il tempo trascorso viene sottratto iterativamente da una variabile di 10 secondi durante lo stato di attesa della risposta (Listato 3.14).

```
    # Logica degli Stati
    if stato_gioco == "corsa":
        y_rigorista -= 3
        if y_rigorista <= 400:
            stato_gioco = "domanda"
            tempo_inizio_domanda = pygame.time.get_ticks() # Avvia il timer

    elif stato_gioco == "domanda":
        # Calcolo timer
        tempo_trascorso = (pygame.time.get_ticks() - tempo_inizio_domanda) // 1000
        tempo_rimanente = max(0, 10 - tempo_trascorso)

        # Scade il tempo
        if tempo_rimanente == 0:
            esito_tiro = "fuori"
            Xb += 1 # Penalita
            stato_gioco = "tiro"
```

Listato 3.14: Gemini - Gestione logica e temporale del *timer*

La visualizzazione sullo schermo del tempo rimanente è stata inserita nella funzione di disegno dell'UI, posizionata sotto il testo del *quiz*. È stata aggiunta anche una soglia di allerta visiva: il testo diventa rosso negli ultimi 3 secondi per aumentare la pressione sul giocatore (Listato 3.15).

```
def disegna_ui():
    # ... (disegno banner omesse per brevita) ...

```

```
    if stato_gioco == "domanda":
        # ... (disegno testi domande omesse per brevita) ...

        # Disegna il timer
        colore_timer = ROSSO if tempo_rimanente <= 3 else NERO
        testo_timer = font_timer.render(f"Tempo: {tempo_rimanente}s", True, colore_timer
    )
        SCHERMO.blit(testo_timer, (330, 85))
```

Listato 3.15: Gemini - Visualizzazione sullo schermo del *countdown*

La scadenza del *timer* innesca una nuova tipologia di esito, `fuori`, che va ad aggiungersi a `gol` e `parata`. Questa condizione attiva un'animazione dedicata in cui il pallone esce dallo specchio della porta, oltrepassando il limite superiore dello schermo prima di passare alla schermata di risultato (Listato 3.16).

```
    elif stato_gioco == "tiro":
        # ... (condizioni gol e parata omesse per brevita) ...

        elif esito_tiro == "fuori":
            # La palla vola altissima sopra la traversa
            if y_pallone > -50:
                y_pallone -= 7 # Velocita maggiore verso l'alto
                x_pallone -= 1 # Leggera deviazione a sinistra
            else:
                stato_gioco = "risultato"
```

Listato 3.16: Gemini - Risoluzione visiva del tiro fuori bersaglio

Gemini



L'implementazione del *timer* ha, tuttavia, generato un'imprecisione visiva in caso di mancata risposta: la palla manteneva la sua traiettoria centrale, finendo per oltrepassare la porta senza bloccarsi. Per ovviare a questo *bug* visivo, si è richiesto al modello di correggere tale dinamica. La soluzione adottata dall'LLM si basa sull'applicazione alla sfera di una costante traslazione laterale verso destra, combinata all'avanzamento in profondità, per simulare in modo realistico un tiro fuori bersaglio.

3.2.2 Generazione con ChatGPT

Progettazione e Implementazione dell'Architettura

Il test sul secondo modello è iniziato riproponendo lo stesso *prompt* iniziale, così da fornire al nuovo sistema la stessa base di partenza del precedente:

Utente

Dovrei creare un codice Python per la realizzazione di un quiz calcistico. Per la realizzazione avevo immaginato 2 calciatori (stickman), 1 rigorista ed un portiere, dove al momento dell'impatto con il pallone del rigorista compare la domanda alla quale l'utente dovrà rispondere, se la risposta sarà corretta il calciatore segnerà, altrimenti il pallone verrà parato dal portiere.

L'approccio iniziale adottato da questo secondo modello si è rivelato nettamente differente dal precedente. Anziché tradurre immediatamente la richiesta in codice, il sistema ha presentato prima un piano per definire la struttura architetturale:

ChatGPT

È un'ottima idea perché trasforma un semplice quiz in un piccolo gioco interattivo. Ti suggerisco di suddividerlo in moduli, così il codice sarà più semplice da sviluppare e da modificare.
Architettura consigliata: main.py

- `menu.py`;
- `quiz.py`;
- `animazioni.py`;
- `giocatori.py`;
- `pallone.py`;
- `domande.py`;
- `punteggio.py`;
- `assets/`

La risposta evidenzia come il modello abbia deciso di scomporre il problema principale in *script* più piccoli e specializzati. L'architettura suggerita separa il motore di gioco principale (`main.py`) dalle risorse e dalle logiche secondarie (come la gestione dei quiz e delle animazioni). Tale scelta si è rivelata, però, a livello pratico, piuttosto macchinosa. La problematica principale è emersa nella gestione delle dipendenze: ogni volta che il modello generava o modificava uno *script* secondario, si rendeva necessario un ulteriore *prompt* correttivo per aggiornare e riallineare il *file* `main.py` alle nuove logiche. Si riportano successivamente i vari *snippet* dei codici secondari, per poi presentare in conclusione la versione definitiva del `main.py`.

quiz.py

Il primo *script* analizzato si occupa della gestione delle domande durante l'esecuzione. L'inizializzazione della classe configura i *font*, struttura i dizionari (passando a quattro opzioni di risposta) e prepara una lista vuota per memorizzare dinamicamente le aree di collisione dei pulsanti sullo schermo (LIstato 3.17).

```
1 class Quiz:
2     def __init__(self):
```

```
        self.font_domanda = pygame.font.SysFont("Arial", 32)
        self.font_risposte = pygame.font.SysFont("Arial", 26)

        # Lista di dizionari con 4 opzioni di risposta
        self.domande = [
            {
                "domanda": "Chi ha vinto il Mondiale 2006?",
                "risposte": ["Italia", "Francia", "Brasile", "Germania"],
                "corretta": 0
            },
            # ... altre domande omesse per brevita ...
        ]

        random.shuffle(self.domande)
        self.indice = 0
        self.rettangoli = []
```

Listato 3.17: ChatGPT - Struttura dati ad oggetti per il *quiz*

La funzione di disegno non stampa un semplice testo, ma genera iterativamente un'interfaccia utente. Per ogni risposta possibile, il sistema calcola e disegna un rettangolo, salvandone le coordinate spaziali per la successiva validazione dell'*input* (Listato 3.18).

```
    def disegna(self, screen):
        screen.fill((40,120,40))
        domanda = self.domanda_corrente()

        if domanda is None:
            return

        # Rendering del testo della domanda
        testo = self.font_domanda.render(domanda["domanda"], True, (255,255,255))
        screen.blit(testo, (70,60))

        self.rettangoli = []
        y = 170

        # Creazione iterativa dei pulsanti cliccabili
        for risposta in domanda["risposte"]:
            rect = pygame.Rect(120, y, 760, 60)
            pygame.draw.rect(screen, (240,240,240), rect)
            pygame.draw.rect(screen, (0,0,0), rect, 2)

            txt = self.font_risposte.render(risposta, True, (0,0,0))
            screen.blit(txt, (145, y+15))

            self.rettangoli.append(rect)
            y += 90
```

Listato 3.18: ChatGPT - *Rendering* dinamico dell'interfaccia a pulsanti

La validazione delle risposte non si basa più sulla pressione dei tasti, ma sull'interazione diretta sullo schermo. Il metodo riceve la posizione del *click* del *mouse* e ne verifica la sovrapposizione con i rettangoli precedentemente disegnati, restituendo direttamente il risultato della selezione (Listato 3.19).

```
    def click(self, pos):
        domanda = self.domanda_corrente()
        if domanda is None:
            return None

        # Controllo collisione tra cursore e rettangoli
        for indice, rect in enumerate(self.rettangoli):
            if rect.collidepoint(pos):
                corretta = (indice == domanda["corretta"])
                self.indice += 1
                return corretta # Restituisce un booleano al chiamante

        return None
```

Listato 3.19: ChatGPT - Validazione spaziale tramite *input* del *mouse*

giocatore.py

Con questo *script* si consolida l'organizzazione a oggetti introdotta dal modello, isolando la gestione del personaggio (grafica e movimento) in un'unica classe. La logica si semplifica notevolmente poiché non è più il ciclo principale a calcolare la posizione passo dopo passo. Il giocatore aggiorna le proprie coordinate in autonomia, avanzando verso la porta solo dopo aver ricevuto il comando di partenza e fermandosi in automatico una volta raggiunta la posizione del pallone (Listato 3.20).

```
class Giocatore:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.velocita = 3
        self.rincorsa = False
        self.calciato = False

    def muovi(self):
        # Il movimento avviene solo se il flag e attivo e sotto una certa soglia X
        if self.rincorsa and self.x < 240:
            self.x += self.velocita

    def inizia_rincorsa(self):
        self.rincorsa = True

    def reset(self):
        self.x = 200
        self.y = 350
        self.rincorsa = False
        self.calciato = False
```

Listato 3.20: ChatGPT - Classe `Giocatore`: stato e logica di rincorsa

Il metodo di *rendering* si occupa di disegnare il personaggio sullo schermo sfruttando le funzioni di `Pygame`. Utilizzando `self.x` e `self.y` come punti di riferimento, l'intera sagoma dello *stickman* si muove orizzontalmente mantenendo intatte le proporzioni del corpo, senza costringere il codice esterno a ricalcolarne ogni singolo segmento (Listato 3.21).

```
    def disegna(self, screen):
        x = self.x
        y = self.y

        # testa
        pygame.draw.circle(screen, (0,0,0), (x,y), 18)

        # corpo
        pygame.draw.line(screen, (0,0,0), (x,y+18), (x,y+70), 3)

        # braccio sinistro e destro
        pygame.draw.line(screen, (0,0,0), (x,y+35), (x-20,y+55), 3)
        pygame.draw.line(screen, (0,0,0), (x,y+35), (x+20,y+55), 3)

        # gamba sinistra e destra
        pygame.draw.line(screen, (0,0,0), (x,y+70), (x-15,y+100), 3)
        pygame.draw.line(screen, (0,0,0), (x,y+70), (x+15,y+100), 3)
```

Listato 3.21: ChatGPT - *Rendering* dinamico del rigorista

portiere.py

Analogamente alla classe `Giocatore`, lo stato interno è progettato per gestire autonomamente la propria animazione (il tuffo) tramite variabili di stato e di *offset*, svincolando il ciclo principale dal calcolo manuale delle coordinate (Listato 3.22).

```
class Portiere:
    def __init__(self, x, y):
```

```
        self.x = x
        self.y = y
        self.posizione_iniziale_y = y
        self.colore = (30, 144, 255)

        self.tuffo = False
        self.direzione = 0
        self.velocita = 6

    def reset(self):
        self.y = self.posizione_iniziale_y
        self.tuffo = False
        self.direzione = 0

    def inizia_tuffo(self):
        self.tuffo = True
        # per ora tuffo verso il basso
        # in futuro possiamo renderlo casuale
        self.direzione = 1
```

Listato 3.22: ChatGPT - Classe `Portiere`: inizializzazione e logica di stato

La traslazione fisica del tuffo viene calcolata dal metodo `aggiorna`. Invece di richiedere parametri esterni, la classe incrementa la propria coordinata Xb in base alla velocità impostata, interrompendo il movimento non appena l'*avatar* supera la soglia di 70 pixel rispetto alla posizione di partenza (Listato 3.23).

```
    def aggiorna(self):
        if self.tuffo:
            self.y += self.velocita

            if self.y > self.posizione_iniziale_y + 70:
                self.tuffo = False
```

Listato 3.23: ChatGPT - Classe `Portiere`: traslazione del tuffo

pallone.py

L'inizializzazione memorizza le coordinate di partenza per facilitare il ripristino tra un tiro e l'altro, utilizzando una variabile di stato per determinare se la palla deve essere soggetta a traslazione spaziale (Listato 3.24).

```
class Pallone:
    def __init__(self, x, y):
        # posizione iniziale e di partenza per il reset
        self.x = x
        self.y = y
        self.x_iniziale = x
        self.y_iniziale = y

        # velocita tiro e stato
        self.velocita_x = 8
        self.in_movimento = False

    def reset(self):
        self.x = self.x_iniziale
        self.y = self.y_iniziale
        self.in_movimento = False

    def calcia(self):
        self.in_movimento = True
```

Listato 3.24: ChatGPT - Classe `Pallone`: stato e inizializzazione

Il movimento è gestito da un metodo di aggiornamento che incrementa l'asse orizzontale. La classe dispone anche di un metodo di validazione spaziale (`arrivato_in_porta`) che non solo blocca l'animazione, ma restituisce un valore booleano (`True`) quando la palla

supera una certa coordinata, attivando, così, la transizione di stato nel ciclo principale del programma (Listato 3.25).

```
    def aggiorna(self):
        if self.in_movimento:
            self.x += self.velocita_x

    def arrivato_in_porta(self):
        if self.x >= 850:
            self.in_movimento = False
            return True

        return False
```

Listato 3.25: ChatGPT - Classe `Pallone`: traslazione e *trigger* di fine corsa

animazioni.py

L'inizializzazione crea l'istanza della palla, mentre il metodo di reset la riporta alla posizione di partenza. Il metodo dedicato al tiro aggiorna e disegna la traiettoria in tempo reale, interrompendo l'animazione non appena il pallone raggiunge la coordinata della porta. Infine, i metodi per il gol e la parata si occupano esclusivamente di stampare sul video i relativi testi di esito, senza più gestire alcun movimento (Listato 3.26).

```
from pallone import Pallone

class Animazione:
    def __init__(self):
        self.pallone = Pallone(260, 420)
        self.finita = False

    def reset(self):
        self.pallone.reset()
        self.finita = False
```

Listato 3.26: ChatGPT - Composizione ad oggetti e inizializzazione

main.py

Il modulo `main.py` agisce da gestore coordinatore per i diversi componenti di una macchina a stati finiti. A differenza dell'approccio procedurale, le transizioni di stato non si basano più su stringhe di testo (come "menu" o "gioco"), ma sono gestite esclusivamente tramite costanti numeriche (Listato 3.27).

```
# STATI
MENU = 0
TIRO = 1
DOMANDA = 2
GOL = 3
PARATA = 4

stato = MENU

# OGGETTI
quiz = Quiz()
animazione = Animazione()
giocatore = Giocatore(200, 350)
portiere = Portiere(880, 300)

score = 0
```

Listato 3.27: ChatGPT - Inizializzazione degli oggetti e macchina a stati

Nella gestione dei punteggi, il codice generato evidenzia un allontanamento dalle specifiche iniziali. Il modello non ha implementato il sistema duale richiesto, omettendo del tutto il

tracciamento dei tiri falliti (`Xb`). Viene utilizzata un'unica variabile globale `score`(corrispondente a `Xa`), che viene incrementata esclusivamente in caso di validazione positiva del *click* del *mouse* (Listato 3.28).

```
        # RISPOSTA QUIZ
        elif stato == DOMANDA:
            if event.type == pygame.MOUSEBUTTONDOWN:
                risposta = quiz.click(event.pos)

                if risposta is True:
                    score += 1 # Svolge la funzione della variabile Xa (Gol)
                    animazione.reset()
                    stato = GOL

                elif risposta is False:
                    animazione.reset()
                    portiere.inizia_tuffo()
                    stato = PARATA
```

Listato 3.28: ChatGPT - Gestione degli eventi e incremento di `Xa` (`score`)

Nel blocco di aggiornamento visivo, il main chiama i metodi di *rendering* delle rispettive classi. Si evidenzia, tuttavia, una criticità nella gestione delle schermate di esito: per ripristinare lo stato di TIRO, il modello ricorre a un controllo sui millisecondi di gioco. Tale soluzione risulta instabile, poiché vincola la durata delle schermate alla velocità di esecuzione del computer anziché affidarsi a un vero e proprio *timer* (Listato 3.29).

```
    elif stato == GOL:
        disegna_campo()
        giocatore.disegna(screen)
        portiere.disegna(screen)
        animazione.gol(screen)
        mostra_punteggio()

        # Condizione di uscita basata sul tick globale
        if pygame.time.get_ticks() % 120 == 0:
            animazione.reset()
            stato = TIRO

    elif stato == PARATA:
        # ... (chiamate di rendering omesse per brevita) ...
        portiere.aggiorna()
        animazione.parata(screen)

        if pygame.time.get_ticks() % 120 == 0:
            animazione.reset()
            portiere.reset()
            stato = TIRO
```

Listato 3.29: ChatGPT - *Rendering* degli esiti e transizione di stato temporizzata

Lista Domande

Definita l'architettura generale dello *script*, il passaggio successivo ha riguardato la generazione del *file* dedicato ai quesiti:

> **Utente**
>
> Crea il *file json* contenente le domande a tema.

