POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE
Corso di Laurea Magistrale in Ingegneria Informatica

FORMAL LANGUAGES AND COMPILERS
Prof. Floriano Scioscia

Verso Linguaggi di Programmazione "LLM-Friendly": Un'Analisi del Costo Computazionale per la Generazione di Codice

Autore
Giovanni Felle

Anno Accademico 2024/2025

Indice

Capitolo 1

Introduzione

1.1 Contesto

L'evoluzione dei Large Language Models (LLM) ha portato a una crescente adozione della generazione automatica di codice e all'introduzione dell'Intelligenza Artificiale (IA) come strumento fondamentale nella fase di sviluppo e debugging dello stesso [18]. Diventa, dunque, strategico comprendere quali linguaggi di programmazione (LP) siano più efficienti dal punto di vista computazionale per gli LLM. In particolare, in contesti in cui la generazione automatica di codice via LLM si rivela essere un componente fondamentale dello sviluppo, sarà cruciale individuare i linguaggi che, a parità di dati a disposizione per la fase di training, comportano un costo computazionale minore sia in fase di addestramento che di inferenza. È fondamentale distinguere tra: le risorse computazionali impiegate dall'LLM durante il processo di generazione del codice (tempo di inferenza, elaborazione dei token, energia utilizzata durante l'inferenza) e l'efficienza in termini di prestazioni, memoria o energia utilizzata dal codice prodotto dall'LLM all'esecuzione. Sebbene entrambi contribuiscano al costo complessivo del progetto, questo lavoro si concentrerà principalmente sul primo aspetto, con discussioni pertinenti sul secondo in quanto influisce sull'utilità complessiva e sulla spesa a lungo termine.

1.2 Obiettivi

Il presente lavoro si pone l'obiettivo di esaminare - in una prima parte - lo stato dell'arte, ovvero raccogliere e analizzare studi teorici o sperimentali che affrontano la relazione tra linguaggio di programmazione e costo computazionale per un LLM Code Generator. Si indagheranno le caratteristiche strutturali, sintattiche e semantiche che rendono un LP più adatto alla generazione automatica efficiente (LLM-friendly). Si esploreranno eventuali proposte o LP ottimizzati specialmente per ridurre i costi nella generazione, e quali siano i principi alla base di essi.

Infine, si prefigge di fornire un'analisi comparativa approfondita delle capacità dei Large Language Models nella generazione di codice, con un focus specifico sull'accuratezza e sull'efficienza in termini di numero di token generati. L'obiettivo ultimo è identificare le sinergie tra le caratteristiche dei diversi LLM e la natura dei linguaggi e degli algoritmi, per informare scelte più consapevoli nello sviluppo assistito dall'IA.

In particolare, gli obiettivi di questo elaborato sono:

- **Generazione Automatizzata del Corpus di Codice:** Implementare un sistema per l'inferenza automatica di otto LLM su una matrice di cinque algoritmi e dieci linguaggi di programmazione, producendo un corpus diversificato di codice sorgente.
- **Pre-elaborazione e Normalizzazione del Codice:** Sviluppare e applicare un processo di pulizia e standardizzazione per rimuovere elementi non essenziali (come commenti e formattazioni Markdown) dagli script generati, garantendo una base dati coerente per le successive fasi di analisi.
- **Quantificazione dell'Efficienza (Conteggio Token):** Misurare il numero esatto di token per ogni script di codice prodotto, utilizzando tokenizer specifici per modello forniti da Hugging Face, al fine di valutare la concisione e l'efficienza degli output degli LLM.
- **Valutazione della Qualità del Codice:** Implementare una metodologia di valutazione rigorosa, basata su una scala personalizzata da 0 a 5, per assegnare un punteggio a ogni script in base alla sua correttezza lessicale, sintattica, logica e funzionale.
- **Analisi Comparativa e Visualizzazione dei Dati:** Aggregare e analizzare i dati raccolti per identificare le performance relative dei diversi LLM in base al linguaggio, all'algoritmo e alla loro dimensione/specializzazione costruendo una dashboard interattiva per visualizzare le tendenze e le correlazioni emergenti.

L'elaborato sarà strutturato nel seguente modo: questo capitolo è il primo e introduce il lettore al contesto di ricerca e agli obiettivi specifici che guidano lo studio. Il Capitolo 2 affronta lo Stato dell'Arte, esaminando studi teorici e sperimentali preesistenti che hanno indagato la relazione tra linguaggi di programmazione e costo computazionale per i generatori di codice basati su LLM, esplorando le caratteristiche che rendono un linguaggio "LLM-friendly". Il Capitolo 3 si concentra sulla metodologia di ricerca, descrivendo in dettaglio l'ambiente hardware utilizzato, i criteri di selezione e le specifiche degli otto LLM, dei cinque algoritmi e dei dieci linguaggi di programmazione, la definizione del prompt, e la scala di valutazione utilizzata, oltre alla metodologia di conteggio dei token. Verrà anche presentata una descrizione dettagliata delle fasi del processo di lavoro. Il Capitolo 4 presenta

l'analisi dei risultati, illustrando i dati aggregati relativi alle performance (punteggio e token) dei modelli su diverse dimensioni (per LLM, per linguaggio, per algoritmo) e fornendo un'analisi dettagliata di specifici casi di studio attraverso grafici comparativi. Il Capitolo **??** conclude l'elaborato, riassumendo le scoperte principali, interpretando i risultati ottenuti, evidenziando le implicazioni pratiche dello studio e proponendo direzioni per futuri lavori di ricerca e sviluppo.

Capitolo 2

Stato dell'Arte

2.1 I costi degli LLM

Comprendere i costi degli LLM risulta essere il primo passo da intraprendere nello sviluppo della presente ricerca. Come riportato da Gad Benram in [6], i fattori che influenzano il costo dell'inferenza (il processo di generazione dell'output) sono diversi e sono cambiati nel corso del tempo. Richieste di calcolo su input di grandi dimensioni (vedi il concetto di contesto e prompt engineering), l'esigenza di una latenza più bassa nella generazione, o le richieste di un output complessi e lunghi portano generalmente a un uso di maggiori risorse computazionali o infrastrutture ottimizzate [6]. In alternativa, come nel caso di GPT-4 o concorrenti, il tutto si riduce a un costo in denaro per numero di token. Infine, gli LLM consumano una quantità considerevole di energia durante l'inferenza, specialmente per compiti computazionalmente onerosi come la generazione di codice, sollevando preoccupazioni ambientali [8].

Nonostante le differenze di costo assoluto siano destinate a scendere, come spiegato in [5], le differenze di costo relative, determiante dalla tokenizzazione, sono destinate a persistere, rendendo la scelta di un LP una scelta di tipo economico.

2.2 Valutazione di LLM Code Generator

In Xu et al. [17], viene fornita una valutazione sistematica dei vari LLM, presenti in letteratura, per la generazione di codice. Inoltre, gli autori presentano PolyCoder, un LLM con 2.7B parametri, basato sull'architettura di GPT-2 che è stato allenato con 249GB di codice scritto in 12 linguaggi diversi. Viene mostrata la differenza nell'impiego di tre diversi tipi di LLM: auto-regressivi da sinistra-verso-destra, ovvero che prevedono la probabilità del prossimo token da quelli precedenti, che risultano essere i migliori per task generativi; modelli di linguaggio mascherato, che mascherano token casuali e chiedono al modello di predirli, sono i migliori per

l'analisi del codice; e modelli encoder-decoder in cui l'encoder si occupa di anlizzare l'input e il decoder genera un output. Nonostante l'obiettivo del documento studiato sia diverso da quello proposto nel lavoro presente, è possibile estrapolare dei dati empirici e delle osservazione che permetto di trarre delle considerazione utili a supporto delle conclusioni della ricerca. Dai risultati riportati si evice che: Python è il linguaggio su cui i modelli ottengono perfrormance migliori, ciò è attribuito all'alta disponibilità di dati di training, sintassi pulita e consistente e al principale ottimizzazioni di alcuni modelli su questo LP. PolyCoder è in grado di superare Codex, in termini di perplexity, nel generare codice in C indicando che è un buon linguaggio in cui un addestramento può produrre prestazioni eccellenti anche senza modelli di enormi dimensioni. Per linguaggi per cui la sintassi risulta più complessa e meno standardizzata, o presentano strutture grammaticali ricche o altamente dinamiche, come Scala, Ruby, PHP o TypeScript si verifica un perplexity molto più alta.

2.3 Concisione del codice

Il cuore del funzionamento degli LLM è descritto da un concetto fondamentale: i token. I token sono unità discrete di testo (parole, parti di esse o simboli) in cui un LLM suddivide l'input o l'output prima di elaborarlo [15]. Quindi, risulta facile dedurre che meno token sono richiesti, meno lavoro dovrà fare un LLM. Inoltre, come dimostrato da Velaga in [16], la tokenizzazione del codice è meno efficiente rispetto a quella del linguaggio naturale. Data la presenza di molti simboli e testo che non fanno parte del linguaggio naturale, la tokenizzazione di codice presenta spesso un rapporto caratteri-token sfavorevole. Inoltre, i metodi di tokenizzazioni variano tra i diversi LLM, anche utilizzando lo stesso input. Quindi anche se un LP risulta concettualmente "semplice", la sua struttura sintattica può portare a un conteggio di token più elevato. Un LP "meno costoso" potrebbe essere visto come un linguaggio che nella visione del LLM abbia una "densità di token" minore.

