Construção de um Modelo de Desenvolvimento/Geração de Código com IA

Gustavo Orlando Costa dos Santos Henriques - 64361
Estudo Orientado
Mestrado em Engenharia Informática
Faculdade de Ciências, Universidade de Lisboa
fc64361@fc.ul.pt

Resumo

Palavras-chave *Inteligência artificial; Grandes modelos de linguagem (LLMs); Engenharia de prompts; Geração automática de código; Engenharia de software*

1 Introdução

Outline. How is the rest of the document structured?
The remainder of this document is organised as follows. Section 2 presents bla bla bla. ...

2 Enquadramento Teórico

Esta secção serve como base teórica para clarificar alguns conceitos importantes, oferecendo uma melhor compreensão das secções que se seguem.

2.1 Geração automática de código

A geração automática de código consiste no processo de produzir componentes de software de forma parcial ou integral a partir de descrições de mais alto nível, como requisitos em linguagem natural, esquemas visuais, modelos formais ou exemplos estruturados. Em vez de o programador escrever manualmente o código fonte, recorrem-se a sistemas capazes de interpretar estas descrições e traduzir o seu conteúdo para implementações concretas, acelerando o desenvolvimento e reduzindo tarefas repetitivas.

Apesar da geração automática de código estar hoje em dia associada a modelos de inteligência artificial, o conceito não é própriamente recente. Já existiam abordagens como o Model-Driven Engineering (MDE), onde o código era gerado a partir de modelos formais, como por exemplo diagramas ou estruturas abstratas que descreviam o sistema e, linguagens específicas de domínio (DSLs), que permitiam escrever descrições declarativas que depois eram traduzidas automaticamente para código. Contudo, estas técnicas dependiam de regras de transformação rígidas e gramáticas estritamente definidas, o que as tornava eficazes apenas em domínios muito controlados e pouco adaptáveis a requisitos mais abertos ou expressos em linguagem natural.

Modelos de Linguagem

Os modelos de linguagem surgiram como uma evolução significativa na geração automática de código ao permitirem interpretar instruções menos estruturadas, como texto em linguagem natural. Estes modelos são treinados em grandes volumes de dados, incluindo código fonte, documentação e exemplos de uso, o que lhes permite aprender padrões sintáticos, estruturas típicas de implementação e relações entre diferentes elementos de um programa.

Uma parte importante desta evolução deve-se à arquitetura Transformer, introduzida em 2017 por Vaswani et al.[19], que se tornou a base dos modelos de linguagem modernos. Os Transformers utilizam mecanismos de atenção para identificar, dentro de uma sequência de texto, quais são as partes mais relevantes para cada palavra, permitindo compreender relações complexas e dependências a longo alcance. Esta arquitetura revelou-se altamente eficiente para tarefas de processamento de linguagem natural e tornou possível treinar modelos de grande escala capazes de compreender e gerar código com elevada coerência.

Ao longo desta evolução têm surgido cada vez mais sistemas que exploram o potencial dos modelos de linguagem para apoiar ou automatizar partes do processo de desenvolvimento de software. Alguns destes sistemas recorrem a um único modelo para interpretar descrições e gerar código de forma direta, enquanto outros combinam vários modelos ou etapas especializadas para orientar, validar ou complementar a geração. Esta diversidade reflete a maturidade crescente da área e o interesse em integrar modelos de linguagem não só como geradores de código, mas também como assistentes ativos no fluxo de trabalho do programador. Exemplos representativos destas abordagens são discutidos de forma detalhada na secção 3.1.

Desafios e Limitações

TODO

2.2 Engenharia de Prompts

Para entendermos este conceito, é primeiro necessário perceber a definição de prompt. Segundo Marvin et al.[13], um prompt é um input em formato textual (ou transcrição de outro meio, como áudio ou imagens) usado para orientar a saída de um modelo artificialmente inteligente, servindo para fornecer instruções e contexto, de modo a que o mesmo gere uma resposta em conformidade com a tarefa desejada.