L'obiettivo di questo *prompt* era isolare le domande all'interno di un archivio esterno strutturato. Il modello non solo ha generato correttamente il listato in formato JSON, ma ha anche sottolineato come la netta separazione tra logica applicativa e livello dei dati sia fondamentale per favorire la scalabilità e la manutenzione del progetto:

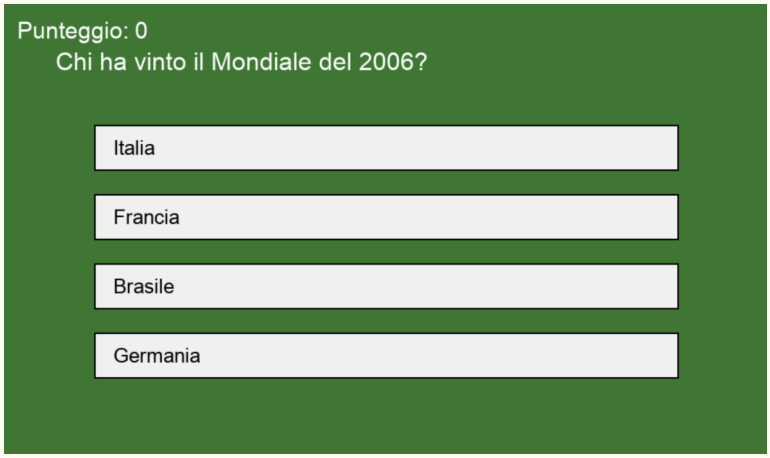
ChatGPT

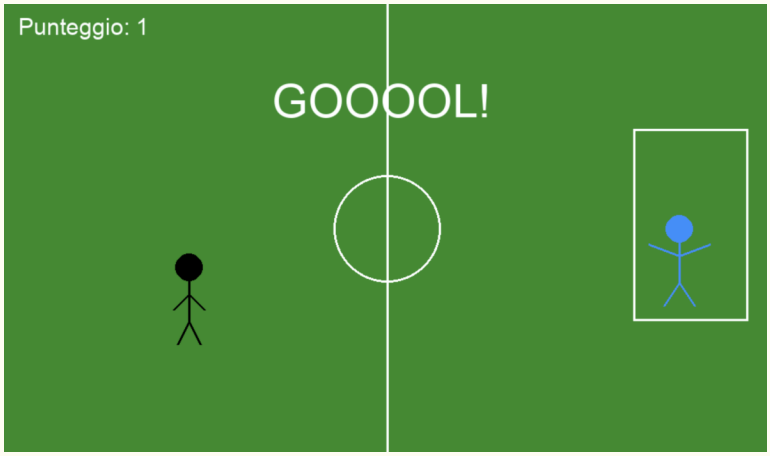
Ottimo. Ora rendiamo il progetto più professionale: spostiamo tutte le domande in un file esterno `domande.json`. Vantaggi:

- puoi aggiungere infinite domande senza modificare Python;
- il codice del quiz rimane pulito;
- in futuro potrai creare categorie (Serie A, Champions, Mondiali, giocatori, ecc.).

Inoltre, come suggerito dal modello, è stato aggiornato il *file* `main.py` per ottimizzare il ciclo di gioco complessivo. L'LLM mirava a rendere più fluide le transizioni di stato e a introdurre una schermata finale.

ChatGPT





Correzione Logica

Durante i *test* di questa versione si è verificato un blocco nella logica di gioco: dopo aver segnato un gol, lo *script* si interrompeva a causa della mancanza di un comando per passare alla domanda successiva. Inizialmente, il modello ha attribuito erroneamente il *bug* al file `main.py`. Tuttavia, a seguito di un inefficace tentativo di correzione e di un secondo *prompt* di sollecito, l'LLM ha riconsiderato il problema, individuando, così, nel modulo `animazioni.py` l'effettiva causa dell'anomalia e procedendo alla sua risoluzione. Di seguito il *prompt* utilizzato:

Utente

Quando l'utente risponde alle domande, dopo "il gol", non è presente nessun tasto/comando per continuare a giocare.

La risposta del modello ha confermato l'assenza di un blocco di attesa, riconoscendo la mancanza nel flusso di gioco:

ChatGPT

Perfetto, hai trovato un problema di flusso del gioco. Attualmente dopo il GOL o la PARATA il programma cambia stato automaticamente, ma non dà tempo al giocatore di leggere il risultato e soprattutto non offre un comando per continuare.

Per gestire fluidamente le pause dopo l'esecuzione del tiro, il controllo del tempo viene delegato al modulo `animazioni.py`. Attraverso un contatore che si incrementa durante la schermata di esito, il sistema attende il raggiungimento di una durata temporale prestabilita per segnalare la fine della fase visiva, sbloccando, così, la transizione verso il quesito successivo (Listato 3.30).

```
class Animazione:
    def __init__(self):
        self.pallone = Pallone(260, 420)
        self.finita = False
        self.timer = 0
        self.durata_risultato = 120   # circa 2 secondi a 60 FPS

    def reset(self):
        self.pallone.reset()
        self.finita = False
        self.timer = 0

    def gol(self, screen):
        self.timer += 1

        # ... (rendering testo "GOOOOL!" omesso per brevita) ...
        self.pallone.disegna(screen)

        if self.timer >= self.durata_risultato:
            self.finita = True

    def parata(self, screen):
        self.timer += 1

        # ... (rendering testo "PARATA!" omesso per brevita) ...

        if self.timer >= self.durata_risultato:
            self.finita = True
```

Listato 3.30: ChatGPT - Implementazione del *timer* interno per le animazioni

Modifica della Visuale dell'Utente

Per cercare di replicare l'interfaccia grafica del sistema precedente è stato chiesto al modello di modificare il punto di vista dell'utente, spostandolo alle spalle del tiratore:

Utente

Richiedo ulteriori modifiche grafiche: vorrei che la prospettiva non fosse laterale, ma che si vedesse "da dietro il tiratore".

Il modello ha proposto di operare una serie di modifiche su più *file* per, infine, riaggiornare il `main.py`.

ChatGPT

Non toccherei subito `main.py`, perché la logica funziona. Interverremo principalmente su:

1. `giocatore.py`: attualmente lo stickman è visto di lato. Lo trasformiamo in:
   - giocatore visto da dietro;
   - corpo più grande in basso allo schermo;
   - posizione fissa sulla battuta del rigore.
2. `campo.py` (nuovo *file* consigliato): in questo momento la logica di disegno è contenuta nel main, conviene spostarla in un file a parte;
3. `portiere.py`: il portiere deve essere ridimensionato, sarà:
   - più piccolo in lontananza;
   - al centro della porta;
   - con possibilità futura di tuffarsi a destra/sinistra.
4. `pallone.py`: il pallone dovrà simulare la profondità.

Successivamete viene riportata la logica del nuovo *file* che, insieme alle modifiche degli altri *script*, permette la creazione della nuova visualizzazione (Listato 3.31 e 3.32).

```python
class Campo:
    def __init__(self, width, height):
        self.width = width
        self.height = height

        # colori
        self.erba = (34, 139, 34)
        self.bianco = (255, 255, 255)
```

Listato 3.31: ChatGPT - Classe `Campo`: Inizializzazione

```python
    def disegna(self, screen):
        screen.fill(self.erba)

        # linee laterali del campo in prospettiva
        pygame.draw.line(screen, self.bianco, (120,600), (390,250), 4)
        pygame.draw.line(screen, self.bianco, (880,600), (610,250), 4)

        # ... (disegno area e dischetto omesso per brevita) ...

        # porta
        pygame.draw.rect(screen, self.bianco, (350,150,300,120), 5)
        pygame.draw.line(screen, self.bianco, (350,150), (650,150), 5)

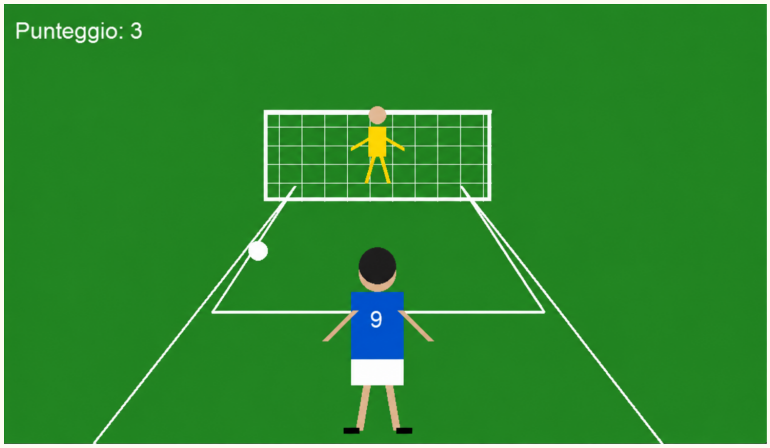
        # rete stilizzata tramite cicli
        for x in range(370,650,30):
            pygame.draw.line(screen, self.bianco, (x,150), (x,270), 1)

        for y in range(170,270,25):
            pygame.draw.line(screen, self.bianco, (350,y), (650,y), 1)
```

Listato 3.32: ChatGPT - *Rendering* della prospettiva e della rete

L'*output*, però, mostrava inizialmente problemi riguardo le proporzioni delle linee del campo.

ChatGPT



Per cercare di risolvere il problema sono state effettuate specifiche richieste al modello, senza, però, riuscire a replicare il risultato ottenuto con il primo LLM.

Aggiunta dei Tifosi

Al fine di allineare l'*output* visivo a quello del primo LLM analizzato, è stata effettuata una richiesta specifica per l'inserimento dei tifosi alle spalle del portiere:

Utente

Nello sfondo, alle spalle del portiere, vorrei aggiungere una gradinata con i tifosi.

In risposta, il modello ha proposto di effetture modifiche specifiche al *file* `campo.py`:

ChatGPT

Ottime modifiche. In questo caso non dobbiamo toccare il `main.py`, perché la grafica del campo è completamente gestita da `campo.py`. Dobbiamo intervenire solo su `campo.py`:

1. separare meglio le linee dell'area piccola e grande;
2. aggiungere uno sfondo con gradinata e tifosi.

Per la richiesta relativa all'inserimento dei tifosi sullo sfondo il modello ha dato *output* soddisfacenti, mentre, anche con ulteriori richieste, per la realizzazione del campo non è riuscito a replicare il risultato del modello precedente. Successivamente vengono riportati frammenti di codice relativi alla logica di `campo.py` per la realizzazione della gradinata (Listato 3.33).

```
    def crea_tifosi(self):
        for fila in range(5):
            for i in range(30):
                self.tifosi.append(
                    (
                        40 + i * 32,
                        40 + fila * 18,
                        random.choice([
                            (255,0,0),
                            (0,0,255),
                            (255,255,0),
                            (255,255,255)
                        ])
                    )
```

```
            )

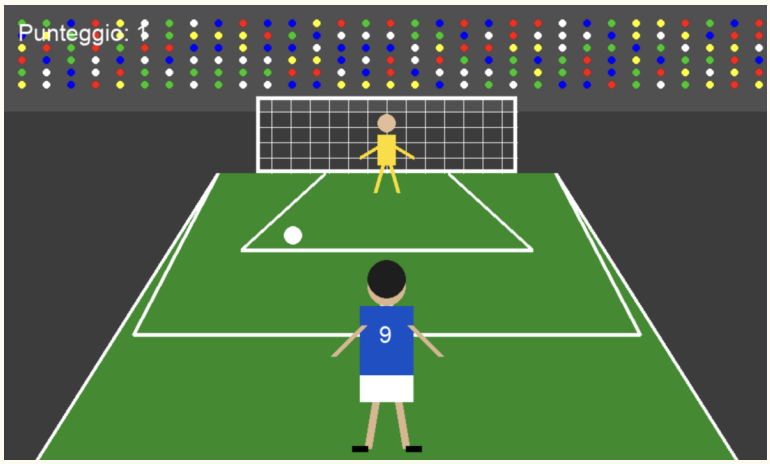
    def disegna(self, screen):
        # ======================
        # SFONDO STADIO E GRADINATA
        # ======================
        screen.fill((70,70,70))

        pygame.draw.rect(
            screen,
            self.gradinata,
            (0, 0, self.width, 140)
        )

        # Rendering file dei tifosi
        for tifoso in self.tifosi:
            pygame.draw.circle(
                screen,
                tifoso[2],
                (tifoso[0], tifoso[1]),
                5
            )
```

Listato 3.33: ChatGPT - Generazione procedurale dei tifosi e della gradinata

ChatGPT



Implementazione del Timer

Al fine di replicare interamente il comportamento logico del risultato ottenuto con il primo modello, è stata richiesta l'Implementazione di un *timer* di dieci secondi:

Utente

Vorre mettere un timer di 10 secondi per poter rispondere alla domanda, al termine del quale il calciatore tirerà la palla fuori dallo specchio della porta (verso destra).

Il modello ha inizialmente tentato di intervenire su diversi *file*; tuttavia, al termine delle modifiche richieste, quando il calciatore arrivava al tiro, non comparivano la domanda e il *timer*. A seguito di un ulteriore sollecito, l'LLM ha affermato di avere una struttura ormai troppo complessa da gestire e ha suggerito un *reset* del codice, pur riproponendo comunque il medesimo approccio appena fallito:

ChatGPT

Siamo arrivati a un punto in cui conviene fare un piccolo "reset" del codice Per farlo, ricostruiamo un progetto coerente con questi file:

- `main.py`;
- `campo.py`;
- `animazioni.py`;
- `giocatore.py`;
- `portiere.py`;
- `pallone.py`;
- `quiz.py`.

3.2.3 Generazione con Copilot

Generazione del Codice di Base

L'analisi prosegue con la valutazione di Copilot, il terzo dei quattro modelli selezionati. Mantenendo invariata l'impostazione sperimentale, l'interazione inizia fornendo in *input* la stessa istruzione di partenza utilizzata per i sistemi precedenti:

Utente

Dovrei creare un codice Python per la realizzazione di un quiz calcistico. Per la realizzazione avevo immaginato 2 calciatori (*stickman*), 1 rigorista ed un portiere, dove al momento dell'impatto con il pallone del rigorista compare la domanda alla quale l'utente dovrà rispondere, se la risposta sarà corretta il calciatore segnerà, altrimenti il pallone verrà parato dal portiere.