Il passo successivo, utile all'analisi condotta nel presente elaborato, è quello di comprendere come questa informazione si traduce nella generazione di codice. L'obiettivo è quindi quello di trovare un modo oggettivo e misurabile per confrontare quanto sono "concisi" (usano meno parole a parità di contenuto informativo) i diversi linguaggi di programmazione. In altre parole, risulta fondamentale decretare quanto un LP possa essere "ridotto" senza perdere informazioni: più il linguaggio è conciso, meno è ridondante, più può essere compresso. L'obiettivo precedentemente citato è il fulcro dello studio condotto da Bergmans et al. [7], che si occupa di misurare la concisione del codice sorgente di diversi linguaggi di programmazione utilizzando la compressione attraverso LZMA2. I risultati permettono confronti più accurati delle metriche tradizionali come le linee di codice (LOC) che non tengono conto dei termini utilizzati per eseguire un'istruzione. Si evince dunque che, a

parità di informazione utile shell script o Python richiedono meno caratteri (e quindi token) rispetto ad altri linguaggi. Dato che, come precedentemente spiegato, per gli LLM il costo computazionale scala con il numero di token, risulta ragionevole supporre che linguaggi più concisi consentano una generazione meno onerosa in inferenza e in fase di addestramento.

In studi come [4] viene mostrato come la lunghezza dei token per lo pseudocodice, derivato da diversi LP, è significativamente più corto del codice sorgente, suggerendo come rappresentazioni più concise riducono il carico di token. Nello specifico, lo pseudocodice derivato da Python conta 111 token al contrario di C++ e Rust che presentano una media di 123 token per una logica equivalente. Nello studio viene evidenziato come il "bottleneck" per linguaggi come Python sia la risoluzione stessa dei problemi, mentre per linguaggi come C++ e Rust la "codifica del linugaggio" risulta essere la fase più critica, portando a una generazione più iterativa e maggiore conteggio di token per ottenere codice eseguibile.

2.4 Linguaggi LLM-friendly

Come riportato in Twist et al. [13], vi è un forte bias da parte degli LLM che mostrano una preferenza verso Python. Risulta, infatti, che Python è preferito nel 90-97% dei casi, anche quando non è il linguaggio più adatto. Questo bias porta a delle implicazioni che si potrebbero tradurre in "codice omogeneizzato" e indurre sviluppatori a fare scelte subottimali, in termini di affidabilità, manutenibilità e sicurezza del codice. Questo bias però non implica in maniera diretta che Python sia "meno costoso" per un LLM, ma semplicemente "più facile" poiché, data la mole di dati di addestramento presenti, l'LLM tende a preferirlo rispetto ad altri LP. Questo crea una sfumatura critica: la preferenza non equivale all'efficienza intrinseca.

La semplicità sintattica rispetto alla ricchezza semantica gioca un ruolo cruciale. LP con una "cultura della semplicità" e tonnellate di dati disponibili (come Python) sono più "LLM-friendly"[2]. Questo perché gli LLM sono fondamentalmente riconoscitori di schemi: schemi più semplici e più esempi rendono più facile per loro generare codice in modo accurato. Al contrario, LP con un grado di fedeltà più alto e più rigidi nella sintassi, con necessità di controllo a basso livello, possono aumentare lo "sforzo" di generazione e aumentare il numero di iterazioni-token richiesti per produrre codice corretto e utilizzabile.

Risulta fondamentale sottolineare infatti che: più dati di addestramento disponibili, portano alla generazione di codice direttamente eseguibile e senza errori. Un linguaggio che richieda un numero di token minore, ma un numero di iterazioni maggiore per permettere all'LLM di produrre codice eseguibile non è migliore di un LP che richiede più token, ma meno iterazioni. Abbassi et al. [3] dimostrano che il codice generato dagli LLM può soffrire di "ridondanza" e "passi non necessari",

suggerendo che anche se un linguaggio è conciso, l'LLM potrebbe generare codice verboso, aumentando ulteriormente il conteggio dei token [11].

Inoltre, vi è da considerare che, sebbene LP più semplici (come Python) e con più dati a disposizione possano risultare "più economici" per generare codice sintatticamente corretto, in casi più specifici e con esigenze mirate altri linguaggi con sistemi di tipi più robusti (come Scala o Rust) potrebbero essere "più economici" in termini di costo complessivo del progetto, riducendo il debug post-generazione [2]. Come mostrato da Solovyeva et al. [12] gli LLM hanno più successo nella generazione di codice Python e Java, rispetto a linguaggi come C++ e Rust, anche se, come facilmente deducibile, lo studio non garantisce la parità di dati di addestramento. Tuttavia, la generazione corretta alla prima iterazione viene bilanciata con differenze intrinseche di prestazione runtime del codice generato nei diversi LP [14].

2.5 Linguaggio Ideale

Dalle sezioni precedenti è emerso che le caratteristiche chiave per l'efficienza di generazione degli LLM sono molteplici. In questa sezione si cercherà di riassumere i concetti fondamentali ricercando delle possibili soluzioni nello stato dell'arte.

Un approccio promettente, presentato da Li et al. [10], prevede l'utilizzo di compilatori per la tokenizzazione preliminare e la sostituzione dei token con unità lessicali, portando la diminuzione del conteggio dei token di input in modo significativo (fino al 33,7%), abbassando il costo computazionale. Questo approccio risulterebbe fondamentale per rendere i LP "meno costosi" per gli LLM, consentendogli di operare in modo più efficiente e riducendo il consumo di token per la generazione. Questo studio, dunque, spinge il concetto di "LLM-friendly" oltre il semplice sviluppo di un linguaggio con sintassi semplice e "riproducibile" con meno token possibili, ma si cimenta nello sviluppo di proprietà strutturali e semantiche più profonde.

LP fortemente tipizzati contribuiscono all'accuratezza e alla riduzione di allucinazioni da parte degli LLM [2]. Sebbene l'utilizzo di linguaggi che prevedono la tipizzazione forte possano aumentare il carico di "codifica del linguaggio" la loro capacità di prevenire errori e garantire la correttezza può portare a significativi risparmi sui costi a lungo termine, riducendo la fase di debug.

L'enorme mole di dati di addestramento, come anticipato, risulta essere un fattore da non sottovalutare. Ideare un nuovo LP orientato alla tokenizzazione limitata, che però non presenta dati di addestramento sufficienti per competere con altri linguaggi in ottica di eseguibilità del codice prodotto, potrebbe risultare controproducente, poiché la riduzione del costo computazionale rischierebbe di essere vanificata da un maggior numero di iterazione utili per rendere un codice utilizzabile.

In letteratura è possibile notare l'introduzione di nuovi LP come MoonBitLang [1] e Mojo [9] che riflettono una crescente consapevolezza della necessità di ottimizzare (o sviluppare) linguaggi per l'IA. MoonBitLang è progettato con l'obiettivo di rendere gli LLM più efficienti nella generazione di codice, puntando a una velocità maggiore di inferenza. Questo tipo di approccio, che si potrebbe definire "IA-nativo", è identificato da una progettazione del linguaggio in cui le caratteristiche intrinseche dello stesso sono pensate per facilitare l'elaborazione da parte degli LLM. Inoltre, il progetto presenta una toolchain integrata che rende il codice facilmente analizzabile dall'IA che risulta di supporto per la fase di refactoring e debug. Mojo, d'altra parte, si presenta come un forte candidato per essere un linguaggio ideale per l'output degli LLM, combinando l'usabilità e la leggibilità di Python con funzionalità di sistema di basso livello [9]. Inoltre, essendo un superset di Python, Mojo può beneficiare dell'ampia comunità di sviluppatori e librerie Python, inclusi pacchetti di interoperabilità in C e C++. Come sottolineato in precedenza, la possibilità di avere un vasto coprus di dati di addestramento e un ecosistema ricco sono fattori fondamentali che volgono a favore dell'utilizzo di Mojo in futuro prossimo. Lattner in [9] sottolinea come la progettazione di Mojo non si basi sull'idea di sfruttare LLM per eliminare i linguaggi di programmazione e il ruolo degli sviluppatori, ma fa si che l'IA utilizzata risulti essere un nuovo membro del team che contribuisce al progetto. Gli LP, come ribadito da Lattner, servono a tre dimensioni critiche: comunicazione umano-computer, comunicazione umano-umano e comunicazione computer-umano. È chiaro dunque che un LP è utile più per il programmatore che per la macchina e un approccio full-IA svilupperebbe un programma in modo diverso dalla scrittura di codice in un linguaggio [9].

Capitolo 3

Progettazione e Sviluppo

Questo capitolo illustra in dettaglio la metodologia adottata per condurre lo studio comparativo sulle performance dei Large Language Models (LLM) nella generazione di codice. Verranno descritti l'ambiente di esecuzione, la selezione dei modelli, degli algoritmi e dei linguaggi di programmazione, la definizione del prompt, e il sistema di valutazione delle risposte, inclusa la metodologia per il conteggio dei token.

