ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Integrazione di RAG e LLM nello Sviluppo del Software

Tesi di laurea in:
Programmazione ad oggetti

Relatore
Prof. Viroli Mirko

Candidato
Bollini Simone

Correlatori
Dott. Aguzzi Gianluca
Dott. Farabegoli Nicolas

IV Sessione di Laurea
Anno Accademico 2023-2024

Abstract

I Large Language Model (LLM) addestrati per sviluppare il codice sono oggi altamente efficaci e in grado di generare soluzioni utili e funzionanti. L'addestramento fatto dai modelli è però su fonti e soluzioni generali, questo non da quindi la possibilità al modello di proppore soluzioni su misura per una specifica richiesta utilizzando casistiche già create dal programmatore o dalla propria azienda per casi simili. Da questo nasce l'esigenza di addestrare il modello per personalizzare le soluzioni proposte, contestualizzandole alla propria realtà aziendale e al proprio stile nel programmare. Il LLM non conosce le librerie interne dell'azienda, i pattern di programmazione adottati e quindi le risposte ottenute sono troppo generiche. Per rispondere a questa esigenza entra in gioco la Retrieval-Augmented Generation (RAG) ovvero il processo di ottimizzazione dell'output di un LLM, per permettergli di fruire una base di conoscenza personalizza, unica e privata, questa **matrice di conoscenza** si inserisce tra quanto già appreso dal modello dai dataset utilizzati in fase di addestramento, estendendo la base dati sulla quale generare l'output con la risposta. Questa tesi sperimenta l'integrazione di un RAG con un LLM per ottenere dal modello risposte personalizzate con conoscenze private e specifiche fornite da un dataset personalizzato.

A Giulia e ai miei figli, il dono più grande.
A tutta la mia famiglia.

Grazie a tutti voi.

Contents

Abstract **iii**

1 Introduzione **1**
1.1 Essere programmatori nel 2025 1

2 Addestrare un LLM per la Generazione del Codice **5**
2.1 Raccolta e Preparazione dei Dati 5
2.2 Pre-Addestramento 6
2.3 Fine-Tuning 7
2.4 Pre-Addestramento vs Fine-Tuning 9
2.5 Architettura del Modello 9
2.6 Valutazione e Ottimizzazione 10
2.6.1 Metriche di Valutazione 10
2.6.2 Tecniche di Ottimizzazione 10

3 RAG **11**
3.1 Introduzione 11
3.2 Funzionamento 12
3.2.1 Creazione di Dati Esterni 12
3.2.2 Recupero delle Informazioni 13
3.2.3 Aumento del Prompt 13
3.2.4 Gestione dell'Aggiornamento 13

4 Implementazione di un Sistema RAG per lo Sviluppo del Software: Caso Studio **15**
4.1 Panoramica del Progetto 15
4.2 Architettura del Sistema 15
4.3 Ambiente di Sviluppo 15
4.4 Implementazione 16
4.4.1 Preparazione dei Dati 16
4.5 LORA 16

4.6 FOrmatazione con i TAG di lamas 17
4.6.1 Creazione del Database Vettoriale 17
4.6.2 Integrazione con il LLM 17
4.6.3 Pipeline RAG 17
4.7 Test e Valutazione 18
4.7.1 Metodologia di Test 18
4.7.2 Risultati 18
4.8 Conclusioni e Considerazioni 18

5 Conclusioni **19**
5.1 Risultati Ottenuti 19
5.2 Impatto sullo Sviluppo Software 19
5.3 Sfide e Considerazioni Future 20
5.4 Prospettive Future 20