La risposta del sistema è stata immediata, validando l'idea di base e suggerendo fin dall'inizio l'impiego di `Pygame` per gestire simultaneamente le logiche del *quiz* e il comparto grafico:

Copilot

Certo, l'idea si presta bene a un piccolo gioco in Python con `Pygame`: animazione del rigore, comparsa della domanda al momento del tiro, esito diverso in base alla risposta.

Copilot racchiude l'intero gioco in un unico *script* senza dividere il codice in classi, ma utilizza una macchina a stati finiti controllata da stringhe testuali anziché da numeri, avvicinandosi concettualmente (ma non sintatticamente) all'architettura a stati di ChatGPT. La vera discrepanza rispetto agli altri modelli si nota nell'interazione dell'utente con il quiz. Il modello implementa, infatti, una casella di testo a inserimento libero (Listato 3.34)

```
questions = [
    {
        "question": "Come si chiama il tiro dagli 11 metri?",
        "answers": ["rigore", "calcio di rigore"]
    },
    {
        "question": "Come si chiama quando la palla entra in porta?",
        "answers": ["gol", "goal", "rete"]
    }
]

def check_answer(answer, accepted_answers):
```

```
    # Pulisce la stringa e la converte in minuscolo per evitare errori di case-
    sensitivity
    answer = answer.strip().lower()
    return answer in accepted_answers
```

Listato 3.34: Copilot - Dizionario con varianti di risposta a inserimento libero

All'interno del ciclo degli eventi, la gestione dell'*input* intercetta la pressione di ogni singolo tasto. Il sistema costruisce dinamicamente la stringa digitata (`user_answer`), supporta il tasto Backspace per le correzioni e utilizza Invio come comando per validare la risposta, confrontandola con l'*array* delle soluzioni accettate (Listato 3.35).

```
        elif state == "QUESTION":
            # Validazione
            if event.key == pygame.K_RETURN:
                current_question = questions[current_question_index]
                correct = check_answer(user_answer, current_question["answers"])
                start_shot(correct)

            # Correzione
            elif event.key == pygame.K_BACKSPACE:
                user_answer = user_answer[:-1]

            # Inserimento caratteri
            else:
                if len(user_answer) < 30:
                    user_answer += event.unicode
```

Listato 3.35: Copilot - Acquisizione dinamica dei caratteri da tastiera

Anche dal punto di vista della fisica, Copilot introduce una novità rispetto ai modelli precedenti, ovvero l'interpolazione lineare. Invece di incrementare semplicemente le coordinate a ogni *frame* a velocità fissa, il modello calcola progressivamente la percentuale di viaggio compiuta dalla palla (`shot_progress`) verso un *target* dinamico, calcolato in base all'esito della risposta.

Come ChatGPT, anche Copilot ha ignorato il conteggio separato degli errori, omettendo totalmente la variabile `Xb` e utilizzando un'unica variabile `score` in sostituzione di `Xa` (Listato 3.36).

```
def start_shot(correct):
    # ... (omesse dichiarazioni global) ...
    shot_correct = correct
    ball_start_x = ball_x
    ball_start_y = ball_y
    shot_progress = 0

    if correct:
        # Se la risposta e corretta: tiro angolato in porta
        ball_target_x = 830
        ball_target_y = 245
        state = "SHOT_GOAL"
    else:
        # Se la risposta e sbagliata: tiro centrale, parato
        ball_target_x = keeper_x - 50
        ball_target_y = keeper_y - 20
        state = "SHOT_SAVE"

def move_ball():
    # ... (omesse dichiarazioni global) ...
    shot_progress += 0.025

    if shot_progress > 1:
        shot_progress = 1

    # Formula dell'interpolazione lineare per il movimento
    ball_x = ball_start_x + (ball_target_x - ball_start_x) * shot_progress
    ball_y = ball_start_y + (ball_target_y - ball_start_y) * shot_progress

```

```
    # Trigger di fine animazione
    if shot_progress >= 1:
        if shot_correct:
            score += 1
```

Listato 3.36: Copilot - Motore fisico basato su interpolazione lineare

Copilot



Modifiche Grafiche

Come per i modelli analizzati in precedenza, l'interazione successiva aveva come scopo quello di adeguare la prospettiva visiva, spostando l'inquadratura alle spalle del tiratore:

Utente

Vorrei apportare delle modifiche grafiche: imposta il campo con una prospettiva frontale anziché laterale, adottando una visuale alle spalle del rigorista.

Copilot

Perfetto, allora cambiamo la prospettiva: invece di vedere il campo lateralmente, l'utente vedrà la scena da dietro il rigorista, con la porta in alto e il pallone che si muove verso la porta.

Di seguito si riportano le modifiche principali al codice. Tutti gli elementi sono stati allineati centralmente rispetto alla larghezza dello schermo. Di conseguenza, la logica del tiro ha subito un aggiornamento: in caso di esito positivo, il pallone non viaggia più verso un generico lato destro, ma il modello seleziona casualmente un angolo (sinistro o destro) rispetto al centro della porta, sfruttando l'interpolazione lineare già implementata (Listato 3.37).

```
# Coordinate principali in vista frontale (allineamento centrale)
penalty_spot_x = WIDTH // 2
penalty_spot_y = 430

kicker_x = WIDTH // 2
kicker_y = 540

keeper_x = WIDTH // 2
keeper_y = 175

def start_shot(correct):
    # ... (omesse dichiarazioni global) ...
    shot_correct = correct
```

```
    ball_start_x = ball_x
    ball_start_y = ball_y
    shot_progress = 0

    if correct:
        # Tiro angolato in porta scegliendo un lato casuale
        side = random.choice(["left", "right"])
        if side == "left":
            ball_target_x = WIDTH // 2 - 110
        else:
            ball_target_x = WIDTH // 2 + 110
        ball_target_y = 100
        state = "SHOT_GOAL"

    else:
        # Tiro centrale, parato dal portiere
        ball_target_x = keeper_x
        ball_target_y = keeper_y
        state = "SHOT_SAVE"
```

Listato 3.37: Copilot - Riorganizzazione spaziale e *target* del tiro in profondita

Per enfatizzare l'illusione ottica della tridimensionalità, il campo è stato disegnato utilizzando bande verticali alternate, una soluzione adottata per accentuare il senso di profondità spaziale. Inoltre, la funzione dedicata alla generazione del giocatore è stata riscritta per tracciare una sagoma vista di schiena, allargando il segmento delle spalle e omettendo i dettagli frontali (Listato 3.38).

```
def draw_field():
    screen.fill(GREEN)

    # Strisce verticali del campo per dare profondita
    stripe_width = WIDTH // 6
    for i in range(6):
        color = GREEN if i % 2 == 0 else DARK_GREEN
        pygame.draw.rect(screen, color, (i * stripe_width, 0, stripe_width, HEIGHT))

    # Porta in alto, vista frontalmente centrata
    goal_x = WIDTH // 2 - 150
    goal_y = 55
    goal_w = 300
    goal_h = 120
    pygame.draw.rect(screen, GREY, (goal_x, goal_y, goal_w, goal_h), 6)

    # ... (disegno area e dischetto omesso per brevita) ...

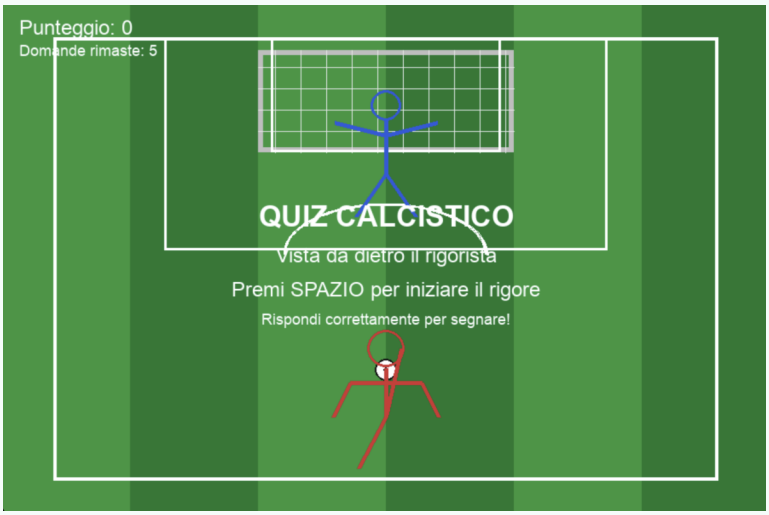
def draw_kicker_back(x, y, kicking=False):
    color = RED

    # Corpo centrale
    pygame.draw.circle(screen, color, (int(x), int(y - 95)), 22, 3)
    pygame.draw.line(screen, color, (x, y - 72), (x, y - 15), 5)

    # Spalle larghe e braccia laterali
    pygame.draw.line(screen, color, (x - 42, y - 55), (x + 42, y - 55), 5)
    pygame.draw.line(screen, color, (x - 42, y - 55), (x - 62, y - 15), 5)
    pygame.draw.line(screen, color, (x + 42, y - 55), (x + 62, y - 15), 5)
```

Listato 3.38: Copilot - Espedienti grafici per l'illusione della profondita

Copilot



Aggiunta dei Tifosi e Correzioni Grafiche

Con questa richiesta, oltre ad aggiungere la presenza di tifosi, sono state modificati elementi grafici e, per allinearsi alla logica dei modelli precedenti, si è corretta la modalità di risposta, sostituendo alla risposta aperta la modalità di risposta chiusa:

Utente

Vorrei fare altre modifiche:

1. aggiungi la presenza di tifosi dietro la porta;
2. aumenta leggermente la dimensione della palla;
3. sostituisci "Punteggio" con "Gol" aggiungi in alto a destra "Parate";
4. aggiungi una rincorsa da quando si avvia il programma a quando il calciatore arriva davanti alla palla, al termine della quale scatterà un timer di 10 secondi per rispondere alla domanda, se l'utente non seleziona nessuna risposta in tempo il calciatore tirerà fuori dallo specchio della porta.

Con questo *prompt* l'aggiornamento si è articolato su due fronti: da un lato l'arricchimento estetico e dell'interfaccia (il pubblico, il dimensionamento della sfera, i nuovi indicatori di punteggio), dall'altro una sostanziale evoluzione delle meccaniche di gioco, tramite l'inserimento di un'animazione preliminare e di un *timer* di risposta:

Copilot

Certo, queste modifiche rendono il gioco molto più simile a un vero rigore visto dagli occhi del tiratore. Ho aggiornato la logica in modo che il giocatore parta subito con la rincorsa, poi compaia una domanda a scelta multipla con timer da 10 secondi.

La prima modifica sostanziale riguarda la sostituzione della singola variabile `score` con le variabili `goals` (`Xa`) e `saves` (`Xb`). Di conseguenza la funzione di *rendering* dell'interfaccia è stata aggiornata per visualizzare i due contatori agli angoli opposti dello schermo. Inoltre, è stata incrementata la costante del raggio del pallone (Listato 3.39).

```
1 # VARIABILI DI GIOCO
```

```
goals = 0  # Rappresenta Xa
saves = 0  # Rappresenta Xb

ball_radius = 17  # Palla leggermente piu grande

def draw_scoreboard():
    gol_text = font_medium.render(f"Gol: {goals}", True, WHITE)
    screen.blit(gol_text, (20, 15))

    parate_text = font_medium.render(f"Parate: {saves}", True, WHITE)
    parate_rect = parate_text.get_rect(topright=(WIDTH - 20, 15))
    screen.blit(parate_text, parate_rect)
```

Listato 3.39: Copilot - Implementazione del punteggio duale (`goals` e `saves`)

La gestione della rincorsa e del *timer* si integra in modo naturale nel ciclo di gioco. L'avvicinamento del calciatore al pallone innesca automaticamente la comparsa del *quiz* e l'avvio del conto alla rovescia. Nel caso in cui l'utente non dovesse fornire una risposta entro i dieci secondi consentiti, il sistema forza in automatico la convalida di un tiro fallito (Listato 3.40).

```
    # LOGICA DEGLI STATI
    if state == "RUNUP":
        kicker_y -= 2.3

        # Quando il calciatore arriva davanti alla palla parte la domanda
        if kicker_y <= 500:
            start_question() # Salva pygame.time.get_ticks() in question_start_time

    elif state == "QUESTION":
        elapsed = (pygame.time.get_ticks() - question_start_time) / 1000

        # Se il tempo scade, il calciatore tira fuori
        if elapsed >= QUESTION_TIME_LIMIT:
            start_shot("out")
```

Listato 3.40: Copilot - Logica di transizione tra rincorsa, *quiz* e *timeout*

Per gestire il tiro fuori bersaglio, il sistema sfrutta la logica di interpolazione lineare che aveva precedentemente impostato. Invece di programmare una nuova animazione da zero, aggiunge una condizione alla funzione `start_shot` che calcola un bersaglio (`ball_target_x` e `ball_target_y`) ampiamente all'esterno delle coordinate della porta, lasciando che la logica di movimento già esistente sposti il pallone verso quel punto (Listato 3.41).

```
def start_shot(result):
    # ... (inizializzazione variabili omessa per brevita) ...

    if result == "goal":
        # ... logica gol ...
        state = "SHOT_GOAL"

    elif result == "save":
        # ... logica parata ...
        state = "SHOT_SAVE"

    elif result == "out":
        # Calcolo di un bersaglio esterno alla porta
        side = random.choice(["left", "right"])

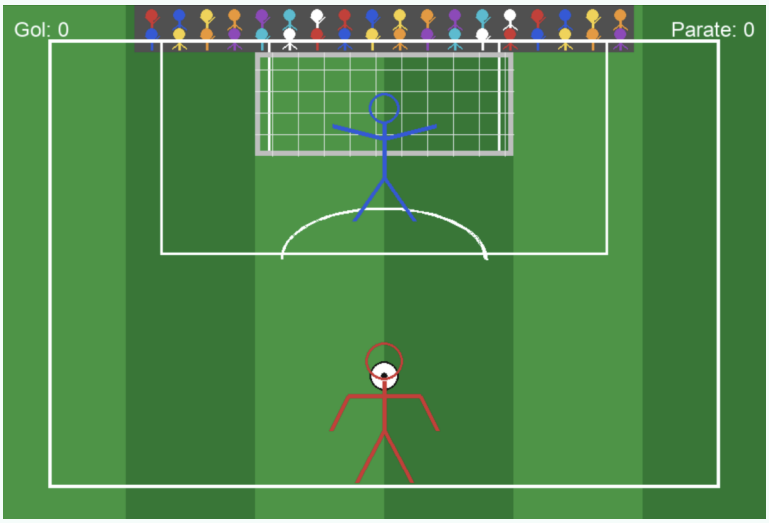
        if side == "left":
            ball_target_x = WIDTH // 2 - 230
        else:
            ball_target_x = WIDTH // 2 + 230

        ball_target_y = 80
        state = "SHOT_OUT"
```

Listato 3.41: Copilot - Calcolo del *target* spaziale per il tiro fuori

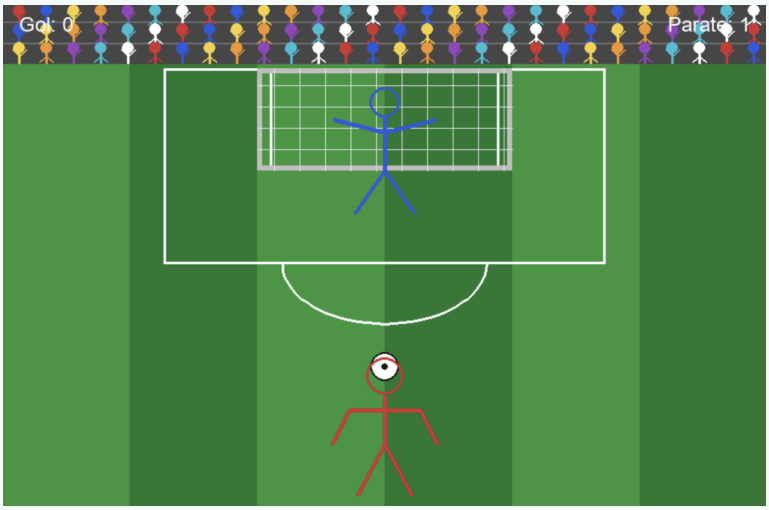
Il codice include la funzione `draw_fans()`, impiegata per generare due file di spettatori dietro la porta; disegnando cerchi e linee alternate, questa soluzione contribuisce a conferire ulteriore profondità alla scena:

Copilot



Un'ulteriore interazione con l'LLM è stata dedicata al perfezionamento dell'interfaccia grafica. Tali correzioni estetiche apportate agli elementi di gioco hanno prodotto il seguente risultato visivo:

Copilot



Divise e Perfezionamenti dei Grafici

Per perfezionare l'estetica degli elementi presenti, all'LLM è stato richiesto di modificare la scala dei calciatori, introdurre variazioni cromatiche per distinguerne le funzioni e allargare la prospettiva dell'area di rigore:

Utente

Vorrei rendere i 2 calciatori leggermente più piccoli, ed entrambi del classico colore degli stickman, ma differenziati da una maglia rossa per il tiratore e verde per il portiere (deve avere anche i guanti). Inoltre vorrei rendere l'area di rigore più larga.