3.1 Ambiente di Esecuzione e Infrastruttura Hardware

Per garantire un controllo completo sull'ambiente di inferenza e per gestire in modo efficiente i modelli scelti, è stata adottata una strategia di *hosting locale* degli LLM. A tal fine, è stato utilizzato **Ollama**, una piattaforma open-source che facilita l'esecuzione locale di grandi modelli linguistici su hardware commodity. Ollama permette di scaricare, eseguire e gestire un'ampia varietà di LLM direttamente sul proprio sistema, sfruttando le risorse hardware disponibili (principalmente la GPU), fornendo inoltre un'API per le chiamate di inferenza. Questa scelta ha permesso anche di eliminare le dipendenze da servizi cloud esterni e di avere un maggiore controllo sulla privacy e la riproducibilità degli esperimenti.

La macchina utilizzata per l'hosting locale degli LLM e l'esecuzione degli esperimenti presenta le seguenti specifiche hardware:

- **GPU:** NVIDIA GeForce RTX 2070 Super con 8 GB di VRAM (Video RAM).
- **CPU:** Intel Core i7-4770, 3.40 GHz.
- **RAM:** 24 GB di memoria RAM DDR3.

3.2 Selezione dei Large Language Models (LLM)

La selezione degli LLM è stata guidata dall'obiettivo di confrontare diverse architetture, dimensioni e specializzazioni - passando da modelli coder a modelli generici - nonché di analizzare l'impatto del *tuning* sul codice. Sono stati scelti un totale di otto modelli, derivanti da quattro famiglie principali, con due varianti di dimensione per ciascuna famiglia. Questa configurazione permette un'analisi intra-famiglia e inter-famiglia.

I modelli inclusi nello studio sono i seguenti:

- `deepseek-coder:1.3b`
- `deepseek-coder:6.7b`
- `qwen3:1.7b`
- `qwen3:8b`
- `qwen2.5-coder:1.5b`
- `qwen2.5-coder:7b`
- `gemma:2b`
- `gemma:7b`

Breve introduzione ai modelli selezionati:

- **DeepSeek Coder:** Sviluppati da DeepSeek, questi modelli sono specificamente addestrati per compiti di programmazione. La loro architettura e i dati di training sono ottimizzati per la generazione e comprensione di codice. La coppia `1.3b` e `6.7b` permette di valutare l'impatto della scalatura sui modelli *coder-specifici*.
- **Qwen (generici e Coder):** Sviluppati da Alibaba Cloud, i modelli Qwen sono noti per le loro capacità multimodali e multilingue. La famiglia `qwen3` rappresenta una versione più generica, mentre `qwen2.5-coder` indica una versione specificamente ottimizzata per il coding. Questa scelta è strategica per confrontare direttamente l'efficacia di un modello che ha subito un *fine-tuning* specifico per il codice (`qwen2.5-coder:1.5b` e `7b`) rispetto a una versione generica (`qwen3:1.7b` e `8b`) della stessa serie. Ciò permette di isolare l'impatto del tuning sulla performance di coding.
- **Gemma:** Rilasciati da Google, i modelli Gemma sono una famiglia di LLM aperti e leggeri, costruiti sulla stessa ricerca e tecnologia utilizzata per creare Gemini. Le varianti `2b` e `7b` sono state incluse per rappresentare un'alternativa di modelli generici con diverse scale parametriche, fornendo un confronto con le altre famiglie di LLM.

La selezione delle dimensioni dei modelli è stata pensata per includere sia candidati più leggeri (nel range 1.3-2 miliardi di parametri) che versioni di dimensioni intermedie/maggiori (nel range 6.7-8 miliardi di parametri). Questo approccio mira a comprendere come la dimensione del modello influenzi direttamente la sua capacità di generare codice di qualità e la sua efficienza (misurata in token).

3.3 Selezione degli Algoritmi

Per valutare la versatilità degli LLM, sono stati selezionati cinque algoritmi distinti, che spaziano dalla semplicità concettuale alla complessità implementativa, con l'obiettivo di coprire diversi paradigmi di programmazione.

Gli algoritmi selezionati sono i seguenti:

- `bubble_sort` (ordinamento)
- `binary_search` (ricerca)
- `fibonacci` (ricorsione/iterazione, sequenza numerica)
- `matrix_multiplication` (operazioni su matrici)
- `is_palindrome` (manipolazione stringhe/numeri)

Questi sono stati scelti per la loro capacità di rappresentare un'ampia gamma di problemi di programmazione:

- **Dalla Semplicità alla Complessità:** `bubble_sort` e `binary_search` sono algoritmi relativamente semplici e ampiamente conosciuti, spesso utilizzati come esercizi base in contesti di programmazione; si può ipotizzare che tutti i modelli abbiano nei dati di training questi algoritmi. `fibonacci` introduce concetti di ricorsione o iterazione più avanzati, introducendo la "libertà di scelta" agli LLM. `matrix_multiplication` e `is_palindrome` sono stati selezionati per la loro natura intrinsecamente più complessa e per la possibilità di diverse interpretazioni implementative.
- **Libertà di Interpretazione:** Come sopracitato per tutti gli algoritmi - in particolare per `fibonacci`, `matrix_multiplication` e `is_palindrome` - non sono state fornite ulteriori informazioni o vincoli specifici oltre al nome dell'algoritmo nel prompt. Questa decisione è stata presa per testare la capacità degli LLM di interpretare autonomamente la richiesta, scegliere un'implementazione ragionevole e mantenere la coerenza nella loro scelta.
- **Requisito Funzionale (Funzioni con Passaggio di Variabili):** È stato considerato di fondamentale importanza che gli algoritmi fossero implementati come

funzioni che accettano e restituiscono valori tramite parametri/variabili, anziché implementazioni statiche o con valori hardcoded. Le soluzioni che non hanno rispettato questo requisito sono state penalizzate nella valutazione, poiché un'implementazione funzionale è cruciale per la riusabilità e la testabilità del codice.

3.4 Selezione dei Linguaggi di Programmazione

La diversità dei linguaggi di programmazione selezionati mira a testare la versatilità degli LLM in un ampio spettro di paradigmi e popolarità, dalla programmazione di sistema a quella web, dalla scienza dei dati al multi-paradigma. I linguaggi sono stati ordinati, per quanto possibile, in base alla loro notorietà e diffusione nel panorama globale dello sviluppo software, prendendo come riferimento indici di popolarità generali come il TIOBE Index[1] o le statistiche di Stack Overflow Developer Survey[2].

I linguaggi inclusi nello studio sono i seguenti:

- **Python:** Estremamente popolare per lo sviluppo web, data science, machine learning e scripting. Noto per la sua sintassi chiara e leggibile e per un vasto ecosistema di librerie. Generalmente tra i primi posti negli indici di popolarità.

- **C++:** Un linguaggio potente e performante, estensione del C, utilizzato per lo sviluppo di sistemi operativi, giochi, applicazioni ad alta performance e software embedded. Richiede una gestione più esplicita della memoria. La sua complessità lo rende meno diffuso tra i principianti, ma rimane un pilastro nell'industria.

- **C:** Fondamentale per la programmazione di sistema, sistemi embedded e kernel di sistemi operativi. È un linguaggio a basso livello che offre un controllo diretto sull'hardware. Nonostante l'età, mantiene una posizione rilevante per compiti specifici.

- **Java:** Un linguaggio *object-oriented* molto diffuso per lo sviluppo enterprise, applicazioni Android e sistemi back-end su larga scala. È un linguaggio robusto e maturo, spesso tra i primi linguaggi più utilizzati.

- **JavaScript:** Il linguaggio fondamentale del web, essenziale per lo sviluppo *frontend* e, con Node.js, anche per il *backend*. La sua onnipresenza lo rende uno dei linguaggi più richiesti e utilizzati, costantemente tra i primi cinque.

[1] https://www.tiobe.com/tiobe-index/

[2] https://survey.stackoverflow.co/

- **R:** Un linguaggio e ambiente software per l'analisi statistica e la grafica. Estremamente popolare nella comunità accademica e scientifica per l'analisi dei dati e la visualizzazione. La sua sintassi è spesso considerata meno intuitiva per i programmatori generalisti.

- **Rust:** Un linguaggio relativamente nuovo focalizzato sulla sicurezza della memoria, la concorrenza e le performance. Sta guadagnando rapidamente popolarità nello sviluppo di sistemi, *web assembly* e *blockchain*. La sua curva di apprendimento è più ripida.

- **Scala:** Un linguaggio multi-paradigma che unisce la programmazione orientata agli oggetti e funzionale, eseguito sulla Java Virtual Machine (JVM). È spesso usato per lo sviluppo di applicazioni distribuite e *big data*. La sua complessità lo rende meno diffuso ma molto apprezzato in nicchie specifiche.

- **Julia:** Un linguaggio ad alte prestazioni progettato per l'analisi numerica e la *computational science*. È compilato *just-in-time* (JIT) e mira a combinare la facilità d'uso di Python con la velocità di C. La sua nicchia di utilizzo è in crescita, ma rimane meno noto rispetto ai giganti.

- **Crystal:** Un linguaggio relativamente giovane con una sintassi ispirata a Ruby, ma con tipizzazione statica e compilazione a codice macchina, mirando a combinare la produttività di Ruby con le performance di C. La sua adozione è ancora limitata, posizionandolo tra i linguaggi meno noti e utilizzati su larga scala.

3.5 Prompt Design e Condizioni di Inferenza

La formulazione del prompt è stata attentamente definita per garantire coerenza e minimizzare le ambiguità, consentendo agli LLM di concentrarsi sulla generazione del codice puro. Il prompt standardizzato utilizzato è il seguente:

> Write ONLY the pure source code for the `{algorithm}` algorithm in `{language}`. DO NOT INCLUDE ANY COMMENTS, explanations, docstrings, markdown formatting, or main function. Absolutely NO #, //, /*, or any other comment syntax. Only pure code.