Uma das influências na qualidade de resposta é a forma como são construídos estes prompts, visto que pequenas alterações nos mesmos são capazes de gerar respostas bastante diferentes[14]. Assim, por volta de 2022, emergiu esta nova disciplina no campo da inteligência artificial chamada prompt engineering, que tem como objetivo conceber prompts e otimizá-los, tornando o uso de LLMs mais eficiente[13]. Outra das razões para ter surgido o prompt engineering deve-se ao facto do custo de treino dos modelos. À medida que a complexidade dos modelos aumenta, os gastos em treino tornam-se cada vez maiores. Cottier et al.[6] estudo afirma que o custo associado aos modelos computacionalmente mais intensivos, tem crescido a uma taxa de 2.4 vezes ao ano desde 2016. Assim, ao recorrer à disciplina de engenharia de prompts, conseguimos afinar os LLMs sem gastar recursos para treinamento. Com isto, é notável a importância das mesmas, visto que conseguem melhorar a qualidade do output sem necessitar de um dispendioso processo de treino.

Desde 2022 têm vindo a ser desenvolvidas várias estratégias de prompt engineering. Estas estratégias têm sempre um propósito específico que querem melhorar, desde a melhoria do raciocínio e lógica do modelo, até à redução de alucinações[17]. Dependendo de cada estratégia utilizada, o prompt é construído de maneiras diferentes. No entanto, de acordo com Marvin et al.[13] existem boas práticas gerais para a elaboração de prompts efetivos, como definir o objetivo, compreender as capacidades do modelo e fornecer contexto.

3 Trabalho relacionado

A presente secção reúne o estado da arte relevante para esta tese, apresentando trabalhos, modelos e ferramentas que abordam problemas relacionados com a aplicação de métodos de inteligência artificial ao desenvolvimento de software, a utilização de técnicas de prompting e avaliação de modelos.

3.1 Geração Automática de Código

Nesta secção são analisados trabalhos que ilustram várias abordagens para automatizar a transformação de descrições diretamente em código.

Hoje em dia já existem vários modelos generativos com a finalidade de desenvolvimento de código, ou seja, modelos que foram construídos com o objetivo principal de gerar como resposta um excerto de código que cumpre os requisitos solicitados pelo ser humano. Codex[5] foi um dos primeiros modelos amplamente conhecidos para transformar descrições textuais em código executável, suportando múltiplas linguagens e tarefas como geração de funções, completamento e tradução entre linguagens. Este modelo serviu ainda de base ao GitHub Copilot, que será abordado mais adiante, estabelecendo um marco na utilização prática de LLMs para auxiliar o desenvolvimento de software.

Dando seguimento a esta ideia, surgiu o CodeLlama[16], uma família de modelos orientados para tarefas de programação como geração, explicação e preenchimento de código. Estes modelos, que foram desenvolvidos com base no modelo Llama2[18], são disponibilizados em três variantes distintas especificadas abaixo e na figura1:

Code Llama, a versão base, implementada para tarefas mais gerais de compreensão e transformação de código;

Code Llama - Python, uma versão treinada adicionalmente com grandes bases de dados de código python, fornecendo um auxílio com mais qualidade em requisitos para esta linguagem em específico;

Code Llama - Instruct, uma variante ajustada para seguir melhor as instruções humanas, oferecendo uma interação conversacional mais aprimorada.

Treinados para processar sintaxe estruturada e múltiplos paradigmas de programação, os modelos CodeLlama representam uma alternativa moderna e de acesso aberto (open-source), ideal para cenários que exigem a conversão de requisitos em linguagem natural diretamente para código.