Il sistema ha confermato di aver compreso correttamente le richieste e, individuato le aree su cui agire per apportare le modifiche:

Copilot

Certo, per ottenere questo effetto ti basta sostituire alcune funzioni del codice precedente. Le modifiche principali sono: stickman più piccoli, corpo nero stile classico, maglia rossa/verde sopra il corpo, guanti per il portiere e area di rigore più larga.

Per il portiere e il rigorista, il modello non si è limitato a cambiare il colore delle linee del corpo, ma ha aggiunto delle forme geometriche sovrapposte (poligoni) per simulare le divise da gioco e i guanti, come riportato dallo *snippet* (Listato 3.42).

```
def draw_kicker_back(x, y, kicking=False):
    stick_color = BLACK
    shirt_color = RED

    # Testa e corpo base
    pygame.draw.circle(screen, stick_color, (int(x), int(y - 82)), 17, 3)
    pygame.draw.line(screen, stick_color, (x, y - 64), (x, y - 18), 4)

    # Aggiunta maglia rossa tramite poligoni sovrapposti
    pygame.draw.polygon(
        screen,
        shirt_color,
        [
            (x - 26, y - 58),
            (x + 26, y - 58),
            (x + 18, y - 25),
            (x - 18, y - 25)
        ]
    )
    # Contorno della maglia
    pygame.draw.polygon(
        screen, stick_color,
        [(x - 26, y - 58), (x + 26, y - 58), (x + 18, y - 25), (x - 18, y - 25)],
        2
    )

    # ... (disegno braccia e gambe omesse per brevita) ...
```

Listato 3.42: Copilot - *Rendering* dettagliato del rigorista con divisa

Per quanto riguarda il portiere, oltre alla divisa verde (`GOALKEEPER_GREEN`), il codice include una funzione di appoggio dedicata al disegno dei guanti. Questi vengono rappresentati come piccoli cerchi bianchi con bordo nero applicati all'estremità delle braccia (Listato 3.43).

```
def draw_glove(x, y):
    pygame.draw.circle(screen, WHITE, (int(x), int(y)), 8)
    pygame.draw.circle(screen, BLACK, (int(x), int(y)), 8, 2)

def draw_keeper_front(x, y, diving=False, direction="left"):
    # ... (gestione logica del tuffo omessa per brevita) ...

    # Esempio posizionamento guanti in posizione di attesa
    left_hand = (x - 52, y - 28)
    right_hand = (x + 52, y - 28)

    pygame.draw.line(screen, stick_color, (x, y - 18), left_hand, 4)
```

```
    pygame.draw.line(screen, stick_color, (x, y - 18), right_hand, 4)

    draw_glove(left_hand[0], left_hand[1])
    draw_glove(right_hand[0], right_hand[1])
```

Listato 3.43: Copilot - *Rendering* dettagliato del portiere con guanti

Per il campo, la dimensione dell'area di rigore è stata ampliata modificando le variabili di larghezza e altezza dei rettangoli che la compongono. È stata, anche, aggiunta la lunetta dell'area di rigore, disegnata calcolando un arco geometrico usando le funzioni `seno` e `coseno` del modulo `math` (Listato 3.44).

```
    # Area di rigore piu larga
    area_x = 120
    area_y = FIELD_TOP + 80
    area_w = 660
    area_h = 230
    area_bottom = area_y + area_h

    pygame.draw.rect(screen, WHITE, (area_x, area_y, area_w, area_h), 3)

    # Lunetta esterna all'area grande (verso il rigorista)
    arc_points = []
    arc_center_x = WIDTH // 2
    arc_center_y = area_bottom
    arc_radius_x = 120
    arc_radius_y = 70

    for angle in range(0, 181, 5):
        rad = math.radians(angle)
        x = arc_center_x + math.cos(rad) * arc_radius_x
        y = arc_center_y + math.sin(rad) * arc_radius_y
        arc_points.append((x, y))

    pygame.draw.lines(screen, WHITE, False, arc_points, 3)
```

Listato 3.44: Copilot - Ricalcolo delle proporzioni dell'area e aggiunta della lunetta tramite trigonometria

Copilot



3.2.4 Generazione con Claude

Generazione del Codice di Base

L'indagine è stata avviata fornendo in *input* la solita istruzione iniziale:

Utente

Dovrei creare un codice Python per la realizzazione di un quiz calcistico. Per la realizzazione avevo immaginato 2 calciatori (stickman), 1 rigorista ed un portiere, dove al momento dell'impatto con il pallone del rigorista compare la domanda alla quale l'utente dovrà rispondere, se la risposta sarà corretta il calciatore segnerà, altrimenti il pallone verrà parato dal portiere.

Il modello ha confermato la fattibilità del progetto e la scelta della libreria `Pygame`, delineando le meccaniche di gioco per dimostrare la corretta comprensione della richiesta:

Claude

Ti preparo uno script Python con `Pygame` che implementa esattamente la meccanica che hai descritto: rigorista e portiere stickman, il pallone parte, all'impatto compare la domanda, risposta giusta = gol, sbagliata = parata.

Dal punto di vista strutturale, Claude opta per la programmazione orientata agli oggetti, ma, a differenza di ChatGPT (che aveva frammentato il codice in moduli separati per giocatore, portiere e pallone), incapsula l'intera logica, la macchina a stati e il ciclo degli eventi all'interno di un'unica classe `Game` (Listato 3.45).

```
# STATI DI GIOCO
STATE_AIM = "aim"          # in attesa che il giocatore calci
STATE_KICKING = "kicking"  # pallone in movimento verso il punto di impatto
STATE_QUESTION = "question"  # domanda visibile, in attesa di risposta
STATE_RESULT = "result"    # mostra esito (gol o parata)
STATE_GAMEOVER = "gameover"

class Game:
    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode((WIDTH, HEIGHT))

        self.state = STATE_AIM
        self.score = 0  # Variabile Xa (manca Xb)
        self.round_num = 0

        # Coordinate della palla
        self.ball_x = float(PENALTY_SPOT_X)
        self.ball_y = float(GROUND_Y)
```

Listato 3.45: Claude - Macchina a stati e classe `Game` monolitica

La vera particolarità della soluzione di Claude risiede nella gestione della fisica e dell'animazione. Il modello interpreta la dinamica dell'azione dividendo il tiro in due fasi distinte: una volta calciato, il pallone inizia la sua traiettoria per poi arrestarsi a metà campo (al raggiungimento della coordinata `IMPACT_X`), condizione che innesca automaticamente la visualizzazione del *quiz* (Listato 3.46).

```
    def update(self):
        if self.state == STATE_KICKING:
            if not hasattr(self, "outcome") or self.outcome is None:
                # fase 1: il pallone va dal dischetto al punto di impatto
                self.ball_x += 6

                if self.ball_x >= IMPACT_X:
                    self.ball_x = IMPACT_X
                    self.state = STATE_QUESTION # Mette in pausa il gioco per la domanda
                    self.build_options()
            else:
                # fase 2: il pallone riprende a muoversi dopo la risposta
                if self.outcome == "goal":
                    self.ball_x += 8
                    self.ball_y += (self.goal_target_y - self.ball_start_y) * 0.02
                    if self.ball_x >= GOAL_X + 60:
```

```
                        self.state = STATE_RESULT
                        self.outcome = None
```

Listato 3.46: Claude - Metodo `update`: divisione del tiro in due fasi

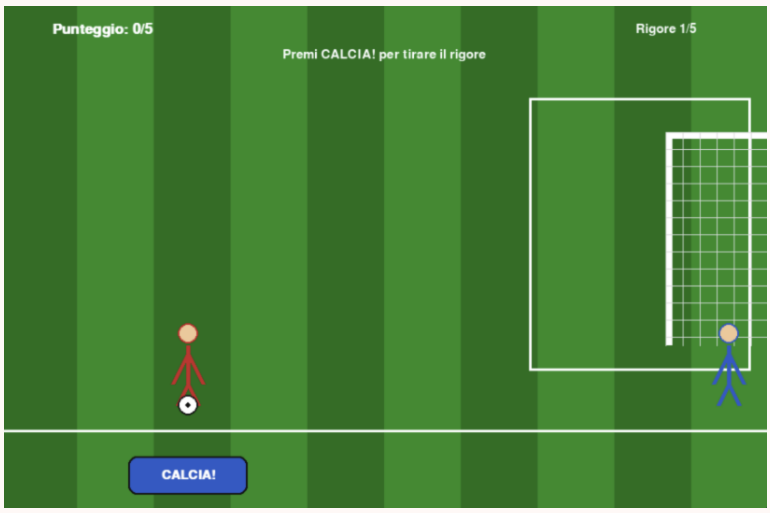
La funzione che elabora la scelta dell'utente (sia tramite *click* del *mouse* che tramite i tasti 1-4 della tastiera) non sposta istantaneamente il pallone, ma imposta la variabile interna `self.outcome` e riattiva lo stato `STATE_KICKING`, attivando, così, la seconda fase dell'animazione vista nello *snippet* precedente (Listato 3.47).

```
    def answer_selected(self, index):
        self.selected_option = index
        correct = self.current_question["corretta"]

        if index == correct:
            self.score += 1
            self.result_text = "GOOOL! Risposta corretta!"
            self.result_color = YELLOW
            self.state = STATE_KICKING  # riprende il movimento verso la porta
            self.outcome = "goal"
        else:
            giusta = self.current_question["opzioni"][correct]
            self.result_text = f"PARATO! La risposta giusta era: {giusta}"
            self.result_color = RED
            self.state = STATE_KICKING
            self.outcome = "save"
```

Listato 3.47: Claude - Validazione della risposta e sblocco dell'animazione

Claude



Con un *prompt* è stato richiesto al modello di limitare il dominio delle domande al campo dei Mondiali di calcio.

Modifica della Prospettiva

Ottenuta la prima versione, è stato sottoposto al modello un nuovo *prompt* per riposizionare la telecamera virtuale alle spalle del rigorista:

Utente

Vorrei apportare delle modifiche grafiche, metti il campo con vista frontale e non laterale, l'utente dovrà avere la visuale da dietro il rigorista.

Il modello non si è limitato a traslare le coordinate bidimensionali, ma ha implementato un vero e proprio sistema di simulazione prospettica. Come descritto nella sua risposta, l'LLM ha ricreato la profondità di campo tramite il rimpicciolimento dinamico del pallone e la proiezione prospettica dell'area, ricalibrando in modo coerente anche le logiche direzionali relative al tuffo del portiere:

Claude

Ho rifatto la grafica come chiesto — ora la visuale è frontale, con la telecamera posizionata dietro il rigorista, come nei videogiochi di calcio durante i rigori:

- rigorista: in primo piano, visto di spalle (niente volto, solo testa/schiena/gambe), con la gamba che si estende in avanti quando calcia;
- porta e portiere: in lontananza al centro dello schermo, in prospettiva (più piccoli, oltre l'orizzonte del campo);
- campo: disegnato come trapezio prospettico (stretto vicino alla porta, largo verso il giocatore), con area di rigore e dischetto;
- pallone: parte grande vicino ai piedi del rigorista e rimpicciolisce mentre vola verso la porta, simulando la profondità;
- portiere: il tuffo è ora sinistra/destra/centro rispetto allo spettatore (coerente con la vista frontale), e si tuffa dalla parte in cui "aveva intuito" il tiro — se sbagli la domanda, il pallone va proprio lì e viene parato; se rispondi bene, va dall'altra parte e segni.