- **Sostituzione Dinamica:** Le stringhe `{algorithm}` e `{language}` sono state sostituite dinamicamente nello script di automazione per interrogare ogni LLM su tutte le combinazioni definite.

- **Zero-shot prompting:** Il prompt è stato strutturato in modo da essere semplice, infatti viene fornito all'LLM solo un'istruzione senza esempi di come dovrebbe essere la risposta.

- **Purezza del Codice Richiesta:** La direttiva "ONLY the pure source code" e le successive specificazioni "DO NOT INCLUDE ANY COMMENTS, explanations, docstrings, markdown formatting, or main function. Absolutely NO #, //, /*, or any other comment syntax. Only pure code." sono state incluse per imporre agli LLM di generare un output minimalista e focalizzato esclusivamente sul corpo della funzione dell'algoritmo, senza elementi aggiuntivi che avrebbero potuto alterare il conteggio dei token o la purezza del codice.

- **Inferenza Singola:** Ogni modello è stato interrogato per *inferenza singola*, senza mantenere uno storico della conversazione (modalità *stateless*). Questa scelta è stata cruciale per evitare che l'output di un modello fosse influenzato da contesti precedenti o da interazioni di chat, garantendo che ogni generazione di codice fosse una risposta indipendente al prompt specifico.

- **Lingua del Prompt:** Il prompt è stato formulato in inglese per prevenire potenziali incomprensioni o bias legati alla conoscenza dell'italiano da parte di alcuni modelli, specialmente quelli di dimensioni più ridotte, che potrebbero avere una maggiore familiarità con l'inglese nel loro corpus di training.

- **Post-Elaborazione dei Commenti:** Nonostante la richiesta esplicita di evitare i commenti, alcuni modelli non hanno rispettato questa direttiva, includendo commenti o elementi markdown nel loro output. Pertanto, è stata eseguita una fase di pulizia degli script generati prima della valutazione finale. *Gli script originali non modificati sono disponibili nel repository del progetto per ragioni di trasparenza e riproducibilità.*

3.6 Scala di Valutazione

La valutazione delle performance degli LLM è stata condotta utilizzando una scala di punteggio personalizzata da 0 a 5, progettata per catturare la qualità e la correttezza del codice generato, dalla sua assenza totale alla perfezione funzionale.

La scala di valutazione è definita come segue:

- **[0] Codice non generato:** L'LLM non è riuscito a produrre alcun output di codice nel linguaggio di programmazione richiesto: ha fornito un output assente, in un altro linguaggio o solo testo/pseudocodice.
  - **Esempio:** Prompt per "bubble_sort" in Python, l'output prevede del testo/pseudocodice o il "bubble_sort" in un altro linguaggio.

- **[1] Codice generato, ma inutilizzabile (errori lessicali e/o sintattici molto gravi):** Il codice presenta errori lessicali o sintattici così gravi, numerosi o fondamentali da impedire qualsiasi utilizzo dello stesso. Non può essere compilato né interpretato correttamente e risulta privo di senso.

  - **Esempio:** In C++, codice con ‘int main;‘ senza parentesi, o tipi di dati non dichiarati, o ‘for (i=0; i<N; i++);‘ senza un corpo del ciclo valido.

- **[2] Codice generato, ma con gravi errori logici:** Il codice è sintatticamente e semanticamente valido e può essere eseguito senza errori di compilazione o interpretazione, ma presenta una logica completamente errata o fallace. Non svolge correttamente il compito per cui è stato generato.

  - **Esempio:** Implementazione di “bubble_sort” che restituisce un array non ordinato o con elementi mancanti, pur essendo sintatticamente corretto.

- **[3] Codice generato, ma con errori semantici o lievi errori logici:** Il codice è sintatticamente corretto, tuttavia presenta errori semantici oppure lievi errori logici che influiscono sulla precisione del risultato.

  - **Esempio:** Implementazione di “binary_search” che funziona per array pari ma fallisce per array dispari, o che non gestisce correttamente i casi limite (e.g. array vuoto o elemento non trovato).

- **[4] Codice generato, ma con errori lessicali e/o sintattici minori:** Il codice presenta uno o pochi errori sintattici e/o lessicali che sono facili da identificare e correggere. Una volta risolti, può essere utilizzato per la sua correttezza funzionale.

  - **Esempio:** In Java, un punto e virgola mancante alla fine di un’istruzione, o una parentesi graffa chiusa in modo errato che causa un errore di compilazione superficiale.

- **[5] Codice perfettamente funzionante:** Il codice è corretto da tutti i punti di vista: lessico, sintassi, logica e produce l’output atteso. È una soluzione completa e robusta al problema richiesto.

Inizialmente, era stata considerata l’introduzione di un livello aggiuntivo (un livello **[6]**) per valutare l’efficienza e la struttura del codice perfettamente funzionante (declassando l’attuale livello **[5]** per il codice ottimizzato). Tuttavia, si è deciso di mantenere la scala a cinque livelli per evitare di introdurre troppe variabili nella valutazione e allontanarsi dal focus primario del progetto, ovvero valutare la capacità di generare codice corretto con un numero di token utilizzati ristretto. L’obiettivo è stato quello di ottenere un equilibrio tra la granularità della valutazione e la praticità dell’analisi.

3.7 Metodologia di Conteggio dei Token

Il conteggio dei token generati da ciascun LLM è stato un'altra metrica fondamentale per valutare l'efficienza degli output. Questa metrica è cruciale per comprendere il "costo" e la "concisione" del codice prodotto da ciascun modello.

Il conteggio dei token è stato realizzato tramite un modulo Python dedicato, che sfrutta la libreria `transformers` di Hugging Face. **Hugging Face** è una piattaforma leader e una comunità che fornisce strumenti, dataset e modelli pre-addestrati per lo sviluppo di applicazioni di intelligenza artificiale, in particolare nel campo del *Natural Language Processing* (NLP) e dei modelli generativi.

Specificamente, è stata utilizzata la classe `AutoTokenizer` per caricare i *tokenizer* pre-addestrati corrispondenti a ciascun modello. Un *tokenizer* è uno strumento che converte il testo in una sequenza numerica di token, che è il formato che i modelli linguistici possono elaborare. L'utilizzo di tokenizer specifici per ogni modello garantisce che il conteggio dei token sia coerente con il modo in cui il modello stesso "vede" e processa il testo.

Per l'accesso ai modelli e ai tokenizer su Hugging Face, è stato utilizzato il meccanismo di autenticazione tramite `login` con un token API, recuperato da variabili d'ambiente per motivi di sicurezza. Lo script di conteggio ha quindi elaborato il codice generato da ogni LLM, fornendo il numero esatto di token per ogni output. Ulteriori dettagli sull'implementazione specifica di questo script saranno trattati nella sezione successiva, dedicata allo sviluppo degli script di automazione.

3.8 Fasi del Processo di Sviluppo del Progetto

Il progetto è stato concepito e sviluppato attraverso un processo strutturato in cinque fasi principali, ciascuna gestita tramite script Python dedicati, linguaggio selezionato per la sua versatilità e semplicità nelle operazioni di automazione e manipolazione dati.

1. **Generazione degli Script:** Questa prima fase ha avuto come obiettivo la produzione automatica del codice sorgente da parte di ogni LLM per tutte le combinazioni di algoritmi e linguaggi predefinite. L'esecuzione è avvenuta tramite lo script Python `generate.py`. Questo script contiene una logica principale che itera su tutti gli algoritmi, i linguaggi e i modelli selezionati. Per ogni combinazione, viene richiamata una funzione di inferenza specifica alla quale vengono passati il modello corrente e un prompt generato dinamicamente, adattato all'algoritmo e al linguaggio in questione. L'output di codice generato da ogni LLM viene successivamente salvato in una struttura di cartelle così organizzata: ogni file di codice è collocato in una sottocartella che porta il nome dell'algoritmo, a sua volta contenuta in una cartella denominata con il nome del linguaggio, il tutto all'interno di una cartella radice chiamata `scripts`.

Di seguito è rappresentato un esempio della struttura delle cartelle adottata:

```
./scripts/
├── c/
│   ├── bubble_sort/
│   │   ├── deepseek-coder:1.3b.c
│   │   └── deepseek-coder:6.7b.c
│   └── binary_search/
│       └── ...
└── python/
    └── ...
```

2. **Modifica Manuale degli Script:** La seconda fase si è composta di un processo di pre-elaborazione manuale degli script generati, per garantirne la purezza e l'aderenza ai requisiti di valutazione. Inizialmente, la cartella `scripts` è stata rinominata in `original_scripts` e una sua copia è stata creata con il nome `modified_scripts`. Successivamente, è stato effettuato un controllo scrupoloso di ogni singolo script nella cartella `modified_scripts`. L'intervento manuale ha comportato la rimozione di eventuali formattazioni Markdown, testi esplicativi esterni al codice e commenti interni che rispettavano la sintassi del linguaggio specifico (e.g., `#`, `//`, `/* ... */`). Eventuali commenti posti con sintassi errata o elementi non codice non sono stati rimossi e sono stati considerati negativamente nella fase di valutazione successiva.