Adicionalmente, temos também o StarCoder[9] e a sua evolução mais recente, StarCoder2[11], que representam modelos treinados em larga escala no dataset The Stack (The Stack v2 para o modelo StarCoder2), destacando-se pelo suporte multilinguagem, capacidades de preenchimento, janelas de contexto alargadas, permitindo, com este último ponto, analisar blocos mais extensos de código numa única passagem, e mecanismos de treino orientados para segurança e transparência. Graças a estas características, tornam-se referências úteis para compreender boas práticas na construção de LLMs orientados para código, tanto ao nível da arquitetura e dos datasets, como das técnicas necessárias para desenvolver sistemas capazes de converter descrições em código de forma fiável e coerente.

Para além dos modelos dedicados à geração de código, têm surgido ferramentas práticas de assistência ao programador integradas diretamente nas IDEs (ambientes de desenvolvimento integrado). Um exemplo marcante é o GitHub Copilot[1], inicialmente alimentado pelo modelo Codex, uma variante do GPT-3 especializada em código. Com a evolução da ferramenta, o Copilot passou também a incorporar modelos mais avançados, como o GPT-4 e GPT-4o. Hoje em dia já inclui modos autónomos como o modo de agente e o agente de programação, capazes de executar ações dentro da IDE ou desenvolver tarefas completas de forma assistida. Ziegler et al.[24] estudaram empiricamente a ferramenta analisando o comportamento da mesma e o seu impacto no processo de

[1] https://github.com/features/copilot

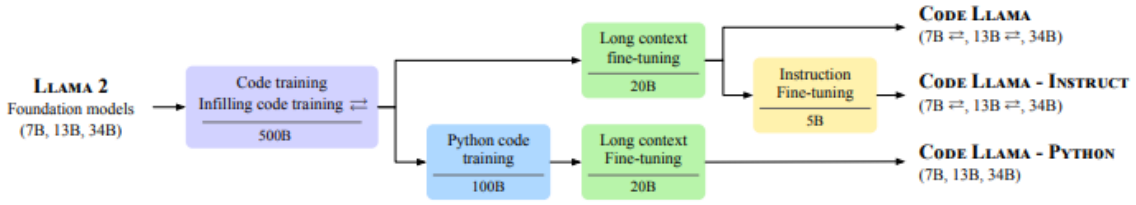

Figura 1. Pipeline de especialização do Code Llama[16].

desenvolvimento, evidenciando como a completamento incremental pode acelerar tarefas de rotina e reduzir o esforço cognitivo do programador.

De forma semelhante, a Amazon apresentou o CodeWhisperer[2], um assistente de programação concebido para gerar sugestões de código contextualizadas em múltiplas linguagens. Yetistiren et al.[22] compararam o CodeWhisperer com outras ferramentas, incluindo o Copilot e o ChatGPT, avaliando a qualidade, correção e segurança das sugestões produzidas. Tal como o Copilot, o CodeWhisperer opera principalmente como um mecanismo de completamento inteligente dentro da IDE, contribuindo para automatizar partes do fluxo de escrita de código e acelerar tarefas repetitivas.

Estas ferramentas demonstram que, para além de abordagens orientadas à geração integral de componentes de software, existe já uma adoção consolidada de LLMs em cenários de completamento de código no desenvolvimento real.

Nos últimos tempos, surgiram também ferramentas que vão de encontro ao conceito de Spec-Driven Development (SDD), onde, em vez de se implementar o código primeiro e documentá-lo posteriormente, começa-se por definir as especificações que irão servir de guia para os agentes de IA. Entre estas destaca-se o Spec-Kit[3], desenvolvido pela equipa de IA do GitHub, uma ferramenta que fornece uma interface de linha de comandos para criar e organizar especificações, planos e tarefas capazes de guiar modelos generativos ao longo de um fluxo de desenvolvimento. De forma complementar, o Kiro[4], desenvolvido pela Kiro Labs, apresenta-se como um ambiente integrado que combina edição de especificações com agentes capazes de gerar e atualizar código com base nesses ficheiros estruturados. Embora ainda recentes, ambas as ferramentas ilustram uma nova tendência no desenvolvimento de software onde se criam fluxos automáticos que partem de especificações formais para produzir componentes coerentes.