La simulazione della profondità di campo è ottenuta modificando le variabili di *rendering*. Per dare l'illusione della distanza, il pallone varia il proprio raggio durante il volo e le strisce orizzontali del campo vengono sostituire da una griglia prospettica che si restringe gradualmente verso il fondo della scena (Listato 3.48).

```
# Geometria prospettica
HORIZON_Y = 205          # linea dell'orizzonte (porta)
GROUND_NEAR_Y = 560      # vicino alla telecamera (rigorista)

# pallone: parte grande, arriva piccolo (prospettiva)
BALL_START_RADIUS = 17
BALL_END_RADIUS = 6

def draw_field(surface):
    # ... (disegno cielo e trapezio campo omessi per brevita) ...

    # strisce del prato in prospettiva (bande convergenti all'orizzonte)
    n_stripes = 10
    for i in range(n_stripes):
        t0 = i / n_stripes
        t1 = (i + 1) / n_stripes
        if i % 2 == 1:
            continue

        # Calcolo vertici del trapezio per ogni singola striscia
        top_left = CENTER_X - field_top_half + 2 * field_top_half * t0
        top_right = CENTER_X - field_top_half + 2 * field_top_half * t1
        bot_left = CENTER_X - field_bottom_half + 2 * field_bottom_half * t0
        bot_right = CENTER_X - field_bottom_half + 2 * field_bottom_half * t1

        pygame.draw.polygon(
            surface, GREEN_DARK,
            [(top_left, HORIZON_Y), (top_right, HORIZON_Y), (bot_right, HEIGHT), (
    bot_left, HEIGHT)],
        )
```

Listato 3.48: Claude - Motore di *rendering* prospettico e punto di fuga

Invece di manipolare direttamente le singole coordinate, la traiettoria del pallone viene gestita tramite una variabile temporale che ne scandisce il tragitto. Il codice sfrutta una funzione di interpolazione lineare per aggiornare contemporaneamente la posizione orizzontale, quella verticale e il raggio della sfera, simulando, in questo modo, la profondità. Viene, inoltre, preservata la peculiare meccanica di pausa a metà traiettoria, che ferma l'azione per consentire la comparsa della domanda (Listato 3.49).

```
def lerp(a, b, t):
    return a + (b - a) * t

class Game:
    def update(self):
        # ... (controllo stato omesso per brevita) ...
        speed = 0.012  # avanzamento per frame
        self.t += speed

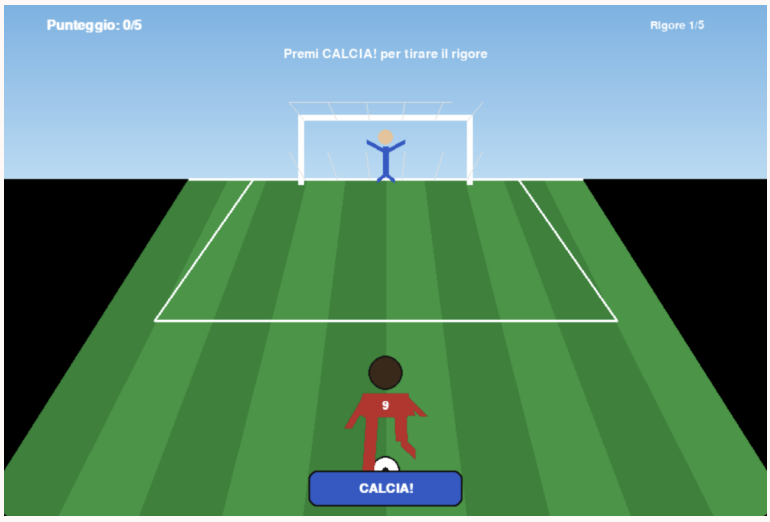
        if self.outcome is None:
            # fase 1: dal dischetto verso il punto di "impatto"
            if self.t >= IMPACT_T:
                self.t = IMPACT_T # Congela il tempo
                self._update_ball_position()
                self.state = STATE_QUESTION # Pausa per il quiz
                self.build_options()
                return
            self._update_ball_position()
        else:
            # fase 2: dal punto di impatto verso l'esito finale
            if self.t >= 1.0:
                self.t = 1.0
                # ... (gestione fine tiro omessa per brevita) ...

    def _update_ball_position(self):
        target_x = SHOT_ZONES[self.shot_zone_name] if self.outcome else CENTER_X
        target_y = TARGET_Y if self.outcome == "goal" else (GOAL_BOTTOM_Y - 6 if self.
    outcome == "save" else HORIZON_Y + 60)

        # Interpolazione contemporanea di X, Y e Z (raggio)
        self.ball_x = lerp(BALL_START_X, target_x, self.t)
        self.ball_y = lerp(BALL_START_Y, target_y, self.t)
        self.ball_radius = lerp(BALL_START_RADIUS, BALL_END_RADIUS, self.t)
```

Listato 3.49: Claude - Interpolazione lineare su 3 assi e "time freeze"

Claude



Aggiunta dei Tifosi e Modifiche Grafiche

Come per gli altri modelli è stata richiesta l'aggiunta dei tifosi dietro la porta, oltre a una serie di modifiche mirate per replicare lo stile precedentemente stabilito:

Utente

Vorrei effettuare ulteriori modifiche:

1. aggiungi la presenza di tifosi dietro la porta;
2. invece che "Punteggio" metti in alto a sinistra "Gol", con sotto numero di rigori rimanenti e, in alto a destra, "Parate";
3. aggiungi una rincorsa da quando si avvia il programma a quando il calciatore arriva davanti alla palla, al termine della quale scatterà un timer di 10 secondi per rispondere alla domanda, se l'utente non seleziona nessuna risposta in tempo il calciatore tirerà fuori dallo specchio della porta.

In questo caso è stato inserito un *prompt* con più richieste, relative sia alla parte grafica, sia a quella animata:

Claude

Fatto, ho implementato tutte e tre le modifiche:

1. Tifosi: ho aggiunto degli spalti dietro la porta, popolati da tanti puntini colorati (con colori casuali ma fissi, per evitare "sfarfallio") che simulano la folla, visibili sopra l'orizzonte dietro la rete.
2. Nuova HUD: in alto a sinistra ora c'è "Gol: X" con sotto "Rigori rimanenti: Y"; in alto a destra "Parate: Z".
3. Rincorsa + timer:
   - all'avvio di ogni rigore il calciatore parte piccolo e lontano (in prospettiva) e corre automaticamente verso il pallone, con una leggera oscillazione per simulare la corsa;
   - appena arriva, calcia automaticamente (non serve più premere alcun pulsante);
   - quando compare la domanda parte un timer di 10 secondi (mostrato sia in numero che con una barra che si svuota, diventa rossa negli ultimi 3 secondi);
   - se il tempo scade senza risposta, il tiro va a lato o sopra la traversa ("FUORI! Tempo scaduto"), non viene parato né segnato — e viene conteggiato separatamente ("Fuori") nel riepilogo finale.

Per l'interfaccia utente, è stato reintrodotto il sistema di punteggio duale (`Xa` per i gol, `Xb` per le parate). Inoltre, sono stati aggiunti anche il conteggio dei rigori rimasti e un contatore specifico per i tiri finiti fuori bersaglio, novità rispetto al codice precedente, (Listato 3.50).

```
class Game:
    def __init__(self):
        # ... (inizializzazione pygame) ...
        self.goals = 0   # Xa (Gol)
        self.saves = 0   # Xb (Parate)
        self.misses = 0  # Tiri fuori
        self.round_num = 0

    def draw(self):
        # ... (rendering campo e personaggi) ...

        # HUD: Gol in alto a sx, Rigori rimanenti sotto, Parate in alto a dx
```

```
13          remaining = max(0, TOTAL_ROUNDS - self.round_num + 1) if self.state !=
    STATE_GAMEOVER else 0
14          draw_text_left(self.screen, f"Gol: {self.goals}", 24, WHITE, (20, 16), bold=True
    )
15          draw_text_left(self.screen, f"Rigori rimanenti: {remaining}", 17, WHITE, (20,
    46))
16          draw_text_right(self.screen, f"Parate: {self.saves}", 24, WHITE, (WIDTH - 20,
    16), bold=True)
```

Listato 3.50: Claude - Sistema di punteggio completo e aggiornamento HUD

La fase di rincorsa, implementata attraverso il nuovo stato `STATE_RUNUP`, sfrutta una funzione matematica di variazione graduale della velocità per garantire un movimento più naturale. Il rigorista viene, inoltre, scalato in proporzione alla distanza per enfatizzare l'effetto prospettico di avvicinamento al pallone (Listato 3.51).

```
1   def draw_striker_for_state(self):
2       if self.state == STATE_RUNUP:
3           t = self.runup_frame / RUNUP_FRAMES
4           ease = t * t * (3 - 2 * t)  # easing morbido (accelera e poi rallenta)
5
6           # Il giocatore parte DIETRO al pallone e corre IN AVANTI
7           scale = lerp(1.18, 1.0, ease) # Si rimpicciolisce allontanandosi
8           y = lerp(RUNUP_START_Y, GROUND_NEAR_Y, ease)
9           x = CENTER_X + math.sin(ease * math.pi * 3) * 14 * (1 - ease)
10          wobble = math.sin(t * math.pi * 16) * (1 - ease * 0.5) # Effetto
    oscillazione
11
12          draw_striker_back(self.screen, x, y, scale, kick_progress=0.0, run_wobble=
    wobble)
```

Listato 3.51: Claude - Gestione matematica della rincorsa prospettica

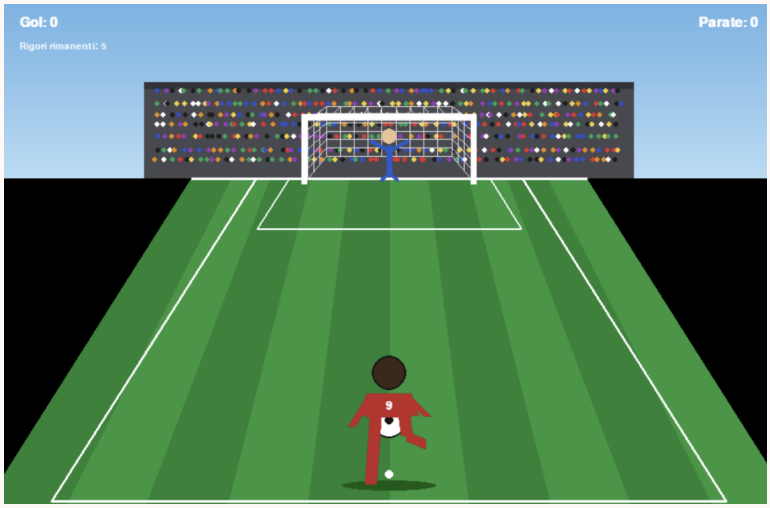
Una volta conclusa la rincorsa, il flusso di gioco avvia il *timer* per la risposta. In caso di mancata selezione da parte dell'utente, il sistema registra automaticamente l'errore e definisce una traiettoria di tiro volutamente fuori bersaglio, indirizzando la sfera verso coordinate esterne allo specchio della porta (Listato 3.52).

```
1   def update(self):
2       # ... (controllo stato) ...
3       if self.state == STATE_QUESTION:
4           if self.question_deadline is not None and pygame.time.get_ticks() >= self.
    question_deadline:
5               self.answer_timeout()
6           return
7
8   def answer_timeout(self):
9       self.misses += 1
10      self.result_text = "FUORI! Tempo scaduto, nessuna risposta data."
11      self.result_color = ORANGE
12      self.outcome = "miss"
13
14      direzione = random.choice(["wide_left", "wide_right", "high"])
15      if direzione == "wide_left":
16          self.miss_target = (GOAL_LEFT_X - 75, HORIZON_Y + 45)
17      elif direzione == "wide_right":
18          self.miss_target = (GOAL_RIGHT_X + 75, HORIZON_Y + 45)
19      else:
20          self.miss_target = (CENTER_X + random.uniform(-40, 40), GOAL_TOP_Y - 95)
21
22      self.keeper_pose = "idle"
23      self.question_deadline = None
24      self.state = STATE_KICKING
```

Listato 3.52: Claude - *Timeout* risposta e calcolo del tiro fuori prospettico

Sono state necessarie ulteriori richieste specifiche per correggere le linee del campo e le proporzioni, oltre alla rincorsa del giocatore (che inizialmente si muoveva all'indietro verso la palla).

Claude



Divise dei Giocatori e Correzioni dell'Animazione

Sebbene il modello avesse già generato il tiratore in divisa, si è reso necessario un ulteriore intervento per definire l'abbigliamento del portiere e risolvere un difetto nell'animazione: il pallone, infatti, compariva sullo schermo solo al termine della rincorsa del giocatore. Di seguito il *prompt* utilizzato:

Utente

Fa in modo che la palla sia già posizionata sul dischetto, non che compaia all'arrivo del giocatore. Inolte vorrei che i calciatori indossassero: per il tiratore una maglia rossa, per il portiere una verde (deve indossare anche i guanti).

In risposta a queste indicazioni, l'LLM ha aggiornato la logica di *rendering*. Come illustrato nel codice seguente è stata corretta la comparsa tardiva del pallone ed è stata implementata la personalizzazione cromatica per l'abbigliamento dei personaggi:

Claude

Fatto

1. Pallone sempre sul dischetto: ora è visibile fin dall'inizio, posizionato sul dischetto durante tutta la rincorsa, invece di comparire solo all'arrivo del giocatore.
2. Divise:
   - il rigorista aveva già la maglia rossa;
   - il portiere ora indossa una maglia verde invece che blu, e ha dei guanti gialli ben visibili su entrambe le mani, sia in posizione di attesa che durante il tuffo.

Di seguito viene riportato lo *snippet* con le principali modifiche (Listato 3.53).

```
    def draw(self):
        # ... (rendering campo) ...
        # Il pallone e sempre disegnato sul dischetto, indipendente dallo stato
        draw_ball(self.screen, self.ball_x, self.ball_y, self.ball_radius)
        self.draw_striker_for_state()

def draw_keeper(surface, zone_x, pose="idle"):
    # ... (calcolo coordinate corpo) ...
```

```
    if pose == "idle":
        # ... (disegno testa e corpo) ...
        hand_l = (x - 22, torso_top - 6)
        hand_r = (x + 22, torso_top - 6)
        pygame.draw.line(surface, KEEPER_COLOR, (x, torso_top + 6), hand_l, 5)
        pygame.draw.line(surface, KEEPER_COLOR, (x, torso_top + 6), hand_r, 5)

        # Disegno specifico dei guanti dorati sulle mani
        pygame.draw.circle(surface, GLOVE_COLOR, (int(hand_l[0]), int(hand_l[1])), 4)
        pygame.draw.circle(surface, GLOVE_COLOR, (int(hand_r[0]), int(hand_r[1])), 4)
```

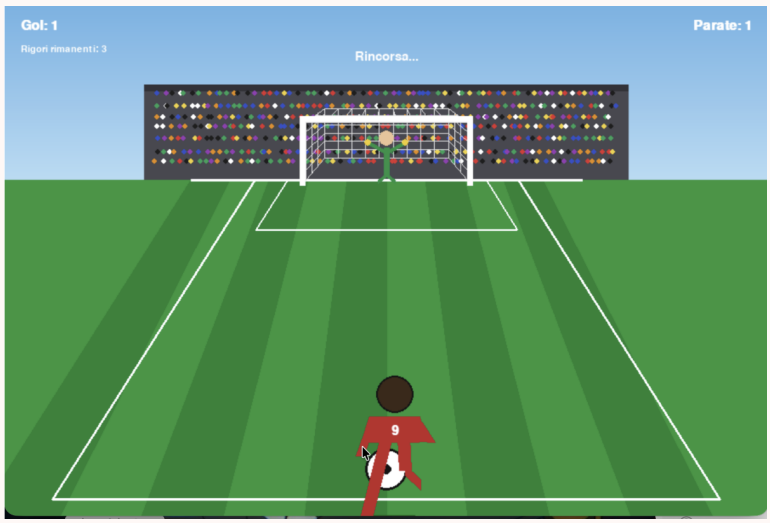
Listato 3.53: Claude - Renderizzazione del pallone statico e *design* del portiere

Claude



A seguito di questo aggiornamento estetico, l'interazione è proseguita con ulteriori richieste di correzione, focalizzate, in particolare, sul riempimento delle "zone nere" laterali dello schermo. Tuttavia, Claude non è riuscito a ottenere il risultato desiderato, evidenziando un limite nella gestione dello sfondo.

Claude



3.3 Valutazione dei Modelli

A conclusione dell'analisi dei singoli casi di studio, si propone un confronto diretto delle prestazioni fornite dai quattro modelli linguistici in esame. Le evidenze emerse durante le fasi di sviluppo sono state riportate nella Tabella 3.1. L'analisi comparativa si fonda su quattro parametri tecnici fondamentali:

- *Architettura del Codice*: distingue tra approcci modulari e monolitici.
- *Gestione dell'Archivio Domande*: evidenzia la separazione tra logica di gioco e dati testuali.
- *Volume del Codice*: parametro utile a far emergere la capacità di sintesi del modello.
- *Soddisfacimento delle Richieste*: descrive il raggiungimento degli obiettivi grafici e funzionali imposti dai *prompt*.