3. **Conteggio dei Token:** La terza fase ha riguardato la quantificazione dei token per ogni script di codice generato, essenziale per l'analisi dell'efficienza dei modelli. Questa operazione è stata gestita dallo script `tokens_count.py`. Lo script è configurato per accedere all'URL di Hugging Face corrispondente a ciascun modello, al fine di prelevare il tokenizer appropriato (come descritto nella Sezione 3.7). L'esecuzione dello script è avvenuta per ciascun linguaggio individualmente, evitando un ciclo automatico: questa scelta metodologica ha permesso di registrare i risultati del conteggio dei token in fogli di calcolo specifici per ogni linguaggio, fungendo da dati grezzi. Lo script accede alle cartelle del linguaggio (all'interno di `modified_scripts`), elabora tutti gli script nelle sottocartelle degli algoritmi e restituisce un file `.txt` contenente tutte le informazioni sul conteggio dei token per ciascun output.

4. **Valutazione della Correttezza degli Script:** La quarta fase è stata dedicata alla valutazione della correttezza funzionale e sintattica degli script. Per ogni algoritmo e linguaggio, è stato eseguito lo script `merge_code.py`. Questo script ha la funzione di unire tutte le singole funzioni di codice generate dai diversi LLM per uno specifico algoritmo e linguaggio in un unico file. I nomi degli LLM sono stati inseriti come segnaposto al di sopra di ogni funzione, per identificare chiaramente l'autore del codice. Successivamente, avvalendosi di compilatori e interpreti online, il codice unito è stato studiato e testato. L'analisi ha incluso l'esecuzione del codice con input di prova per verificarne il funzionamento e l'identificazione di eventuali errori sintattici, logici o semantici.[3] Tutte le valutazioni sono state salvate in un file `.txt`, denominato `{linguaggio}_evaluations.txt` e collocato nella cartella `scripts_evaluations`. In seguito, i dati sono stati trasferiti manualmente nei fogli di calcolo, completando le tabelle dei dati grezzi.

5. **Generazione della Dashboard:** L'ultima fase ha riguardato la visualizzazione e l'analisi dei dati raccolti. Una volta che i dati grezzi (punteggi di valutazione e conteggi dei token) sono stati consolidati, sono stati salvati direttamente all'interno del codice Python come semplici stringhe. Questi dati sono stati poi convertiti in *dataframe* grazie all'utilizzo di opportuni delimitatori. Utilizzando la libreria `streamlit`, è stata costruita una dashboard interattiva. Questa dashboard ha permesso la produzione di tabelle e grafici riassuntivi, che sono stati successivamente integrati e utilizzati per l'analisi e la presentazione dei risultati finali nella relazione.

In Figura 3.1 è riportato il diagramma di sequenza dell'intero processo.

[3] *Nota Bene:* Per la fase di test e la rilevazione degli errori, l'autore si è avvalso della collaborazione di LLM di grandi dimensioni come GPT (https://chatgpt.com/) e Gemini (https://gemini.google.com/), utilizzandoli come copiloti intelligenti. Questi modelli sono stati fondamentali nel fornire indicazioni chiare su come testare il codice in linguaggi meno familiari all'autore e nell'assistere nell'identificazione di errori complessi, migliorando l'accuratezza e l'efficienza del processo di valutazione.

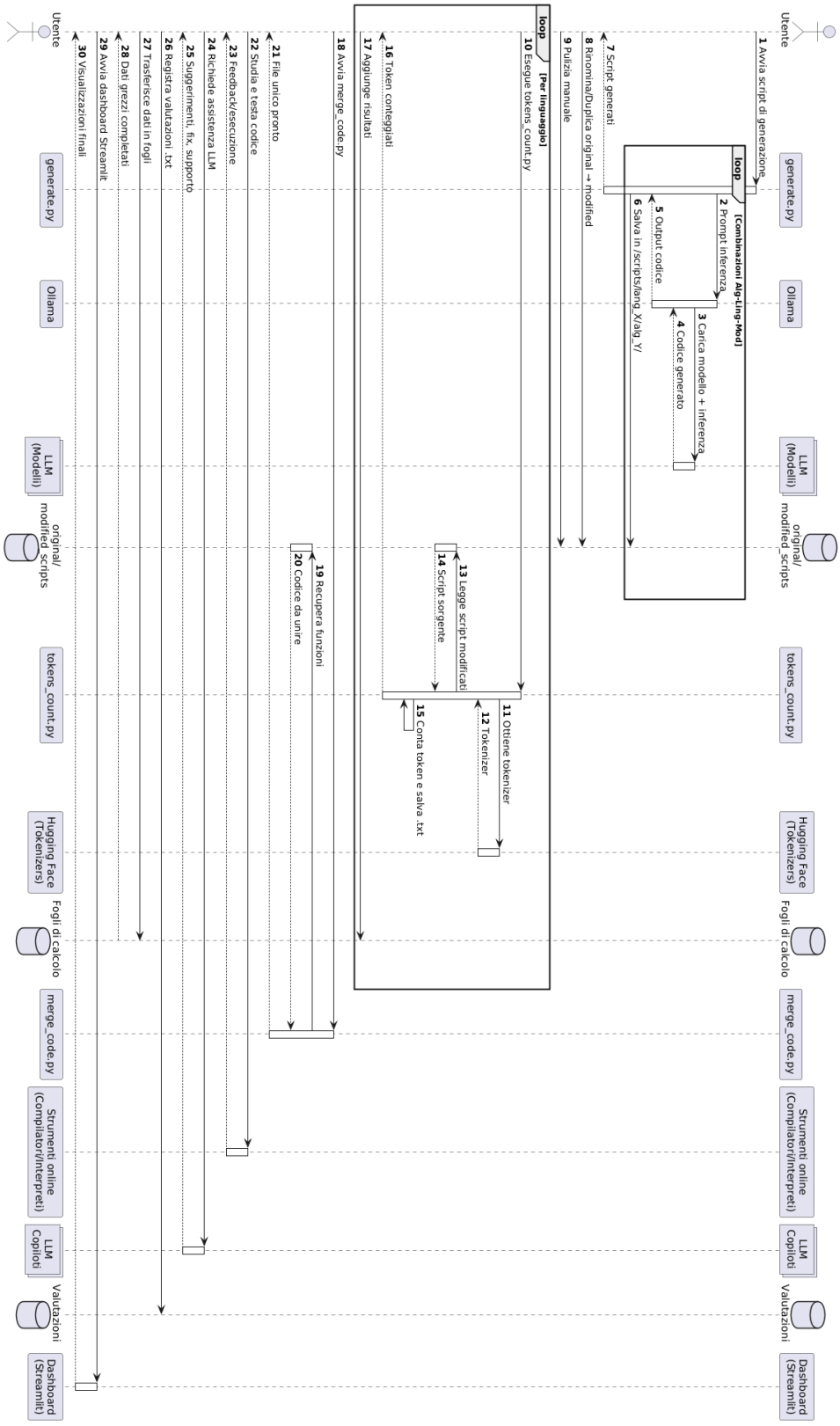


Figura 3.1: Diagramma di Sequenza del Processo di Sviluppo del Progetto

Capitolo 4

Analisi dei Risultati

Questo capitolo presenta l'analisi dettagliata dei risultati ottenuti dal confronto delle performance di otto Large Language Models (LLM) nella generazione di codice per i cinque algoritmi nei dieci linguaggi di programmazione citati nel Capitolo 3. I risultati sono stati valutati in base a una scala personalizzata riportata nella Sezione 3.6 e al numero di token generati nell'output.

4.1 Panoramica delle Performance Complessive degli LLM

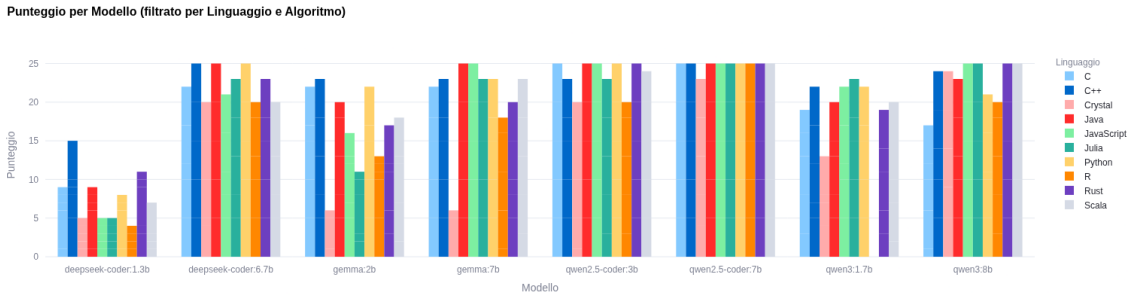
L'analisi aggregata delle performance dei modelli LLM su la varietà di algoritmi e linguaggi di programmazione rivela tendenze significative in termini di accuratezza e concisione del codice generato.

La Tabella 4.1 riassume, per ciascun modello, il punteggio medio assegnato e il numero medio di token prodotti durante la generazione delle soluzioni.