Além destas ferramentas, existe já investigação académica que explora abordagens semelhantes. Patil et al.[15] apresentam o spec2code, um framework que combina modelos de linguagem com verificadores formais para gerar código C a partir de especificações estruturadas. Existe também investigação centrada na geração automática das próprias especificações formais. Um exemplo recente é o SpecGen, proposto por Ma et al.[12], que utiliza modelos de linguagem para produzir contratos formais, a partir de código-fonte ou descrições textuais. O sistema gera especificações em linguagens como JML[2] e avalia a sua consistência e verificabilidade utilizando verificadores formais. O SpecGen demonstra o papel crescente dos LLMs na automatização de etapas tradicionalmente manuais.

Para além das abordagens baseadas em especificações, surgem também trabalhos que seguem caminhos distintos na geração automática de código. Embora não se integrem no mesmo enquadramento teórico, constituem contribuições relevantes por explorarem problemas mais específicos, como a transformação de interfaces visuais em estruturas front-end. Xu et al.[21] propõem um sistema que converte imagens de interfaces web em código HTML e CSS. O método assenta num fluxo composto por três etapas principais: geração de um dataset com imagens e respetivo código, deteção dos componentes visuais da interface recorrendo a modelos como CNN[1] e Faster R-CNN[3] e, produção do código através de uma arquitetura que combina uma CNN para extração visual com uma LSTM[4] responsável por gerar a sequência de código. Este trabalho demonstra uma abordagem interessante para a transformação do design em estruturas front-end executáveis, alinhando-se com a componente da tese dedicada à geração automática de código para a interface.

A figura2 ilustra a progressão dos sistemas analisados, apresentando uma linha temporal que sintetiza o seu aparecimento. Contudo, há que ter em conta que atualmente existe um ecossistema muito vasto de ferramentas e modelos generativos orientados a programação que vai muito para além do que foi mencionado nesta secção. A sua evolução tem sido contínua e acelerada, com contribuições de várias organizações como OpenAI, Meta, Google, AWS e a comunidade de código aberto.

3.2 Engenharia de Prompts

Como já foi referido na secção 2.2, a forma como as instruções são fornecidas a um modelo de linguagem tem um impacto direto na qualidade do código que o mesmo consegue gerar.

[2]https://aws.amazon.com/pt/q/developer/
[3]https://github.com/github/spec-kit
[4]https://kiro.dev

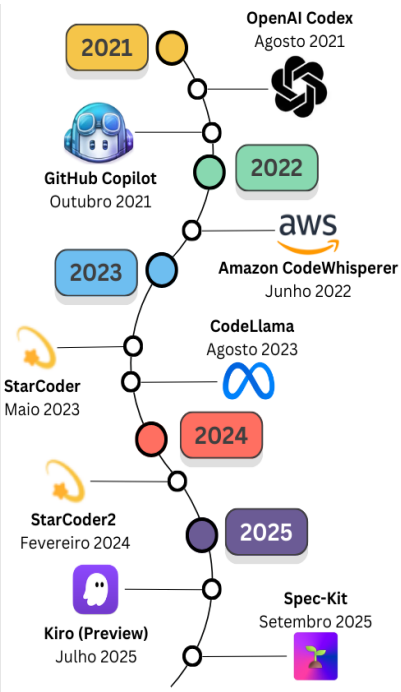

Figura 2. Linha temporal dos modelos e ferramentas de geração de código analisados nesta secção.

Diversos trabalhos mostram que técnicas de engenharia de prompts podem melhorar significativamente o desempenho em tarefas de linguagem natural para código. Esta secção apresenta estudos que aplicam estas técnicas em cenários reais de geração de código, evidenciando a sua importância prática no desenvolvimento de sistemas baseados em LLMs.