Modello LLM	Architettura del Codice	Gestione Archivio Domande	Volume del Codice	Soddisfacimento
ChatGPT (GPT-4o)	Modulare (più file)	Esterna (file `.json`)	Esteso	Totale
Gemini (3.1 Pro)	Monolitica (*file* unico)	Interna allo *script*	Sintetico	Totale
Copilot	Monolitica (*file* unico)	Interna allo *script*	Esteso	Totale
Claude (3.5 Sonnet)	Monolitica (*file* unico)	Interna allo *script*	Esteso	Parziale (limiti *layout* laterale)

Tabella 3.1: Confronto delle caratteristiche dei modelli LLM

CAPITOLO 4

Un *Case Study* per Confrontare Diversi LLM nella Generazione di Immagini

Questo capitolo presenta gli obiettivi del case study per la generazione di immagini, delineando i requisiti e le dinamiche del test condotto. L'analisi prosegue esaminando nel dettaglio il processo di elaborazione eseguito con le quattro architetture selezionate, riportando i principali prompt utilizzati e i relativi output visivi. La sezione finale propone un confronto comparativo mirato a far emergere le differenze strutturali e qualitative dei risultati ottenuti.

4.1 Definizione del *Case Study*

La valutazione della capacità di generazione di immagini dei modelli è stata condotta in continuità tematica con il *case study* precedente, incentrando nuovamente l'indagine sul tema calcistico attraverso la richiesta di molteplici elaborati visivi. L'analisi inizia con richieste semplici, per poi variare di *prompt* in *prompt*, introducendo scenari inediti o incrementando la difficoltà delle richieste stesse. Questo approccio funge, quindi, da *stress test* mirato a spingere i sistemi al proprio limite operativo per indurli all'errore. L'obiettivo dello studio è, pertanto, individuare gli attuali limiti dell'IA generativa in tale ambito, evidenziandone le ambiguità e misurando la precisione con cui le varie architetture riescono a interpretare e tradurre visivamente le istruzioni fornite.

4.2 Analisi dei Modelli

Di seguito vengono presentati i risultati dei *test* condotti sui quattro modelli selezionati. Al fine di garantire un confronto equo, a tutti i sistemi è stato sottoposto il medesimo *prompt* iniziale, coerentemente con l'approccio già adottato nel *case study* riguardante la generazione di codice. Successivamente, le istruzioni sono state adattate iterativamente in funzione degli *output* visivi generati da ciascun sistema, diversificando l'interazione per approfondire i singoli comportamenti.

4.2.1 Generazione con Gemini

Generazione dello Scenario di Base

La fase di analisi si è aperta con una richiesta semplice, volta a osservare in che modo il modello gestisse la composizione di uno scenario ambientale caratterizzato da più soggetti in interazione tra loro. Di seguito viene riportato il *prompt* utilizzato:

Utente

Generami l'immagine di un ragazzo che gioca a calcio in un campetto insieme ai suoi amici.

Gemini



Il sistema ha elaborato l'istruzione iniziale restituendo un *output* in linea con le aspettative. La scena riprodotta rispetta i parametri descritti, confermando una corretta gestione spaziale e un'adeguata integrazione dei soggetti nell'ambiente.

Variazione dell'Ambientazione

La fase successiva dell'analisi ha introdotto una variazione del contesto spaziale. La scena si sposta da un comune campetto a un grande stadio affollato per la finale di un torneo, richiedendo, inoltre, la generazione e l'organizzazione di una vasta folla sullo sfondo. La complessità è stata ulteriormente incrementata imponendo un'equa divisione sugli spalti dei tifosi delle due squadre. Di seguito viene riportato il *prompt* per la modifica richiesta:

Utente

Fai che la partita non sia una semplice partita al campetto, ma sia una finale di un torneo in uno stadio pieno pieno di tifosi delle 2 squadre (fai ad esempio squadra blu e rossa generiche, dividendo i colori dello stadio tra le 2 tifoserie).

Gemini



L'analisi dell'immagine prodotta rivela molteplici criticità. Nello specifico, emergono i seguenti problemi:

- *Allucinazioni testuali*: alcune delle scritte presenti sugli striscioni esposti dalle tifoserie appaiono deformate e illeggibili, evidenziando le difficoltà nella generazione di testo coerente. In Figura 4.1 viene riportato uno *zoom* sull'immagine, volto a illustrare un esempio del fenomeno di degradazione testuale.



Figura 4.1: *Zoom* per evidenziare il fenomeno della degradazione testuale

- *Errori di proporzione*: le dimensioni dei giocatori in campo risultano visibilmente sbilanciate rispetto alla reale ampiezza di quest'ultimo, danneggiando il realismo della scena.
- *Assenza di elementi chiave*: mancano totalmente le porte e le bandierine, componenti fondamentali per la coerenza logica dell'ambientazione richiesta.

- *Incongruenze strutturali*: le linee tracciate sul terreno di gioco formano una croce irregolare, generando una configurazione estranea alla reale geometria di un campo da calcio.

Per risolvere i diversi problemi, oltre alla correzione delle proporzioni del campo, è stato chiesto al modello di aggiungere gli elementi chiave mancanti:

Utente

Cerca di aggiustare le proporzioni del campo. Inoltre aggiungi le porte e le bandierine.

Gemini



L'*output* mostra un netto miglioramento nella coerenza dell'ambientazione. Il sistema ha integrato con successo gli elementi richiesti, correggendo, inoltre, la geometria delle linee per ricostruire la corretta struttura del campo. Sebbene la scala dei giocatori sia stata ridimensionata, le proporzioni delle figure continuano, tuttavia, a risultare sbilanciate rispetto alle dimensioni effettive del terreno di gioco.

Creazione di una Prima Pagina Sportiva

Per testare ulteriori campi di operatività del modello, è stato imposto un cambio di formato. L'obiettivo era valutare se il sistema fosse in grado di evolvere la precedente rappresentazione scenica in un prodotto editoriale, simulando la prima pagina di un quotidiano sportivo. Di seguito viene riportato il *prompt* utilizzato:

Utente

Fai in modo che la stessa immagine diventi la prima pagina di un giornale sportivo che descrive la vittoria di una delle 2 squadre.

Gemini



Il modello ha assecondato la richiesta, riuscendo a riportare l'immagine precedentemente elaborata all'interno di un *layout* giornalistico. Il sistema ha, inoltre, integrato in modo coerente gli elementi aggiuntivi pertinenti alla narrazione sportiva, arricchendo la composizione con riferimenti testuali sull'esito della partita e con il trofeo come nuovo elemento grafico.

Cambio di Prospettiva e nuovo Scenario

Per incrementare ulteriormente il livello di complessità, è stato chiesto al modello di operare una ricontestualizzazione della scena. L'obiettivo era verificare se il sistema fosse in grado di preservare la coerenza visiva della prima pagina appena generata, trasformandola in un semplice oggetto di scena. Di seguito viene riportato il *prompt*:

Utente

Genera l'immagine di un uomo che legge questo giornale.

Gemini



L'esito di questa generazione ha fatto emergere una criticità legata al cambio di prospettiva. Sebbene il modello sia riuscito a inquadrare l'oggetto nel nuovo contesto, si osserva una perdita di risoluzione: le scritte precedentemente strutturate sulla prima pagina del giornale si sono infatti degradate, trasformandosi in tratti confusi e illeggibili a causa della riduzione in scala dell'elemento. Tale distorsione risulta massima nella scritta in verticale. In Figura 4.2 si riporta un frammento ingrandito dell'immagine precedentemente generata dal modello.



Figura 4.2: *Zoom* per evidenziare il fenomeno della degradazione testuale

4.2.2 Generazione con ChatGPT

Generazione dello Scenario di Base

Il *test* di generazione visiva condotto con ChatGPT è iniziato riprendendo il *prompt* di partenza impiegato per il modello precedente. Come per l'analisi del codice, l'impiego dello stesso *input* assicura una base di partenza neutrale per valutare le differenze interpretative tra i due sistemi:

Utente

Generami l'immagine di un ragazzo che gioca a calcio in un campetto insieme ai suoi amici.

ChatGPT



Anche in questo caso, il modello ha elaborato con successo l'istruzione iniziale. Come per il sistema precedente, la scena generata rispetta fedelmente le richieste del *prompt*, restituendo un'immagine coerente con le aspettative e confermando un'accurata comprensione del contesto descritto.

inquadratura Ravvicinata sugli Spalti

Riproponendo a ChatGPT il *prompt* relativo al cambio di ambientazione all'interno dello stadio, il modello ha restituito un'immagine coerente con le richieste. Tuttavia, avendo generato un numero limitato di scritte, non sono emersi difetti testuali evidenti. Per forzare il sistema su questo specifico aspetto e metterne alla prova le reali capacità, è stata, quindi, richiesta un'ulteriore inquadratura ravvicinata sugli spalti, aumentando il numero di striscioni:

Utente

Genera un'immagine con vista sui tifosi allo stadio mostrando quindi i vari striscioni di incitamento per le 2 squadre.

ChatGPT



Dall'analisi dell'output prodotto, come mostrato in Figura 4.3, risulta che la scena mantiene una struttura visiva generale piuttosto coerente. Tuttavia, alcune delle scritte presenti sugli striscioni iniziano a mostrare lievi degradazioni, confermando la difficoltà del modello nel generare testi nitidi se collocati all'interno di un'immagine particolarmente affollata.



Figura 4.3: *Zoom* per evidenziare il fenomeno della degradazione testuale

Inquadratura dal basso e Gestione degli Elementi di Sfondo

Per testare ulteriormente i limiti del modello, l'interazione è proseguita con un *prompt* ad elevata densità di informazioni. La richiesta è stata formulata con l'obiettivo di alzare il

livello di difficoltà dell'elaborazione, imponendo un cambio di prospettiva. Al sistema è stato, infatti, richiesto di gestire contemporaneamente molteplici giocatori in azione, la geometria del terreno di gioco e la presenza di folla e testi in secondo piano. Di seguito si riporta il *prompt*:

Utente

Genera una immagine con visuale dal basso di una partita di calcio tra 2 squadre (rossa e blu) con: un giocatore che sta effettuando un cross, mentre due avversari entrano nell'area di rigore. Tutte le linee di campo devono essere visibili (linee laterali, dell'area di rigore e metà campo). Dietro la porta c'è una grande curva dello stadio piena di tifosi, con diversi striscioni con scritte di incoraggiamento per le 2 squadre.

ChatGPT



L'*output* dimostra come l'elevata densità di informazioni abbia mandato in crisi la comprensione del modello. Il sistema ha, infatti, fallito nella resa delle proporzioni e nella struttura del terreno di gioco, posizionando le due porte su lati adiacenti (perpendicolari tra loro) anziché alle classiche estremità opposte (parallele) del campo. A questa anomalia si aggiunge l'errata rappresentazione dell'area di rigore, le cui linee deviano senza riuscire a completare il classico perimetro rettangolare. Nonostante tali criticità, il sistema ha gestito correttamente la componente testuale richiesta per la realizzazione dei vari striscioni della curva.

4.2.3 Generazione con Copilot

Generazione dello Scenario di Base

Coerentemente con l'approccio adottato per i modelli precedenti, anche per Copilot si propone il medesimo *prompt* iniziale, garantendo, così, un confronto equo tra i vari sistemi. Di seguito si riporta il *prompt*:

Utente

Generami l'immagine di un ragazzo che gioca a calcio in un campetto insieme ai suoi amici.

Copilot

Mi dispiace, non sono riuscito a generare l'immagine a causa di una limitazione del sistema. Puoi provare a riformulare la richiesta in modo più dettagliato, ad esempio:
"Un adolescente che gioca a calcio su un piccolo campo all'aperto con alcuni amici, giornata di sole, atmosfera sportiva e allegra, stile fotografico realistico."
Oppure:
"Gruppo di ragazzi che giocano a calcio in un campetto di quartiere al tramonto, con erba verde e cielo colorato, immagine realistica." Con una descrizione leggermente diversa posso riprovare.

A differenza dei modelli precedenti, la prima interazione con Copilot ha restituito un esito negativo. Il sistema ha, infatti, dichiarato l'impossibilità di elaborare la richiesta a causa di una generica "limitazione del sistema", proponendo delle alternative per aggirare il problema. Si è deciso, pertanto, di accogliere il suggerimento testando il modello con il seguente *prompt*:

Utente

Un adolescente che gioca a calcio su un piccolo campo all'aperto con alcuni amici, giornata di sole, atmosfera sportiva e allegra, stile fotografico realistico.

Copilot

Mi dispiace, non sono riuscito a generare l'immagine a causa di una limitazione del sistema. Puoi provare con una descrizione completamente diversa e originale, cambiando tutti gli elementi principali (personaggi, abbigliamento, attività e ambientazione). Ad esempio: "Un gruppo di giovani che si divertono in un parco durante una giornata di sole, stile fotografico realistico, atmosfera allegra e dinamica." Con una nuova descrizione posso riprovare.

Sorprendentemente, anche l'elaborazione del *prompt* suggerito dal modello stesso è stata interrotta a causa di una "limitazione del sistema", con l'invito, questa volta, a modificare completamente la richiesta abbandonando il contesto calcistico. Al fine di sbloccare la generazione e ottenere un *output* valutabile, si è deciso, pertanto, di assecondare nuovamente il modello, fornendo come *input* la sua seconda proposta:

Utente

Un gruppo di giovani che si divertono in un parco durante una giornata di sole, stile fotografico realistico, atmosfera allegra e dinamica

Copilot



Tramite la seconda variante, la generazione è finalmente andata a buon fine. Il modello ha elaborato la richiesta superando le limitazioni precedenti e ha restituito un'immagine coerente con la descrizione, sebbene il tema risultasse ormai differente dal contesto calcistico iniziale dello studio. Per forzare il sistema a tornare sull'obiettivo originario dell'analisi, è stato richiesto un intervento correttivo mirato tramite il seguente *prompt*:

Utente

Vorrei modificare l'immagine e far sì che i ragazzi giochino a calcio tra di loro.

Copilot



L'*output* prodotto dimostra come il sistema, a seguito di molteplici richieste, sia stato in grado di superare le varie limitazioni. L'immagine è stata, infatti, generata correttamente, fornendo un risultato finalmente accettabile che consente di procedere con i successivi *test* previsti dallo studio.

Variazione dell'Ambientazione

Stabilizzato il sistema sul tema calcistico, il *test* è passato alla fase successiva. Al modello è stato, pertanto, richiesto di generare uno stadio colmo di spettatori, imponendo una rigida divisione visiva del pubblico in base ai colori delle due formazioni in campo:

Utente

Fai che la partita non sia una semplice partita al campetto, ma sia una finale di un torneo in uno stadio pieno pieno di tifosi delle 2 squadre (fai ad esempio squadra blu e rossa generiche, dividendo i colori dello stadio tra le 2 tifoserie).

Copilot

Mi dispiace, non sono riuscito a generare questa variante dell'immagine a causa di una limitazione del sistema. Puoi provare con una descrizione completamente nuova e originale cambiando tutti gli elementi principali (personaggi, colori, ambientazione, abbigliamento e azione), ad esempio una scena sportiva generica in un grande stadio senza riferimenti specifici alle immagini precedenti.