21

Bibliography **21**

List of Figures

3.1 Flusso di una request ad un LLM integrato con un RAG 12

List of Listings

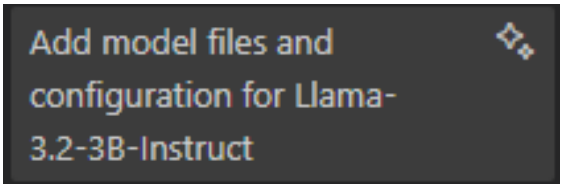
listings/HelloWorld.java 20

Chapter 1

Introduzione

1.1 Essere programmatori nel 2025

Sono disponibili tantissimi (IDE) per lo sviluppo del codice uno di questi è **Visual Studio Code**, mentre **Github** può essere lo strumento utilizzato per contenere e condividere progetti per lavore in maniera collaborativa. Può anche essere molto utile, **COLAB** che permette di eseguire in remoto codice che richiede molta memoria su GPU spesso non disponbili localmente. Questi esempi mostrano una panoramica vasta e complessa, con un frequente cambio di software per realizzare un programma, modifiche fatte localmente su Visual Studio Code vengono trasferite su GitHub e poi riprese su Colab dove a sua volta vengono eseguiti Commit e Push sul progetto radice presente su GitHub. La cosa che accomuna questi strumenti oggi è che dispongono tutti di assistenti di programmazione basati sull'intelligenza artificiale, in grado di completare il codice, suggerire correzioni e creare documentazione pertinente. Lo schema di lavoro appenda descritto è stato da me attuato per realizzare questa tesi, ho utilizzato Visual Studio Code per scrivere il codice Pyhon, GitHub per condividere il progetto e Colab per eseguire la maggior parte del codice. Una delle funzionalità offerte da questi assistenti è la la funzione di **Github Copilot** 'Generate Commit Message with Copilot' che propone il testo da utilizzare come descrizione di un commit, ho provato a riscontrare quanto fosse contestualizzato e coerente con quanto aggiornato e ho ottenuto il seguente risultato:



Ho trovato coerente e giusto quanto proposto ed eseguito il Commit. Quanto è riuscito a fare Copilot è strabiliante, in pochi istanti ha analizzato il contesto dando come output una risposta semplice ma coerente rispetto a quanto cambiato. L'uso di questi strumenti rende il lavoro molto più dinamico e permette di ridurre le interruzioni per cercare una soluzione o per trovare le giuste parole per descrivere quanto fatto.

L'intelligenza artificiale sta rivoluzionando il modo in cui il software viene sviluppato, dando la possibilità a strumenti come Copilot di esplodere tutto il loro potenziale permettono di creare la spina dorsale di un progetto lasciando al programmatore il compito di verificare e correggere solo in parte il codice perché indirizzati e condizionati da quanto proposto. In progetti complessi questo non riduce il ruolo del programmatore, anzi lo eleva a compiti più precisi e complessi lasciando la stesura di parti del codice semplici e ripetitive al software stesso. Sapere cosa chiedere e formulare correttamente le domande al LLM è fondamentale, esplicitando nel dettaglio con parole chiave mirate come deve essere realizzato il codice.

Altro compito complesso per il programmatore è non farsi troppo ammaliare dalle soluzioni proposte perché non sempre necessarie per quanto richiesto oppure diversa da quanto già conosciuto per realizzare una determinata funzione. Questo nuovo modo di lavorare per mette di conoscere nuove soluzioni ma comporta test e tempo non sempre disponibile. Il programmatore deve avere il controllo del progetto accettando generazione del codice automatica solo dove consapevole di quanto proposto e del suo impatto anche in casi di revisione e manutenzione futuri. L'ultimo miglio da percorrere è la personalizzazione delle risposte del LLM, per ottenere risposte coerenti con quanto già realizzato e conosciuto, per fare questo entra in gioco la RAG.

Chapter 2

Addestrare un LLM per la Generazione del Codice

L'addestramento di LLM per la generazione di codice di programmazione richiede una serie di passaggi metodici e risorse computazionali significative. Conoscere questo processo è utile per la successiva integrazione con la RAG. La procedura si divide nelle selle seguenti fasi:

2.1 Raccolta e Preparazione dei Dati

La qualità e la quantità dei dati per l'addestramento è di primaria importanza per prepare un modello alla generazione di codice in maniera efficace. È quindi essenziale utilizzare per il training codice sorgente proveniente da molteplici fonti tra cui codice sorgente, file Readme, documentazione tecnica, commenti nel codice, pagine Wiki, API e discussioni su forum specializzati in programmazione. In rete è possibile trovare diverso materiale open source tra cui dataset già etichettati. Alcuni dataset hanno un valore altissimo, per tutelare il costo per produrli per certi dataset è previsto il diritto d'autore. I dati si dividono in due tipologie:

- **Dati Strutturati**: seguono un formato specifico e predefinito.
- **Dati non Strutturati**: non sono organizzati e sono quindi più difficili da interpretare dal modello.