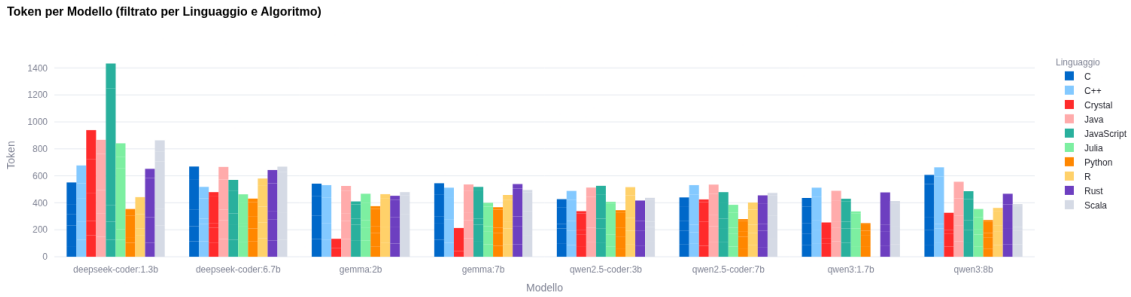
Tabella 4.1: Punteggi medi e token medi generati per ciascun LLM.

Modello LLM	**Punteggio Medio**	**Token Medi**
qwen2.5-coder:7b	4.96	88.08
qwen2.5-coder:3b	4.70	88.28
qwen3:8b	4.58	89.66
deepseek-coder:6.7b	4.48	113.74
gemma:7b	4.16	91.66
qwen3:1.7b	3.60	71.92
gemma:2b	3.36	87.56
deepseek-coder:1.3b	1.56	152.38

Le Figure 4.1a e 4.1b mostrano le performance complessive dei diversi modelli nei vari linguaggi e algoritmi.



(a) Punteggi degli LLM filtrati per linguaggi e algoritmi.



(b) Token utilizzati dagli LLM filtrati per linguaggi e algoritmi.

Figura 4.1: Confronto tra le performance medie (punteggio) e l'efficienza (token) dei modelli LLM analizzati.

Considerazioni Chiave

- **Correlazione con la Dimensione del Modello:** I modelli di dimensioni maggiori (ad esempio, `deepseek-coder:6.7b`, `qwen2.5-coder:7b`, `gemma:7b`, `qwen3:8b`) ottengono sistematicamente punteggi medi superiori, suggerendo una maggiore capacità di generare codice corretto e funzionale (con punteggi pari a 5). In particolare, `qwen2.5-coder:7b` si distingue come uno dei migliori modelli in termini di accuratezza complessiva, mostrando la superiorità nella produzione di codice presente nei modelli coder.

- **Efficienza in termini di Token:** Sebbene modelli come `deepseek-coder:6.7b` riportino valori elevati in accuratezza, modelli come `qwen2.5-coder:7b`, `gemma:7b`, `qwen2.5-coder:3b` e `qwen3:8b` dimostrano un'efficienza superiore, combinando punteggi elevati con un numero medio di token significativamente inferiore. Ciò indica una maggiore concisione nella generazione del codice senza comprometterne la qualità.

- **Performance dei Modelli di Piccola Scala:** I modelli di dimensioni ridotte, in particolare `deepseek-coder:1.3b` e `qwen3:1.7b`, mostrano performance

complessivamente inferiori. Il primo, pur generando la quantità più elevata di token medi, ottiene il punteggio medio più basso, suggerendo un'efficienza molto limitata e frequenti errori. Il secondo, sebbene più conciso, raggiunge comunque punteggi modesti, evidenziando difficoltà ricorrenti nella generazione di codice valido, inclusi numerosi casi di *codice non generato*, in particolare in linguaggi meno comuni come R e Crystal, come confermato dalle occorrenze di codice non generato.

4.2 Performance per Linguaggio di Programmazione

L'efficacia degli LLM varia considerevolmente a seconda del linguaggio di programmazione richiesto. La difficoltà percepita di un linguaggio per gli LLM può essere indicativa della sua complessità sintattica, della sua popolarità nei dataset di training o della specificità dei suoi idiomi.

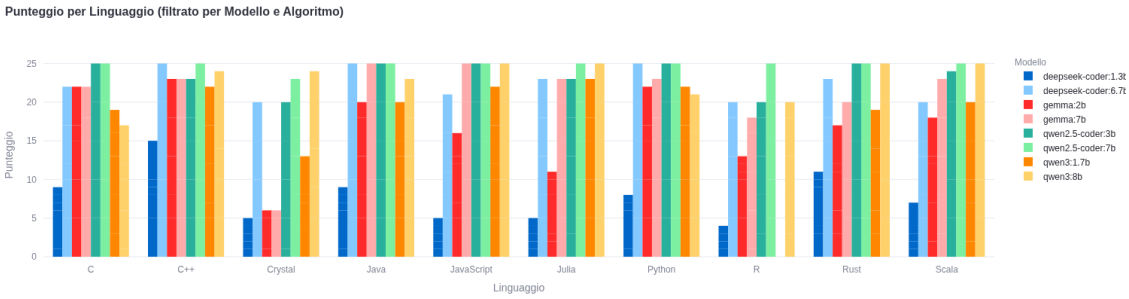
Tabella 4.2: Punteggi Medi e Token Medi per Linguaggio di Programmazione

Linguaggio	**Punteggio Medio**	**Token Medi**
C++	4.50	110.80
Java	4.30	117.18
Python	4.28	66.80
Rust	4.12	102.55
JavaScript	4.10	121.30
Scala	4.05	105.45
C	4.03	105.42
Julia	3.95	91.32
R	3.00	80.60
Crystal	2.92	77.68

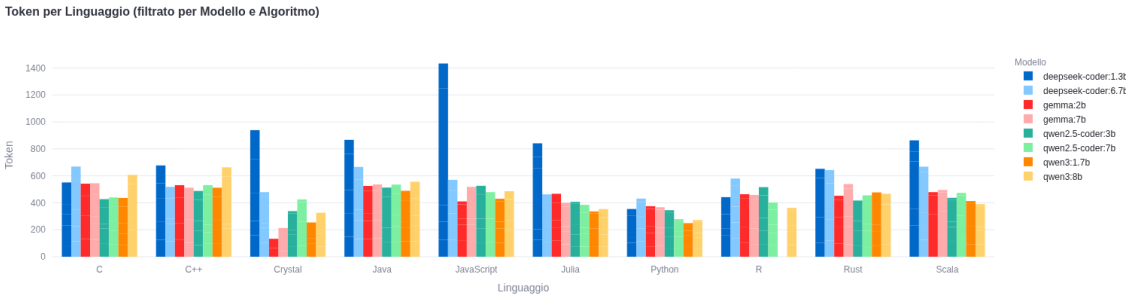
Le Figure 4.2a e 4.2b mostrano le performance complessive dei diversi modelli nei diversi linguaggi.

Considerazioni Chiave:

- **Eccellenza in C++:** C++ si distingue come il linguaggio in cui gli LLM hanno performato meglio complessivamente, suggerendo una robusta capacità di gestire la sua sintassi complessa e le sue peculiarità.
- **Linguaggi "Legacy" e Popolari:** Java e Python, seguiti da Javascript, mostrano performance solide, spesso generando codice funzionante o con errori leggeri. Questo potrebbe essere attribuito alla loro vasta presenza nei dataset di



(a) Punteggi degli LLM per singoli linguaggi.



(b) Token utilizzati dagli LLM per singoli linguaggi.

Figura 4.2: Confronto tra le performance medie (punteggio) e l'efficienza (token) dei modelli LLM analizzati.

training, che permette agli LLM di apprendere schemi e idiomi comuni. Python si distingue per l'efficienza dei token, producendo soluzioni concise.

- **Sfide con Linguaggi Specifici:** Crystal e R si rivelano i linguaggi più ostici per la maggior parte degli LLM, con punteggi medi significativamente inferiori. Molti modelli, inclusi alcuni dei più performanti, hanno registrato punteggi bassi in questi linguaggi. Questo potrebbe essere dovuto alla loro minore rappresentazione nei dati di training, alla loro sintassi peculiare o a una minore diffusione complessiva rispetto ad altri linguaggi.

  – In **Crystal**, gemma:2b e gemma:7b hanno mostrato particolari difficoltà, con diverso codice generato in altri linguaggi o punteggi molto bassi. Anche deepseek-coder:1.3b e qwen3:1.7b hanno avuto performance estremamente scarse, con frequenti fallimenti totali.

  – In **R**, la situazione è ancora più critica per i modelli più piccoli, con deepseek-coder:1.3b e qwen3:1.7b che spesso non producono output validi, confermando il caso specifico di `qwen3:1.7b` in `R` che ha generato codice scritto in altri

linguaggi. Anche gemma:7b e qwen2.5-coder:3b mostrano cali di performance, con più frequenti errori logici o semantici arrivando anche a produrre codice inutilizzabile.

- **Versatilità dei Top Performer:** I modelli top-tier come deepseek-coder:6.7b, qwen2.5-coder:7b e gemma:7b mantengono prestazioni elevate nella maggior parte dei linguaggi, ma anche loro mostrano una diminuzione in Crystal e R, seppur meno pronunciata rispetto ai modelli più piccoli. Tuttavia, anche per questi modelli, il rischio di ottenere codice con errori logici significativi o addirittura inutilizzabile aumenta drasticamente in questi linguaggi meno comuni.

4.3 Performance per Algoritmo

La complessità e la natura intrinseca di un algoritmo influenzano la capacità degli LLM di generare una soluzione corretta.