L'esito di questa richiesta ha fatto emergere un ulteriore problema del sistema. Il modello ha, infatti, nuovamente bloccato la generazione a causa di una generica "limitazione". Tuttavia, la particolarità principale risiede nella sua proposta di risoluzione: il sistema ha esplicitamente richiesto di interrompere la continuità con il lavoro svolto e di formulare un *prompt* completamente slegato dalle elaborazioni precedenti. Questo comportamento dimostra come l'architettura in questione fatichi a sostenere un processo continuo di modifiche, finendo per imporre un vero e proprio *reset* della sessione e costringendo l'utente a ripartire da zero.

Dinamica del Calcio di Punizione

L'indagine è proseguita con un nuovo prompt a tema calcistico, slegato dalle elaborazioni precedenti e progettato per mettere immediatamente alla prova il modello. Affrontando la dinamica di un calcio da fermo, al sistema è stato richiesto di gestire contemporaneamente le posture di tre elementi differenti:

- il battitore in preparazione;
- il blocco unito dei difensori (la barriera);
- il portiere intento a fornire indicazioni.

Di seguito si riporta il *prompt* fornito al modello:

Utente

Genera un'immagine che mostri un calciatore che deve battere una punizione da fuori area mostrando quindi la barriera della squadra difendente con i giocatori uniti ed il portiere che dà indicazioni ai suoi giocatori.

Copilot



Come mostrato nella Figura 4.4, l'*output* evidenzia un errore nella generazione delle mani dei giocatori posizionati in barriera. Queste ultime non presentano una struttura definita, ma si fondono tra loro creando una massa innaturale in cui si perde la distinzione tra le due mani e il numero delle dita. Tale difetto visivo compromette il realismo della composizione, dimostrando la difficoltà del modello nel riprodurre correttamente le estremità umane in situazioni di intreccio o sovrapposizione.

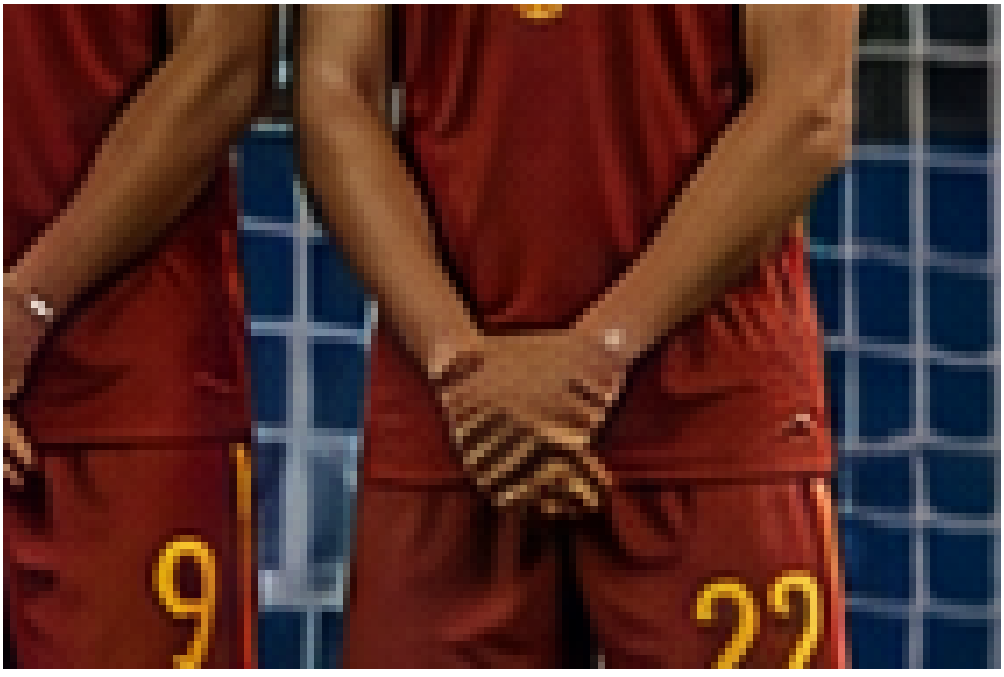


Figura 4.4: *Zoom* per evidenziare il fenomeno della degradazione delle mani

Generazione Testuale sugli Spalti

Per valutare coerentemente il modello con gli altri sistemi, l'analisi si è spostata sulle sue capacità di generazione testuale. Riproponendo il medesimo *prompt* impiegato per l'architettura precedente, è stata richiesta un'inquadratura sulle curve dello stadio, così da produrre un'alta densità di elementi testuali e misurarne l'effettiva leggibilità:

Utente

Genera un'immagine con vista sui tifosi allo stadio mostrando quindi i vari striscioni di incitamento per le 2 squadre.

Copilot



Il risultato evidenzia un progressivo deterioramento dei caratteri. Pur mantenendo una coerenza complessiva nel testo principale, si nota un leggero degrado nello striscione rosso posizionato nella porzione superiore degli spalti. Inoltre, come mostrato dal dettaglio ingrandito in Figura 4.5, è presente una corruzione tipografica che rende incomprensibile un elemento testuale collocato sullo sfondo:

Inquadratura dal Basso e Gestione degli Elementi di Sfondo

Con il *prompt* successivo si ripete il *test* ad alta densità di vincoli del modello precedente. L'obiettivo è forzare il sistema a gestire simultaneamente molteplici richieste, quali la prospettiva dell'inquadratura dal basso, la correttezza delle linee del terreno di gioco e la densità dei giocatori in movimento, oltre ai dettagli testuali sugli spalti:

Utente

Genera una immagine con visuale dal basso di una partita di calcio tra 2 squadre (rossa e blu) con: un giocatore che sta effettuando un cross, mentre due avversari entrano nell'area di rigore. Tutte le linee di campo devono essere visibili (linee laterali, dell'area di rigore e metà campo). Dietro la porta c'è una grande curva dello stadio piena di tifosi, con diversi striscioni con scritte di incoraggiamento per le 2 squadre.



Figura 4.5: Zoom per evidenziare il fenomeno della degradazione testuale

Copilot



L'*output* dimostra come l'elevata densità di informazioni abbia mandato in crisi la coerenza logica del modello. Un'evidente anomalia riguarda la disposizione delle linee, con la lunetta del calcio d'angolo collocata erroneamente a metà della linea di fondo campo. A questo errore si aggiunge un'incongruenza temporale nello svolgimento dell'azione: nonostante la palla sia già in volo, il battitore mantiene una postura di tiro incompatibile con la traiettoria della sfera. Un ulteriore errore generativo è dato dal posizionamento delle squadre in campo, disposte in maniera casuale e irrealistica rispetto alla situazione richiesta; infatti, normalmente, in un calcio d'angolo, la squadra attaccante riempie l'area di rigore avversaria.

4.2.4 Generazione con Claude

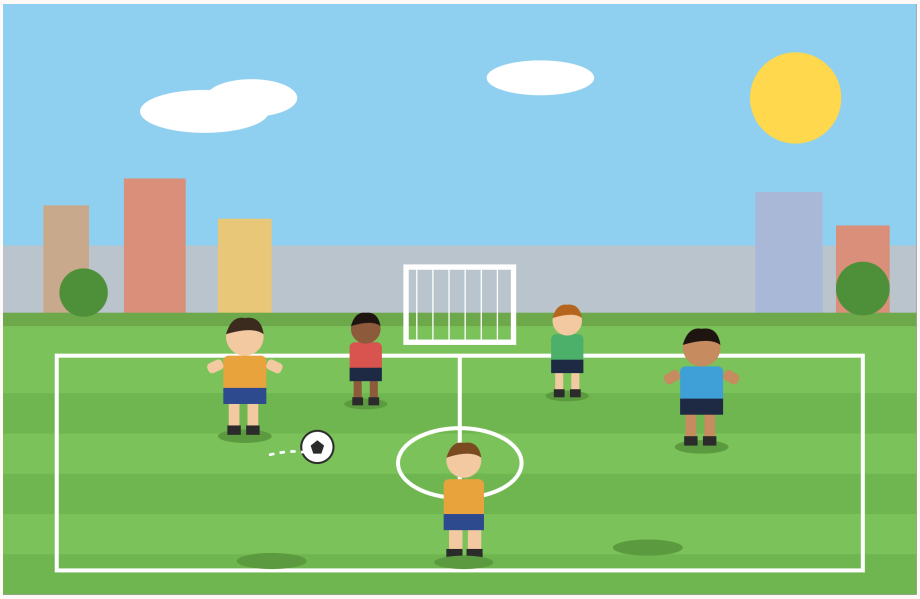
Generazione dello Scenario di Base

Passando a Claude, l'ultimo modello preso in esame, l'indagine riparte dal medesimo *prompt* iniziale, al fine di mantenere una rigorosa coerenza con le architetture precedentemente analizzate:

Utente

Generami l'immagine di un ragazzo che gioca a calcio in un campetto con i suoi amici.

Claude



A differenza dei modelli studiati in precedenza, Claude è privo di un motore dedicato alla generazione di *output* fotorealistici. Il sistema ha, quindi, processato il *prompt* elaborando direttamente del codice SVG, traducendo la richiesta testuale in una grafica puramente vettoriale. Come risultato si ottiene un'illustrazione bidimensionale minimale. La configurazione del campo è definita da zone verdi e linee, sulle quali si muovono figure stilizzate composte da forme geometriche elementari, che richiamano la sagoma umana (seppur in alcuni casi prive di braccia), accompagnate da un pallone in movimento. La scena viene, infine, completata da uno sfondo in cui l'orizzonte e gli elementi paesaggistici sono, a loro volta, sintetizzati attraverso l'uso esclusivo di geometrie di base.

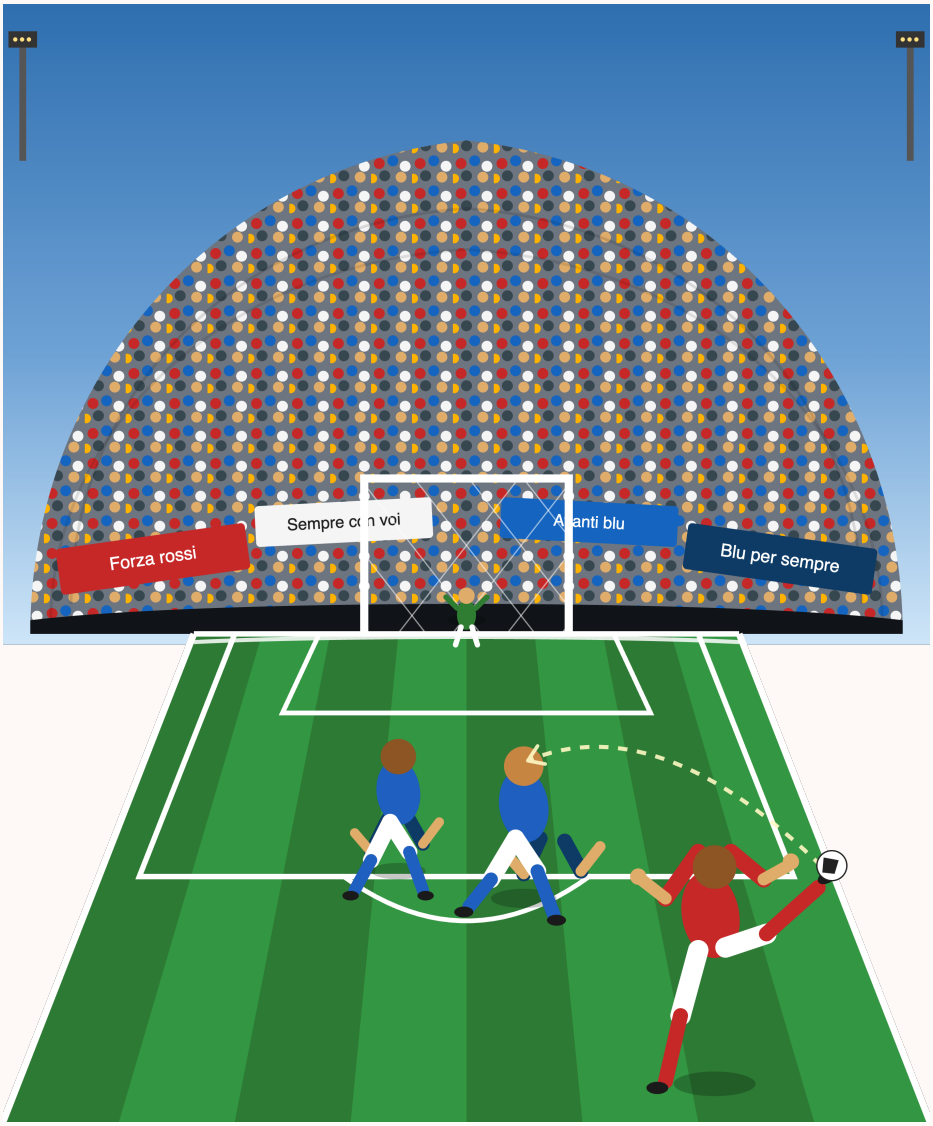
Inquadratura dal Basso e Gestione degli Elementi di Sfondo

Nonostante le differenze tra Claude e gli altri modelli, per garantire la completa coerenza dell'analisi, al sistema è stato imposto il medesimo *test* ad alta densità di vincoli utilizzato per le altre architetture. L'obiettivo è verificare in che misura il modello riesca a tradurre in coordinate matematiche e sintassi SVG una scena complessa. Di seguito viene riproposto il *prompt* relativo al cambio di prospettiva e alla generazione di molteplici elementi:

Utente

Genera una immagine con visuale dal basso di una partita di calcio tra 2 squadre (rossa e blu) con: un giocatore che sta effettuando un cross, mentre due avversari entrano nell'area di rigore. Tutte le linee di campo devono essere visibili (linee laterali, dell'area di rigore e metà campo). Dietro la porta c'è una grande curva dello stadio piena di tifosi, con diversi striscioni con scritte di incoraggiamento per le 2 squadre.

Claude



Oltre all'assenza di realismo, l'analisi dell'immagine rivela gravi anomalie nella scomposizione delle forme vettoriali. Si osserva come la struttura dei giocatori sia del tutto irregolare: le braccia dei due difensori, ad esempio, risultano staccate dalla sagoma principale del corpo. Sono stati, inoltre, generati elementi privi di logica fisica, come i due lampioni sospesi nel cielo in corrispondenza degli angoli superiori.

4.3 Valutazione dei Modelli

L'analisi comparativa ha rivelato come le architetture testate presentino diverse criticità, in particolare nella generazione testuale, nel rispetto delle proporzioni spaziali, nella coerenza dei dettagli anatomici e nel superamento di specifiche limitazioni intrinseche. Di seguito si riportano i limiti emersi per ciascun modello:

- *Gemini*: il modello, pur avendo elaborato le richieste generando i relativi *output* visivi senza apparenti ostacoli, ha mostrato evidenti criticità nel mantenimento delle corrette proporzioni in scenari ad alta densità compositiva, oltre a una degradazione testuale.
- *ChatGPT*: l'architettura ha evidenziato un lieve deterioramento dei caratteri testuali, mostrando, tuttavia, maggiori difficoltà nella gestione del *prompt* caratterizzato dal cambio della prospettiva e da un'elevata densità di elementi da rappresentare.
- *Copilot*: il modello, a differenza dei precedenti, è stato più volte limitato dai suoi vincoli intrinseci. Pur riuscendo a generare gli *output* desiderati assecondando i suggerimenti dell'architettura, i risultati ottenuti hanno, comunque, rivelato difetti nella generazione anatomica, una lieve corruzione testuale e una generale incapacità di processare coerentemente il *prompt* ad alta densità di richieste.
- *Claude*: il sistema ha mostrato le prestazioni più critiche a causa della sua incapacità nativa di generare contenuti fotorealistici. Nonostante questo limite strutturale, sottoponendo l'architettura al medesimo *test* ad alta densità di vincoli, si è osservato come il modello abbia prodotto sagome vettoriali irregolari e distribuito elementi secondari sullo sfondo senza alcuna coerenza logica.