La raccolta di dati va visionata con cura, se non si conosce la provenienza del codice è possibile che contenga bug o codice opsoleto che possono essere trasmessi al modello. I dati raccolti devono essere quindi puliti e pre-processati per rimuovere errori e informazioni non pertinenti, garantendo così un dataset di alta qualità per l'addestramento. Sui dataset viene utilizzato un tokenizer specializzato che riconosce costrutti di programmazione come keyword, operatori e strutture sintattiche.

2.2 Pre-Addestramento

Il pre-addestramento di un LLM da utilizzare per la generazione di codice richiede un approccio specifico. A differenza del pre-addestramento generico, utilizzando i dataset precedentemente prepareti il modello impara a:

- Predire il completamento del codice
- Comprendere la struttura sintattica dei linguaggi di programmazione
- Riconoscere pattern comuni nel codice
- Identificare le relazioni tra diversi blocchi di codice

Un esempio pratico di pre-addestramento può essere implementato utilizzando la libreria transformers [WDS+20, FGT+20]:

```
from transformers import RobertaConfig, RobertaTokenizerFast

# Configurazione del modello per il codice
config = RobertaConfig(
    vocab_size=50000,  # Dimensione del vocabolario
    max_position_embeddings=514,  # Lunghezza massima sequenza
    num_attention_heads=12,  # Teste di attenzione
    num_hidden_layers=6,  # Strati nascosti
    type_vocab_size=1  # Tipo di vocabolario
)

# Tokenizer specializzato per il codice
tokenizer = RobertaTokenizerFast.from_pretrained(
    "microsoft/codebert-base",
    max_length=512,
    truncation=True,
```

```
    padding=True
)
```

Durante questa fase, il modello sviluppa una comprensione profonda della sintassi e della semantica del codice, che verrà poi raffinata durante il fine-tuning per compiti specifici di generazione del codice.

2.3 Fine-Tuning

Il fine-tuning è la fase in cui il modello viene specializzato per la generazione di codice, documentazione e risposta a quesiti specifici del contesto di programmazione. Durante questa fase, il modello affina le sue capacità attraverso:

- **Dataset Specializzati**: Utilizzo di dataset contenenti:
  - Coppie di descrizioni-implementazioni
  - Documentazione tecnica e commenti
  - Esempi di bug fixing e refactoring
- **Tecniche di Apprendimento**:
  - **Apprendimento Supervisionato**: Training su coppie input-output predefinite
  - **Apprendimento per Rinforzo**: Ottimizzazione basata su feedback e metriche di qualità
  - **Few-shot Learning**: Adattamento a nuovi contesti con pochi esempi

Un esempio pratico di fine-tuning può essere implementato utilizzando la libreria transformers [WDS+20]:

```
from transformers import Trainer, TrainingArguments
from datasets import load_dataset

# Caricamento del dataset per il fine-tuning
dataset = load_dataset("code_search_net", "python")

# Configurazione del training
training_args = TrainingArguments(
```

```
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=10,
    evaluation_strategy="epoch"
)

# Inizializzazione del trainer
trainer = Trainer(
    model=model,                 # Modello pre-addestrato
    args=training_args,          # Argomenti di training
    train_dataset=dataset["train"],
    eval_dataset=dataset["validation"],
    tokenizer=tokenizer,         # Tokenizer specializzato per il codice
)

# Avvio del fine-tuning
trainer.train()
```

Durante il fine-tuning, il modello sviluppa capacità specifiche come:

- Generazione di codice a partire da descrizioni in linguaggio naturale
- Completamento intelligente del codice basato sul contesto
- Creazione di documentazione tecnica
- Identificazione e correzione di bug
- Refactoring del codice seguendo best practices

Il processo di fine-tuning richiede un attento bilanciamento tra:

- **Overfitting**: Evitare che il modello memorizzi i dati di training
- **Generalizzazione**: Mantenere la capacità di adattarsi a nuovi contesti
- **Prestazioni**: Ottimizzare la velocità e la qualità delle risposte

2.4 Pre-Addestramento vs Fine-Tuning

È importante comprendere la distinzione tra queste due fasi dell'addestramento:
Pre-Addestramento
Il pre-addestramento è la fase iniziale dove il modello:

- Acquisisce una comprensione **generale** del linguaggio di programmazione
- Viene addestrato su **grandi quantità** di codice sorgente generico
- Impara le strutture base e la sintassi del linguaggio
- Non è ancora specializzato per compiti specifici

Fine-Tuning
Il fine-tuning è invece la fase di specializzazione dove il modello:

- Si adatta a un **dominio specifico** o a compiti particolari
- Utilizza dataset più piccoli ma **mirati**
- Affina le conoscenze per generare codice specifico per il tuo caso d'uso
- Viene ottimizzato per le esigenze specifiche del progetto

Analogia: Si può paragonare a:

- Pre-addestramento: Imparare la grammatica e il vocabolario di base di una lingua
- Fine-tuning: Specializzarsi nel linguaggio tecnico di un settore specifico

2.5 Architettura del Modello

Gli LLM utilizzano tipicamente architetture basate su trasformatori, che sono particolarmente efficaci nell'elaborazione di sequenze di dati, come il testo e il codice. I trasformatori utilizzano meccanismi di auto-attenzione per valutare l'importanza di diversi elementi in una sequenza, permettendo al modello di comprendere le relazioni a lungo raggio tra parole o token. Questa capacità è cruciale per la generazione di codice, dove le dipendenze tra variabili e funzioni possono estendersi su intere porzioni di codice.

2.6 Valutazione e Ottimizzazione

Una volta addestrato, il modello deve essere rigorosamente valutato utilizzando metriche specifiche per la generazione di codice, come la correttezza sintattica, la funzionalità e l'efficienza del codice prodotto. I risultati della valutazione guidano ulteriori ottimizzazioni, che possono includere aggiustamenti dei pesi del modello, modifiche all'architettura o l'inclusione di dati di addestramento aggiuntivi per affrontare eventuali carenze.

2.6.1 Metriche di Valutazione

- **Correttezza Sintattica**: Verifica che il codice generato sia sintatticamente corretto.
- **Funzionalità**: Verifica che il codice generato realizzi la funzionalità desiderata.
- **Efficienza**: Valuta le prestazioni del codice in termini di tempo di esecuzione e utilizzo delle risorse.

2.6.2 Tecniche di Ottimizzazione

- **Aggiustamento dei Pesi**: Modifica dei pesi del modello per migliorare le prestazioni.
- **Modifiche all'Architettura**: Introduzione di nuove componenti o modifiche a quelle esistenti.
- **Integrazione di Dati Aggiuntivi**: Utilizzo di ulteriori dati di addestramento per migliorare le prestazioni.

Chapter 3

RAG

3.1 Introduzione

Il RAG **Retrieval-Augmented Generation**, (in italiano *Generazione Aumentata tramite Recupero*) è un sistema che permette di migliorare l'output di un LLM estendendo la sua conoscenza con nuove informazioni, al di fuori dai suoi dati di addestramento. Allo scopo di:

- ottenere risposte personalizzate provenienti da librerie e codice custom;
- migliorare il codice generato rendendolo più specifico al dominio riducendo le allucinazioni;
- facilitare l'assistenza da parte del modello nella fase di debugging migliorando la sua comprensione di sistemi complessi;
- supportare la creazione di documentazione aggiornata;
- permettere all'interno di un Team di migliorare la coerenza del codice scritto da diversi programmatori;
- realizzare naturalmente senza forzature, codice più moderno proponendo librerie e standard comuni.
- evitare risposte imprecise a causa della confusione terminologica, in cui diverse fonti utilizzano la stessa terminologia per parlare di cose diverse.