Um dos estudos no contexto da engenharia de prompts aplicada à geração de código é o trabalho de Liu et al.[10], que investiga a forma como diferentes estratégias de prompting influenciam o desempenho do ChatGPT em tarefas de geração de código . Os autores avaliam o modelo no benchmark CodeXGlue e demonstram que a formulação do prompt tem um impacto substancial na qualidade do código gerado. O estudo aplica técnicas como chain-of-thought, instruções comportamentais e otimizações multi-etapa, mostrando, através de métricas como o BLEU e CodeBLEU, que o desempenho do ChatGPT pode melhorar significativamente.

Adicionalmente Wang et al.[20], afirmam que, apesar dos avanços recentes nos LLMs, continua a ser difícil garantir que estes produzam código correto e robusto de forma consistente. Para mitigar este problema, os autores propõem o PET-Select, um método que seleciona automaticamente a técnica de prompting mais adequada com base na complexidade do problema. Avaliado em benchmarks como MBPP e HumanEval, o PET-Select mostrou melhorias moderadas na qualidade do código e reduções significativas no custo de geração, demonstrando que a escolha informada da estratégia de prompting pode ter impacto real no desempenho de modelos orientados à programação.

Um outro contributo é apresentado por Khojah et al.[8], que exploraram o impacto de diferentes técnicas de prompt engineering na geração de código. Os autores introduzem o CodePromptEval, um dataset composto por 7072 prompts concebido para avaliar cinco técnicas distintas (few-shot, persona, chain-of-thought, function signature e list of packages) aplicadas à geração de funções completas em 3 modelos, o GPT-4o, Llama 3 e Mistral. O estudo mostra que algumas destas técnicas influenciam de forma significativa a correção e qualidade do código produzido. Contudo, a combinação de várias técnicas não garante necessariamente melhorias adicionais.

Para além das técnicas de prompting aplicadas diretamente ao enunciado da tarefa, existe uma linha de investigação que explora estratégias de closed-loop prompting, onde o modelo é instruído a analisar e refinar o próprio código após a primeira geração. Ding et al.[7] apresentam o CYCLE, um método que combina geração inicial com iterações sucessivas de auto-refinamento orientadas por feedback, mostrando melhorias na qualidade final do código. De forma complementar, Zhou et al.[23] propõem o RefineCoder, que incorpora um mecanismo adaptativo de crítica e correção para permitir que o LLM identifique e ajuste erros no código que produz. Ambos os trabalhos reforçam que a melhoria iterativa guiada pelo próprio modelo pode ser uma alternativa eficaz às abordagens que dependem exclusivamente do prompt inicial, contribuindo para aumentar a fiabilidade do código gerado.

4 «Other Section(s) as Appropriate»

The report should include one or more sections providing a detailed description of the problem you are addressing in the project and your plan to tackle it. Use appropriate section titles for what is presented.

You should explain the methods you are planning to use, or have already started to apply, in your project. This discussion should be grounded in the related work, your own understanding of the problem, and, when available, preliminary results.

In case you already have some preliminary results, consider to include a section devoted to them. This section should describe the work already carried out, what data has already been collected, what analysis and designs have already been done, what methods have been used, what programs and/or preliminary results already exist, etc.

5 Forthcoming Work and Conclusions

This section should include subsections describing the work to be carried out during the remainder of the school year and its objectives. It should also present a chronological plan for the completion of the project. Finally, include a

concluding subsection that summarizes the contributions already made, provides a preliminary self-assessment of the progress achieved so far, and discusses the main difficulties encountered.

Referências

[1] [n. d.]. Convolutional Neural Networks for Visual Recognition site. https://cs231n.github.io/convolutional-networks/. Accessed: 2025-11-22.

[2] [n. d.]. Java Modeling Language (JML) site. https://www.cs.ucf.edu/~leavens/JML/index.shtml. Accessed: 2025-11-22.

[3] [n. d.]. Understanding and Implementing Faster R-CNN site. https://medium.com/@RobuRishabh/understanding-and-implementing-faster-r-cnn-248f7b25ff96. Accessed: 2025-11-22.