CAPITOLO 5

Discussione

In questo capitolo viene presentata una discussione complessiva dello studio, condotto al fine di esplorare e analizzare le potenzialità di diversi LLM e di misurarne le reali capacità operative nella generazione di codice e di immagini. È stato possibile testare le varie architetture tramite la progettazione e la realizzazione di specifici case study. Vengono, pertanto, discussi in seguito i risultati dei due test, evidenziando i rispettivi punti di forza, i punti di debolezza e gli attuali limiti tecnologici dei modelli analizzati.

5.1 Discussione Relativa alla Generazione di Codice

La valutazione delle prestazioni dei quattro modelli linguistici nella generazione di codice, condotta nel terzo capitolo, permette di formulare importanti considerazioni sulle loro attuali capacità di sviluppo. Il dato più rilevante emerso dal confronto riguarda la profonda divergenza nell'approccio architetturale. ChatGPT si è distinto per aver proposto spontaneamente un'architettura a *file* multipli, isolando i dati testuali in un *file* esterno. Al contrario, Gemini, Copilot e Claude adottano un approccio monolitico che, sebbene sufficiente per prototipi rapidi, risulterebbe difficilmente gestibile in scenari di produzione reali. Tuttavia, ChatGPT è stato l'unico modello a non aver portato a termine con successo il *test*. Un secondo elemento di valutazione riguarda l'efficienza delle architetture. A parità di successo funzionale, Gemini ha dimostrato una maggiore capacità di ottimizzazione, raggiungendo la soddisfazione dei requisiti con un volume nettamente inferiore di righe di codice. Infine, l'analisi conferma l'affidabilità degli LLM nella gestione della logica di programmazione, ma evidenzia ancora delle lacune nella traduzione dei vincoli visivi e spaziali in codice. Il parziale fallimento di Claude nella gestione del *layout* laterale suggerisce, infatti, che l'elaborazione delle interfacce grafiche rappresenti ancora un ostacolo per alcuni LLM.

5.2 Discussione Relativa alla Generazione di Immagini

La valutazione sulla generazione di immagini mostra come, nonostante le architetture testate abbiano prodotto svariati *output*, persistano alcune limitazioni strutturali. La difficoltà principale evidenziata dai risultati riguarda la gestione della complessità spaziale e proporzionale. Infatti, all'aumentare della densità di elementi da generare o al variare della prospettiva, le prestazioni peggiorano nei vari modelli. Gemini ha riportato evidenti criticità

nel mantenimento delle proporzioni corrette, mentre ChatGPT e Copilot hanno mostrato un'incapacità generale di elaborazione coerente di scenari ad alta densità, faticando, in particolar modo, sui cambi di prospettiva. Un secondo limite condiviso, che rappresenta un problema ricorrente per i vari modelli di sintesi visiva esaminati, riguarda la degradazione testuale e anatomica. Seppur diversamente, infatti, Gemini, ChatGPT e Copilot hanno tutti evidenziato corruzioni o deterioramenti nella generazione di caratteri testuali. Copilot ha, inoltre, mostrato difetti specifici nella resa anatomica, risultando spesso limitato dai propri vincoli intrinseci. Infine, il caso di Claude ha evidenziato quanto il motore nativo di un sistema ne condizioni l'applicabilità. La sua incapacità strutturale di creare contenuti fotorealistici ha portato, infatti, agli *output* più critici dei *test*, producendo come risultato finale sagome vettoriali irregolari e un'organizzazione degli elementi secondari priva di coerenza logica.

Conclusioni

Nella presente tesi abbiamo esplorato il tema dell'Intelligenza Artificiale Generativa, sia da un punto di vista teorico, che pratico, attraverso lo sviluppo di specifici *case study*. Inizialmente, abbiamo fornito la definizione di Intelligenza Artificiale e ripercorso l'evoluzione storica che ha condotto allo sviluppo dei modelli generativi. Successivamente, abbiamo definito i concetti alla base dei Large Language Model (LLM) per poi analizzare, prima di presentare le architetture in esame, la distinzione fondamentale tra modelli *open source* e proprietari. Infine, abbiamo condotto uno studio pratico sulla generazione di diverse tipologie di *output*, confrontando in modo strutturato i risultati ottenuti dai quattro sistemi selezionati (Gemini, ChatGPT, Copilot e Claude). L'obiettivo principale è stato quello di osservare il comportamento di tali architetture di fronte ai medesimi *case study*, evidenziando le differenze tra i modelli.

Il primo *test* mirava a valutare come i diversi LLM avrebbero risposto alla richiesta di realizzare un *quiz* interattivo tramite uno *script* in Python. Siamo partiti da un'istruzione volta a definire la logica di base, per poi procedere attraverso continue iterazioni con lo scopo di verificare fino a che punto i sistemi riuscissero a soddisfare richieste sempre più complesse.

Il secondo *test* si è, invece, concentrato sulla generazione di immagini. Anche in questo caso siamo partiti da un *prompt* iniziale con pochi dettagli, per poi iterare ulteriori comandi tramite richieste specifiche ed effettuare dei veri e propri *stress test* (come cambi di prospettiva o l'aggiunta di nuovi elementi), confrontando, infine, gli *output* visivi ottenuti dai diversi modelli.

I vari risultati mostrano chiaramente come i modelli di Intelligenza Artificiale Generativa abbiano ormai raggiunto un elevato livello di affidabilità. Se impiegati adeguatamente, si rivelano strumenti di straordinaria utilità, capaci di automatizzare diverse attività e di garantire una disponibilità operativa costante. Tuttavia, dall'analisi emerge un principio fondamentale, ovvero che l'IA non è destinata a sostituire l'uomo, ma ad affiancarlo. È unicamente attraverso questa stretta collaborazione, in cui la tecnologia supporta l'utente, che si può estrarre il vero potenziale di questi modelli.

Guardando agli sviluppi futuri, possiamo prevedere che l'Intelligenza Artificiale Generativa sarà sempre più integrata nelle attività professionali di ogni settore. Il ruolo dell'essere umano si evolverà di conseguenza e, lavorando a stretto contatto con l'IA, l'utente assumerà la funzione di supervisore, una figura fondamentale per validare gli *output* al fine di garantire la qualità del risultato finale.

Bibliografia

ANTHROPIC (2024), «The Claude 3 Model Family: Opus, Sonnet, Haiku», *Anthropic*.

GEMINI TEAM GOOGLE, ANIL, R., BORGEAUD, S. e OTHERS (2023), «Gemini: A Family of Highly Capable Multimodal Models», *arXiv preprint arXiv:2312.11805*.

GOODFELLOW, I., POUGET-ABADIE, J., MIRZA, M. e OTHERS (2014), «Generative Adversarial Nets», *Advances in Neural Information Processing Systems*.

MCKINSEY & COMPANY (2024), «The state of AI in early 2024: Gen AI adoption spikes and traces of value appear», *McKinsey & Company*.

MIKOLOV, T., CHEN, K., CORRADO, G. e DEAN, J. (2013), «Efficient Estimation of Word Representations in Vector Space», *arXiv preprint arXiv:1301.3781*.

RAHMAN, M. I. (2025), *Study of Generative Artificial Intelligence with Machine Learning and Deep Learning*, AGPH Books.

URSINO, D. (2026a), «Parte XII: Generative AI e Ingegneria del Software», Materiale didattico del corso di Ingegneria del Software, Università Politecnica delle Marche, anno Accademico 2025/2026.

URSINO, D. (2026b), «Parte XIII: Epilogo», Materiale didattico del corso di Ingegneria del Software, Università Politecnica delle Marche, anno Accademico 2025/2026.

Sitografia

- Parlamento europeo, Che cos'è l'Intelligenza Artificiale? – https://www.europarl.europa.eu/topics/it/article/20200827STO85804/che-cos-e-l-intelligenza-artificiale-e-come-viene-usata
- IBM, Cos'è l'intelligenza artificiale (AI)? – https://www.ibm.com/it-it/think/topics/artificial-intelligence
- Databricks, Intelligenza Artificiale Generativa – https://www.databricks.com/it/discover/generative-ai
- Greenpeace, Il volto nascosto dell'IA: vorace di risorse e fonte di inquinamento – https://www.greenpeace.org/italy/storia/27245/il-volto-nascosto-della-vorace-di-risorse-e-fonte-di-inquinamento/
- IBM, Cos'è il Deep Learning? – https://www.ibm.com/it-it/think/topics/deep-learning
- Ultralytics, Cosa sono i modelli di diffusione? – https://www.ultralytics.com/it/blog/what-are-diffusion-models-a-quick-and-comprehensive-guide
- Bucap, Cos'è il cluster: una soluzione per garantire la continuità operativa – https://www.bucap.it/news/approfondimenti-tematici/continuta-operativa/cluster-cos-e.htm
- ICT Security Magazine, Generative AI: Evoluzione, Funzionamento e Impatto sulla Trasformazione Digitale – https://www.ictsecuritymagazine.com/articoli/generative-ai/
- Parseur, Come Automatizzare le Attività Ripetitive - Prima il Compito, Poi lo Strumento – https://parseur.com/it/blog/automatizzare-attivita-ripetitive-computer
- IBM, Cos'è la generazione di codice AI? – https://www.ibm.com/it-it/think/topics/ai-code-generation
- Intempra, AI Generativa nel B2B: previsioni e vantaggi secondo McKinsey e Gartner – https://intempra.com/it/AI-Generativa-nel-B2B-previsioni-e-vantaggi-secondo-McKinsey-e-Gartner_blog531.html

- Parlamento europeo, Quali sono i rischi e i vantaggi dell'intelligenza artificiale? – https://www.europarl.europa.eu/topics/it/article/20200918STO87404/quali-sono-i-rischi-e-i-vantaggi-dell-intelligenza-artificiale
- Voxeurop, I rischi dell'intelligenza artificiale per l'umanità – https://voxeurop.eu/it/rischi-intelligenza-artificiale-vissol/
- Neurally, RAG e Fine-tuning a confronto: due approcci per la personalizzazione dei modelli di IA generativa – https://www.neurally.it/rag-e-fine-tuning-a-confronto-due-approcci-per-la-personalizzazione-dei-modelli-di-ia-generativa
- IBM, Cosa sono i parametri degli LLM? – https://www.ibm.com/it-it/think/topics/llm-parameters#692473877
- Databricks, Cosa sono i Large Language Models (LLM)? – https://www.databricks.com/it/blog/what-are-large-language-models
- Prompt Engineering Guide, Impostazioni LLM – https://www.promptingguide.ai/it/introduction/settings
- IBM, Che cos'è il prompt engineering? – https://www.ibm.com/it-it/think/topics/prompt-engineering
- Ultralytics, Batch Size – https://www.ultralytics.com/it/glossary/batch-size
- IBM, Cosa sono i benchmark LLM? – https://www.ibm.com/it-it/think/topics/llm-benchmarks
- Ultralytics, Context Window – https://www.ultralytics.com/it/glossary/context-window
- Red Hat, Cosa si intende per LLMOps? – https://www.redhat.com/it/topics/ai/llmops
- Everpure, Che cos'è l'architettura gestita in modo olistico? – https://www.everpuredata.com/it/knowledge/what-is-holistically-managed-architecture.html
- Gemini, Una panoramica dell'app Gemini – https://gemini.google/it/overview/?hl=it#introduction
- Gemini API, Eseguire il codice – https://ai.google.dev/gemini-api/docs/code-execution?hl=it
- engineering.com, What are input and output tokens in AI? – https://www.engineering.com/what-are-input-and-output-tokens-in-ai/
- Guida di App Gemini, Genera e modifica immagini con le app Gemini – https://support.google.com/gemini/answer/14286560?hl=it&co=GENIE.Platform%3DDesktop
- Gemini, Limitazioni note delle interfacce basate su LLM come Gemini – https://gemini.google/it/overview/?hl=it#limitations

- 365 Data Science, The History of OpenAI – https://365datascience.com/trending/the-evolution-of-chatgpt-history-and-future/?utm_source=gemini#3
- OpenAI, Note di rilascio dei modelli – https://help.openai.com/it-it/articles/9624314-model-release-notes
- OpenAI Developers, Code Interpreter – https://developers.openai.com/api/docs/guides/tools-code-interpreter
- OpenAI Developers, Image generation – https://developers.openai.com/api/docs/guides/tools-image-generation
- Aware, The Evolution of Copilot: From Code to Comprehensive AI Assistant – https://www.aware.co.th/the-evolution-of-copilot-from-code-to-comprehensive-ai-assistant/?utm_source=gemini
- Lenovo, Che cos'è uno snippet? – https://www.lenovo.com/it/it/glossary/snippet/
- Microsoft Learn, Usare l'interprete di codice in un prompt per generare ed eseguire codice Python – https://learn.microsoft.com/it-it/microsoft-copilot-studio/code-interpreter-for-prompts
- Microsoft, Utilizzo della generazione di immagini in Microsoft Copilot – https://support.microsoft.com/it-it/microsoft-copilot/using-image-generation-in-microsoft-copilot
- Microsoft Learn, Limitations – https://learn.microsoft.com/en-us/azure/foundry/responsible-ai/openai/transparency-note?tabs=text#limitations
- Claude Platform Docs, Introduzione a Claude – https://platform.claude.com/docs/it/intro
- Anthropic, Introducing Claude – https://www.anthropic.com/news/introducing-claude?utm_source=gemini
- Anthropic, Claude 2 – https://www.anthropic.com/news/claude-2?utm_source=gemini
- Anthropic, Introducing the next generation of Claude – https://www.anthropic.com/news/claude-3-family?utm_source=gemini
- Claude Platform Docs, Strumento di esecuzione del codice – https://platform.claude.com/docs/it/agents-and-tools/tool-use/code-execution-tool
- Claude Support, Claude può produrre immagini? – https://support.claude.com/it/articles/9002504-claude-puo-produrre-immagini

Ringraziamenti

Il primo e più grande ringraziamento va ai miei genitori, che mi hanno sempre sostenuto e spronato lungo tutto questo percorso, e ai miei fratelli, ai quali auguro il meglio per il loro futuro e le loro carriere.

Un pensiero affettuoso va anche ai miei parenti che, nonostante la distanza, mi hanno fatto sentire la loro vicinanza con le chiamate e il costante interesse.

Desidero, inoltre, ringraziare il Professore Domenico Ursino per avermi accompagnato lungo quest'ultima fase con estrema disponibilità e professionalità, fornendomi sempre preziosi chiarimenti e puntuali correzioni.

Un ultimo ringraziamento va a Davide, Marco, Sara e tutto l'Insieme Quo, amici insostituibili prima ancora che compagni di corso, la cui presenza è stata fondamentale per alleviare l'ansia e le paranoie prima degli esami e senza i quali non avrei costruito i ricordi preziosi di questi tre anni.

Se oggi ho raggiunto questo traguardo, è soprattutto grazie al vostro supporto. Per questo motivo, considero questa laurea non una vittoria personale, ma un successo di tutti noi.