3.2 Funzionamento

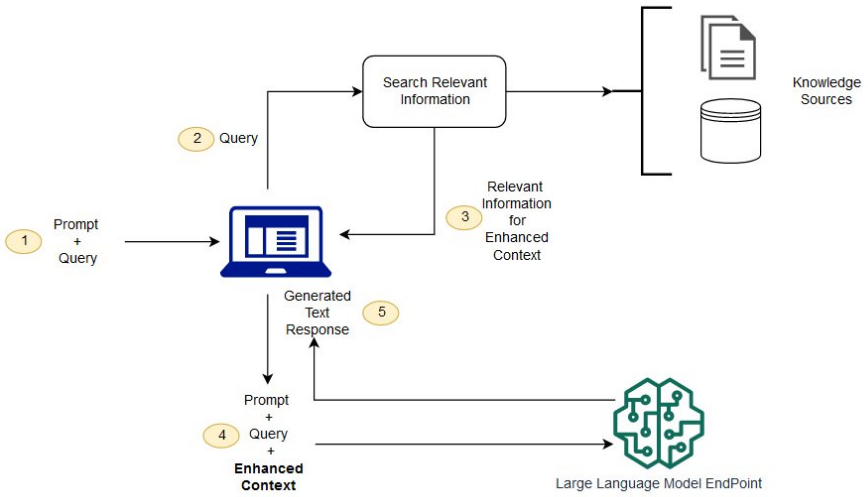


Figure 3.1: Flusso di una request ad un LLM integrato con un RAG

RAG è un sistema che integra il processo di generazione del linguaggio con un meccanismo di recupero delle informazioni. Come illustrato in fig. 3.1, il funzionamento si articola in quattro fasi principali:

3.2.1 Creazione di Dati Esterni

Il sistema RAG utilizza dati esterni al training set originale del LLM, provenienti da diverse fonti come:

- API e database
- Archivi documentali
- File di testo e codice

Questi dati vengono convertiti in rappresentazioni numeriche (embedding) e archiviati in un database vettoriale, creando una knowledge base accessibile al modello.

3.2.2 Recupero delle Informazioni

Quando l'utente sottopone una query:

- La domanda viene convertita in un vettore
- Il sistema cerca nel database vettoriale le informazioni più pertinenti
- Viene calcolata la rilevanza attraverso calcoli matematici vettoriali

3.2.3 Aumento del Prompt

Il sistema RAG arricchisce il prompt dell'utente:

- Aggiunge le informazioni recuperate al contesto
- Utilizza tecniche di prompt engineering per ottimizzare la comunicazione con il LLM
- Fornisce al modello un contesto arricchito per generare risposte più accurate

3.2.4 Gestione dell'Aggiornamento

Per mantenere l'efficacia del sistema nel tempo:

- Aggiornamento asincrono dei documenti
- Ricalcolo degli embedding per i nuovi dati
- Possibilità di aggiornamenti in tempo reale o batch

Questo approccio permette di superare le limitazioni dei LLM tradizionali, fornendo risposte più accurate e contestualizzate grazie all'integrazione di conoscenze esterne aggiornate.

Chapter 4

Implementazione di un Sistema RAG per lo Sviluppo del Software: Caso Studio

4.1 Panoramica del Progetto

In questo capitolo viene presentata un'implementazione pratica di un sistema RAG per sviluppare software all'interno dell'ecosistema specifico di una software-house. Il progetto dimostra come integrare un LLM con una base di conoscenza personalizzata per migliorare la generazione di codice.

4.2 Architettura del Sistema

4.3 Ambiente di Sviluppo

Il sistema è stato sviluppato utilizzando:

- Python 3.9+
- LangChain per l'orchestrazione RAG
- OpenAI API per il modello di linguaggio

```
# Requisiti del progetto
requirements = {
    "langchain": "0.0.300",
    "openai": "1.3.0",
    "python-dotenv": "1.0.0"
}
```

4.4 Implementazione

4.4.1 Preparazione dei Dati

nsloth permette di addestrare in maniera efficente il modello

```
from unsloth import FastLanguageModel
import torch
max_seq_length = 2048 # Choose any! We auto support RoPE Scaling internally!
dtype = None # None for auto detection. Float16 for Tesla T4, V100, Bfloat16 for Ampere+
load_in_4bit = False # Impostare a True per ridurre i pesi a 4bit quantizzandoli per ridurre uso di memoria, visto che il modello e' piccolo lascio Float16.

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/Llama-3.2-3B-Instruct", # or choose "unsloth/Llama-3.2-1B-Instruct"
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
    # token = "hf_...", # use one if using gated models like meta-llama/Llama-2-7b-hf
)
```