[4] [n. d.]. Understanding LSTMs site. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 2025-11-22.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[6] Ben Cottier, Robi Rahman, Loredana Fattorini, Nestor Maslej, and David Owen. 2024. The rising costs of training frontier AI models. *CoRR* abs/2405.21015 (2024). arXiv:2405.21015 doi:10.48550/ARXIV.2405.21015

[7] Yangruibo Ding, Marcus J. Min, Gail E. Kaiser, and Baishakhi Ray. 2024. CYCLE: Learning to Self-Refine the Code Generation. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 392–418. doi:10.1145/3649825

[8] Ranim Khojah, Francisco Gomes de Oliveira Neto, Mazen Mohamad, and Philipp Leitner. 2025. The Impact of Prompt Programming on Function-Level Code Generation. *IEEE Trans. Software Eng.* 51, 8 (2025), 2381–2395. doi:10.1109/TSE.2025.3587794

[9] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *Trans. Mach. Learn. Res.* 2023 (2023). https://openreview.net/forum?id=KoFOg41haE

[10] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving ChatGPT Prompt for Code Generation. *CoRR* abs/2305.08360 (2023). arXiv:2305.08360 doi:10.48550/ARXIV.2305.08360

[11] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian J. McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, and et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *CoRR* abs/2402.19173 (2024). arXiv:2402.19173 doi:10.48550/ARXIV.2402.19173

[12] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 16–28. doi:10.1109/ICSE55347.2025.00129

[13] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2023. Prompt engineering in large language models. In *International conference on data intelligence and cognitive informatics*. Springer, 387–402.

[14] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2149–2160. doi:10.1109/ICSE48619.2023.00181

[15] Minal Suresh Patil, Gustav Ung, and Mattias Nyberg. 2024. Towards Specification-Driven LLM-Based Generation of Embedded Automotive Software. In *Bridging the Gap Between AI and Reality - Second International Conference, AISoLA 2024, Crete, Greece, October 30 - November 3, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 15217)*, Bernhard Steffen (Ed.). Springer, 125–144. doi:10.1007/978-3-031-75434-0_9

[16] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). arXiv:2308.12950 doi:10.48550/ARXIV.2308.12950

[17] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. *CoRR* abs/2402.07927 (2024). arXiv:2402.07927 doi:10.48550/ARXIV.2402.07927

[18] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian,

Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). arXiv:2307.09288 doi:10.48550/ARXIV.2307.09288

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[20] Chung-Yu Wang, Alireza DaghighFarsoodeh, and Hung Viet Pham. 2024. Selection of Prompt Engineering Techniques for Code Generation through Predicting Code Complexity. *CoRR* abs/2409.16416 (2024). arXiv:2409.16416 doi:10.48550/ARXIV.2409.16416

[21] Yong Xu, Lili Bo, Xiaobing Sun, Bin Li, Jing Jiang, and Wei Zhou. 2021. image2emmet: Automatic code generation from web user interface image. *J. Softw. Evol. Process.* 33, 8 (2021). doi:10.1002/SMR.2369

[22] Burak Yetistiren, Isik Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *CoRR* abs/2304.10778 (2023). arXiv:2304.10778 doi:10.48550/ARXIV.2304.10778

[23] Changzhi Zhou, Xinyu Zhang, Dandan Song, Xiancai Chen, Wanli Gu, Huipeng Ma, Yuhang Tian, Mengdi Zhang, and Linmei Hu. 2025. RefineCoder: Iterative Improving of Large Language Models via Adaptive Critique Refinement for Code Generation. arXiv:2502.09183 [cs.CL] https://arxiv.org/abs/2502.09183

[24] Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, X. Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. 2022. Productivity Assessment of Neural Code Completion. *CoRR* abs/2205.06537 (2022). arXiv:2205.06537 doi:10.48550/ARXIV.2205.06537