Tabella 4.3: Punteggi Medi e Token Medi per Algoritmo

Algoritmo	**Punteggio Medio**	**Token Medi**
fibonacci	4.15	70.99
is_palindrome	4.01	57.10
binary_search	4.00	106.65
bubble_sort	3.85	98.82
matrix_multiplication	3.61	155.99

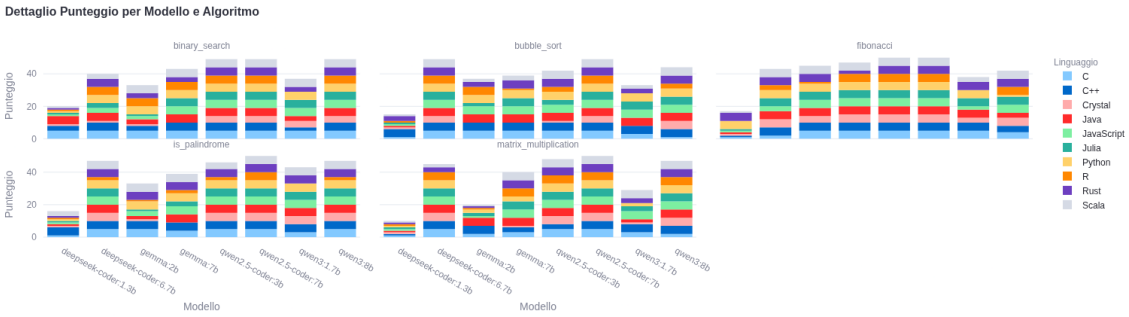
Le Figure 4.3a e 4.3b mostrano le performance complessive dei diversi modelli per singolo algoritmo.

Considerazioni Chiave:

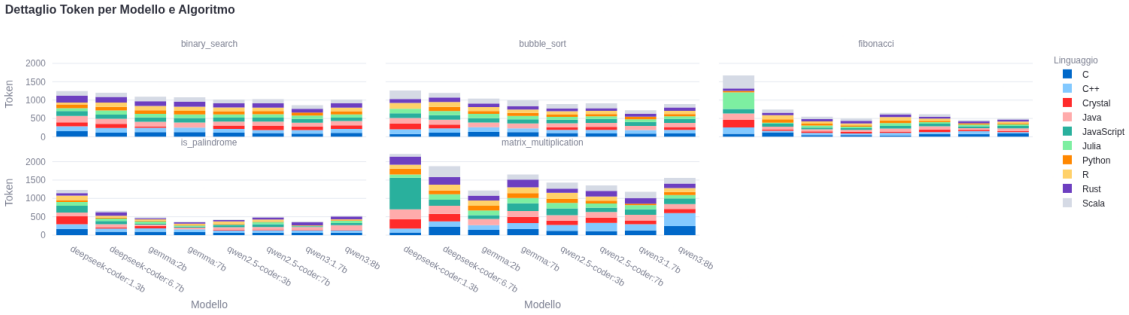
- **Algoritmi più Semplici:** `fibonacci`, `is_palindrome` e `binary_search` si sono rivelati gli algoritmi in cui gli LLM hanno performato meglio, spesso producendo codice perfettamente funzionante o con solo errori minori. Questo è atteso, data la loro natura relativamente semplice e la loro frequente presenza negli esempi di codice nei dati di addestramento. `is_palindrome` spicca anche per il numero di token più basso, indicando soluzioni concise.

- **Algoritmi più Complessi:** `matrix_multiplication` e `bubble_sort` hanno presentato maggiori sfide, con punteggi medi inferiori. In questi casi, è più probabile incontrare codice con errori logici o semantici o persino inutilizzabile.

  – `matrix_multiplication` è l'algoritmo che ha richiesto il maggior numero di token, il che è comprensibile data la sua natura più complessa e la necessità di gestire cicli annidati e strutture dati multi-dimensionali. Nonostante



(a) Punteggi degli LLM per singoli algoritmi filtrati per linguaggi.



(b) Token utilizzati dagli LLM per singoli algoritmi filtrati per linguaggi.

Figura 4.3: Confronto tra le performance medie (punteggio) e l'efficienza (token) dei modelli LLM analizzati.

ciò, alcuni modelli (come i deepseek-coder:6.7b e qwen2.5-coder:7b/3b) sono riusciti a ottenere punteggi elevati.

– `bubble_sort`, pur essendo un algoritmo di ordinamento concettualmente semplice, ha mostrato una performance media moderata, forse a causa della sua inefficienza intrinseca che talvolta porta gli LLM a generare codice più "pedante" o meno ottimizzato in termini di concisione, o con errori semantici minori.

- **Valutazione indipendente:** `fibonacci`, `is_palindrome` e `matrix_multiplication` sono gli algoritmi che hanno presentato più versioni alternative (e.g. `is_palindrome` è stato implementato sia con stringhe, sia in forma puramente numerica, sia in modalità ibride); nonostante ciò la correttezza dell'implementazione è stata valutata in modo totalmente indipendente, garantendo il punteggio massimo a tutte le soluzioni funzionanti.

4.4 Relazione tra Punteggio e Numero di Token

Analizzando la relazione tra il punteggio di valutazione e il numero di token generati, emergono alcune osservazioni interessanti:

- **L'inefficienza dei token nei modelli più piccoli spesso correla con bassa qualità:** deepseek-coder:1.3b è l'esempio più eclatante. Pur generando un volume elevato di token (media di 143.47), ottiene il punteggio più basso (1.30). Questo suggerisce che i suoi output sono spesso prolissi e scorretti, rientrando nelle categorie di codice inutilizzabile o con gravi errori logici.
- **I modelli più efficienti sono i migliori performer:** I modelli che si classificano ai vertici per punteggio medio (deepseek-coder:6.7b, qwen2.5-coder:7b, gemma:7b, qwen2.5-coder:3b, qwen3:8b) tendono a generare un numero di token relativamente basso o moderato. In particolare, qwen2.5-coder:7b, qwen2.5-coder:3b e qwen3:8b dimostrano un'eccellente concisione (meno di 90 token medi) pur mantenendo punteggi molto elevati. Ciò indica che questi modelli sono capaci di identificare e implementare soluzioni ottimali con il minimo indispensabile di codice.
- **Casi di "fallimento totale" (0 token):** Le occorrenze di codice non generato indicano un fallimento completo nella generazione di codice significativo. Questo è più frequente nei modelli più piccoli (es. deepseek-coder:1.3b, qwen3:1.7b, gemma:2b) e nei linguaggi più complessi o meno comuni (Crystal, R), suggerendo che in questi contesti, i modelli non riescono proprio a produrre un output coerente con la richiesta.

La Figura 4.4 mostra la relazione punteggio/token in base ai modelli e gli algoritmi richiesti.

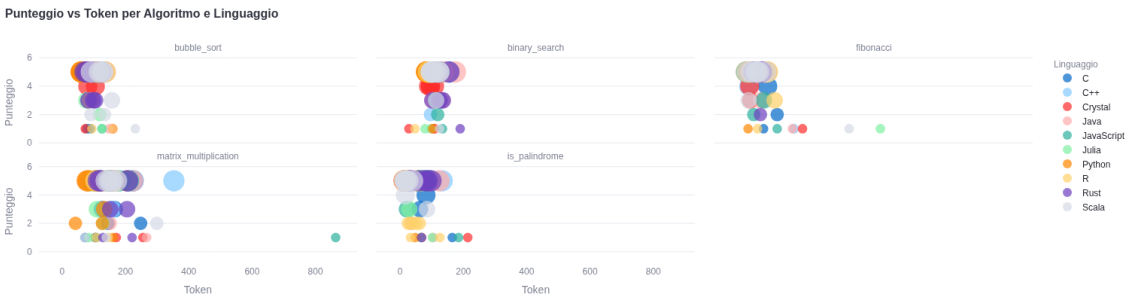


Figura 4.4: Punteggio in relazione ai token.

Capitolo 5

Conclusioni

5.1 Riepilogo delle Scoperte Principali

Da quanto dedotto dal Capitolo 2 la scelta di un LP è un aspetto cruciale dello sviluppo software automatizzato. La prototipazione rapida e l'assenza di necessità di prestazioni di runtime rendono Python e Javascript tra i candidati preferibili grazie all'efficienza del codice generato e alla mole di dati disponibili. Per applicazioni in cui le prestazioni e la sicurezza giocano un ruolo fondamentale, i linguaggi citati precedentemente sarebbero meno indicati. Ad oggi non vi è la possibilità di identificare un linguaggio migliore, ma sta allo sviluppatore prendere questo tipo di decisioni o in alternativa sfruttare la potenza del prompt engineering per lasciare che l'LLM (salvo bias discussi nel capitolo sopracitato) lo guidi nella scelta più opportuna. Il futuro dello sviluppo software automatizzato coinvolgerà sempre in maniera più significativa l'interazione con assistenti IA per la generazione di codice e la risoluzione di bug ed è per questo che l'obiettivo del presente elaborato era quello di cercare di capire con quale linguaggio il costo fosse inferiore. Come visto, non esiste una soluzione universalmente ottimale, ma vi sono effettivamente delle preferenze. Linguaggi che hanno una sintassi semplice, più vicina al linguaggio naturale e con una mole vasta di dati di addestramento sono da preferire per uno sviluppo automatizzato. L'introduzione di un nuovo LP, per quanto l'idea possa apparire innovativa, sarebbe priva di dati di addestramento o ne avrebbe in quantità minore rispetto a possibili alternative. L'idea alla base di Mojo, creare un superset di un LP già ottimizzato e LLM-friendly, potrebbe rivelarsi un'idea vincente in un futuro prossimo. È verosimile che nei prossimi anni assisteremo non solo all'evoluzione degli LLM e delle loro architetture, ma anche alla co-evoluzione dei LP stessi, tramite framework, librerie o superset sempre più pensati per collaborare con l'IA e non solo per gli esseri umani.