4.5 LORA

LoRA (Low-Rank Adaptation of Large Language Models) è una tecnica di allenamento popolare e leggera che riduce significativamente il numero di parametri allenabili. Funziona inserendo un numero inferiore di nuovi pesi nel modello e solo questi sono addestrati. Ciò rende l'allenamento con LoRA molto più veloce, efficiente in termini di memoria e produce pesi modello più piccoli (alcune centinaia di MB), che sono più facili da archiviare e condividere.

```
    model = FastLanguageModel.get_peft_model(
    model,
    r = 16, # Choose any number > 0 ! Suggested 8, 16, 32, 64, 128
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                      "gate_proj", "up_proj", "down_proj",],
    lora_alpha = 16,
    lora_dropout = 0, # Supports any, but = 0 is optimized
    bias = "none",    # Supports any, but = "none" is optimized
    # [NEW] "unsloth" uses 30% less VRAM, fits 2x larger batch sizes!
    use_gradient_checkpointing = "unsloth", # True or "unsloth" for very long
        context
    random_state = 3407,
    use_rslora = False,  # We support rank stabilized LoRA
    loftq_config = None, # And LoftQ
)
```

Usando SFTTrainer

4.6 FOrmatazione con i TAG di lamas

```
from unsloth.chat_templates import train_on_responses_only
trainer = train_on_responses_only(
    trainer,
    instruction_part = "<|start_header_id|>user<|end_header_id|>\n\n",
    response_part = "<|start_header_id|>assistant<|end_header_id|>\n\n",
)
```

4.6.1 Creazione del Database Vettoriale

[Mostra il tuo codice per la creazione degli embedding]

4.6.2 Integrazione con il LLM

[Mostra il tuo codice per l'integrazione con il modello]

4.6.3 Pipeline RAG

[Mostra il tuo codice per il processo RAG completo]

4.7 Test e Valutazione

4.7.1 Metodologia di Test

[Descrivi come hai testato il sistema]

4.7.2 Risultati

[Mostra i risultati ottenuti]

4.8 Conclusioni e Considerazioni

[Discuti i risultati e le possibili migliorie]

Chapter 5

Conclusioni

5.1 Risultati Ottenuti

In questa tesi abbiamo esplorato l'integrazione di RAG e LLM nello sviluppo del software, dimostrando come questi strumenti possano migliorare significativamente il processo di sviluppo. I risultati principali includono:

- Implementazione di un sistema RAG per la generazione di codice contestualizzato
- Analisi delle prestazioni e dell'efficacia del sistema
- Identificazione di potenziali aree di miglioramento

5.2 Impatto sullo Sviluppo Software

L'integrazione di strumenti basati su AI nel processo di sviluppo software sta rivoluzionando il settore. Durante il periodo di sviluppo di questa tesi (Ottobre 2024 - Gennaio 2025), abbiamo osservato:

- Rapida evoluzione degli strumenti di AI per lo sviluppo software
- Crescente disponibilità di soluzioni open source
- Miglioramento continuo nelle capacità di generazione e comprensione del codice

```java
public class HelloWorld {
  public static void main(String[] args) {
    // Prints "Hello, World" to the terminal window.
    System.out.println("Hello, World");
  }
}
```

5.3 Sfide e Considerazioni Future

Nonostante i progressi significativi, rimangono diverse sfide da affrontare:

- Bilanciamento tra automazione e controllo umano
- Gestione delle implicazioni etiche
- Necessità di mantenere competenze tecniche fondamentali

5.4 Prospettive Future

Le tecnologie RAG e LLM mostrano un potenziale significativo per:

- Accelerare lo sviluppo software
- Migliorare la qualità del codice
- Facilitare la documentazione e la manutenzione

La rapida evoluzione di questi strumenti suggerisce che siamo solo all'inizio di una trasformazione significativa nel modo in cui il software viene sviluppato e mantenuto.

Bibliography

[Doc24] Huggingface Docs. Lora, dec 2024.

[Doc25] GitHub Docs. Asking github copilot questions in your ide, jan 2025.

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[Git24] GitHub. Github copilot is more than a tool, it's an ally, dec 2024.

[Met24] Meta. Llama-3.3-70b-instruct, dec 2024.

[SBO23] Ahmed R. Sadik, Sebastian Brulin, and Markus Olhofer. Coding by design: Gpt-4 empowers agile model driven development, 2023.

[WDS+20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics, 2020.