5.2 Punti Fondamentali dell'Elaborato

Il lavoro svolto nei Capitoli 3 e 4 sottolinea dei risultati importanti, in linea con lo Stato dell'Arte, e aggiunge ulteriori considerazioni evidenziando i fattori più importanti:

- **L'Importanza della Dimensione del Modello:** I risultati evidenziano chiaramente che, per la task di generazione di codice in un contesto multi-linguaggio e multi-algoritmo, la dimensione del modello è un fattore predittivo significativo delle performance. I modelli con un numero maggiore di parametri tendono a mostrare una comprensione più profonda e una maggiore capacità di sintesi del codice, traducendosi in una maggiore probabilità di generare codice perfettamente funzionante.
- **La Curva di Apprendimento dei Linguaggi:** L'ampia variazione nelle performance tra i diversi linguaggi suggerisce che la capacità di un LLM di generare codice non è uniforme. Linguaggi più specifici o meno rappresentati nei corpus di training costituiscono ancora una sfida, portando a un aumento di codice inutilizzabile o con gravi errori logici. Per un'applicazione pratica, sarebbe fondamentale selezionare un LLM addestrato specificamente o con un'ampia esposizione al linguaggio di interesse.
- **Ottimizzazione tra Qualità e Token:** Il trade-off tra punteggio di qualità e numero di token è cruciale per applicazioni in cui la latenza o i costi legati ai token rappresentano una preoccupazione.
- **Punti di Fallimento Critici:** L'identificazione di scenari di "fallimento totale", ovvero codice non generato, è fondamentale. Questi casi non indicano semplicemente una scarsa performance, ma una incapacità del modello di produrre qualsiasi output utilizzabile. Questo suggerisce la necessità di strategie di fallback o di modelli specializzati per linguaggi e algoritmi specifici in cui le performance sono imprevedibili e il rischio di non ottenere alcun codice è elevato.
- **Specificità degli Algoritmi:** La difficoltà relativa degli algoritmi suggerisce che la complessità del problema influenza direttamente le capacità del modello. Algoritmi che richiedono una maggiore logica di controllo o manipolazione di strutture dati complesse tendono a mettere maggiormente alla prova gli LLM, aumentando la probabilità di ottenere codice con errori logici o semantici.

In sintesi, i dati indicano un panorama differenziato delle capacità degli LLM nel coding. Nonostante i modelli più grandi dimostrino una superiorità generale, l'efficienza e la specializzazione in specifici linguaggi o tipi di problemi rimangono fattori distintivi. I risultati di questa analisi gettano le basi per considerazioni più approfondite nella selezione e nell'applicazione pratica degli LLM per compiti di generazione di codice.

5.3 Sviluppi Futuri

Sebbene tutti gli obiettivi preposti da questo elaborato siano stati raggiunti e questo studio abbia fornito una solida base per comprendere le capacità degli LLM nella generazione di codice, è stata anche aperta la strada a numerosi sviluppi e approfondimenti futuri:

- **Automazione delle Fasi Manuali:** Un'area di sviluppo cruciale è l'automazione delle fasi attualmente manuali del processo di valutazione. Questo include l'integrazione di *tool di analisi sintattica, lessicale e semantica* automatizzati per la pulizia del codice e la rilevazione degli errori, eliminando la necessità di interventi umani. L'automazione della raccolta dei dati, mediante pipeline robusti e scalabili, ridurrebbe significativamente i tempi e le risorse necessarie per esperimenti futuri.

- **Espansione Orizzontale e Verticale della Ricerca:**

  - **Orizzontale:** Ampliare la portata dello studio includendo un numero maggiore di LP, per coprire un più vasto spettro di paradigmi e nicchie di utilizzo, introdurre ulteriori *LLM*, inclusi modelli emergenti o quelli con architetture diverse, e prevedere un set più ampio di *algoritmi*, consentirebbe una generalizzazione maggiore dei risultati.
  - **Verticale:** Approfondire l'analisi testando *LLM più grandi* (es. oltre 10B o 50B di parametri) per osservare come le performance e l'efficienza scalino con l'aumento delle dimensioni; parallelamente, l'introduzione di *algoritmi di maggiore complessità* o con requisiti specifici di efficienza (es. algoritmi paralleli, algoritmi su strutture dati complesse) permetterebbe di testare i limiti delle capacità attuali degli LLM.

- **Analisi Approfondita delle Metriche di Efficienza del Codice Generato:** Sebbene questo studio si sia concentrato principalmente sul costo computazionale di generazione (token), un'estensione futura potrebbe includere una valutazione automatizzata delle *performance a runtime* del codice generato (tempo di esecuzione, consumo di memoria, utilizzo della CPU/GPU) e della sua *qualità strutturale* (leggibilità, manutenibilità, aderenza a standard di codice), andando oltre la sola correttezza funzionale[1].

- **Impatto del Prompt Engineering Avanzato:** Esplorare l'influenza di tecniche di *prompt engineering* più sofisticate (es. *few-shot learning*, *chain-of-thought prompting*) sulla qualità e l'efficienza del codice generato, analizzando come la

[1] Verrebbe quindi ripresa l'idea iniziale di espandere la scala di valutazione, vista nella Sezione 3.6, con ulteriori valori.

formulazione del prompt possa guidare l'LLM verso soluzioni più ottimali o idiomatiche per specifici linguaggi.

- **Confronto tra Hosting Locale e Cloud-based LLM:** Valutare le differenze prestazionali e di costo tra l'hosting locale degli LLM (come con Ollama) e l'utilizzo di API di LLM basati su cloud, fornendo indicazioni pratiche per gli sviluppatori sulla scelta dell'infrastruttura più adatta.

Questi sviluppi permetterebbero di arricchire ulteriormente la comprensione delle dinamiche tra LLM e generazione di codice, offrendo strumenti e conoscenze ancora più avanzati per l'adozione dell'IA nello sviluppo software.

Bibliografia

[1] Moonbit: Exploring the design of an ai-native language toolchain, February 2024. Accessed: 2025-05-22.

[2] The role of the scala language and compiler and tooling in the age of llm-supported "automatic coding". https://contributors.scala-lang.org/t/the-role-of-the-scala-language-and-compiler-and-tooling-in-the-age-of-llm-supported-automatic-coding/7042, 2025.

[3] A. A. Abbassi, L. D. Silva, A. Nikanjam, and F. Khomh. Unveiling inefficiencies in llm-generated code: Toward a comprehensive taxonomy, 2025.

[4] Anonymous. Where is the bottleneck of llm code generation? a study isolating llm performance on language-coding from problem-solving. In *ACL ARR 2025 February Submission*, 2025.

[5] G. Appenzeller. Welcome to llmflation - llm inference cost is going down fast. urlhttps://a16z.com/llmflation-llm-inference-cost/, 2025.

[6] G. Benram. Understanding the cost of large language models (llms). urlhttps://www.tensorops.ai/post/understanding-the-cost-of-large-language-models-llms, 2024.

[7] L. Bergmans, X. Schrijen, E. Ouwehand, and M. Bruntink. Measuring source code conciseness across programming languages using compression. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, page 47–57. IEEE, Sept. 2021.

[8] M. Buis. Balancing energy efficiency and accuracy in llm-driven code generation. urlhttps://studenttheses.uu.nl/handle/20.500.12932/48350, 2025.

[9] C. Lattner. Do llms eliminate the need for programming languages?, June 2023. Accessed: 2025-05-22.

[10] Z. Li and X. Lu. Research on compressed input sequences based on compiler tokenization. *Information*, 16(2), 2025.

[11] T. Lu, L. Yangning, L. Wang, B. Lin, J. Tang, W. Xu, H.-T. Zheng, Y. Li, B. An, Z. Wei, and Y. Xu. From token to line: Enhancing code generation with a long-term perspective, 04 2025.

[12] L. Solovyeva, S. Weidmann, and F. Castor. Ai-powered, but power-hungry? energy efficiency of llm-generated code. In *Proceedings of the FORGE 2025 Workshop, co-located with ICSE 2025*, pages 14:36–14:48, Lisbon, Portugal, April 2025. Presented in Session 1: FM for Code Generation, Chair: Lili Wei.

[13] L. Twist, J. M. Zhang, M. Harman, D. Syme, J. Noppen, and D. Nauck. Llms love python: A study of llms' bias for programming languages and libraries, 2025.

[14] Unknown. Rust + webassembly: Building infrastructure for large language model ecosystems, October 2023. Accessed: 2025-05-22.

[15] Unknown. What is a token in ai? understanding how ai processes language with tokenization.
urlhttps://nebius.com/blog/posts/what-is-token-in-ai, 2025. Accessed: 2025-05-22.

[16] S. P. Velaga. Ai-assisted code generation and optimization: Leveraging machine learning to enhance software development processes. *International Journal of Innovations in Engineering Research and Technology*, 7(09):177–186, 9 2020.

[17] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. A systematic evaluation of large language models of code, 2022.

[18] X. Zhang, J. Zhai, S. Ma, Q. Bao, W. Jiang, C. Shen, and Y. Liu. Unveiling provider bias in large language models for code generation, 2025.