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

Os avanços recentes na inteligência artificial generativa têm permitido automatizar partes do desenvolvimento de software. No entanto, a adoção destas tecnologias em contextos empresariais exige a sua adaptação a práticas internas e fluxos de trabalho já estabelecidos. Esta dissertação, desenvolvida em colaboração com a Trust Systems, tem como objetivo analisar, selecionar e combinar tecnologias de inteligência artificial, para construir uma *pipeline* que automatize etapas relevantes do processo de desenvolvimento de software da empresa. O estudo inclui a utilização de técnicas de *prompt engineering* que assegurem a qualidade e coerência dos artefactos gerados.

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

1 Introdução

A inteligência artificial generativa tem vindo a transformar várias etapas do processo de desenvolvimento de software, permitindo automatizar tarefas que tradicionalmente exigiam significativa intervenção humana[13]. Modelos de linguagem de grande escala tornaram-se capazes de interpretar instruções e gerar código a partir dessas descrições[9], bem como produzir explicações, documentação e outros artefactos associados ao ciclo de vida do software[4].
Apesar do crescimento destas tecnologias, o desafio é conseguir adaptar as mesmas de forma eficaz ao processo concreto de desenvolvimento utilizado em contextos empresariais. O problema abordado nesta dissertação consiste, portanto, em aplicar, adaptar e combinar tecnologias de inteligência artificial, de forma a automatizar etapas do processo de desenvolvimento de software da Trust Systems, garantindo alinhamento com os seus métodos e constrangimentos operacionais.

Motivação

A automatização de partes do desenvolvimento de software pode trazer benefícios significativos para a empresa, nomeadamente a redução de esforço manual[19], aceleração da prototipagem e maior capacidade de resposta a requisitos em evolução. No contexto da Trust Systems, onde a manipulação de dados sensíveis impõe restrições adicionais ao uso de plataformas externas, torna-se particularmente relevante compreender como integrar soluções baseadas em modelos de linguagem de forma controlada, segura e compatível com o processo atual. Esta investigação procura, assim, apoiar a adoção responsável e eficiente destas tecnologias no ambiente real da empresa.

Objetivo

O objetivo geral desta dissertação é desenvolver e avaliar uma *pipeline* modular que utilize tecnologias de inteligência artificial, nomeadamente modelos de linguagem, para automatizar etapas relevantes do processo de desenvolvimento de software da Trust Systems. Os objetivos específicos incluem:

- Avaliar tecnologias existentes de IA e selecionar as mais adequadas ao contexto interno;
- Definir e implementar uma *pipeline* baseada em *prompt engineering*;
- Gerar artefactos relevantes, como especificações, design e código;
- Avaliar a qualidade das saídas produzidas e comparar diferentes estratégias de prompting.

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.

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, recorre-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 inteligência artificial generativa, o conceito não é propriamente recente. Já existiam abordagens como o Model-Driven Engineering (MDE), onde o código é gerado a partir de modelos formais, como por exemplo diagramas ou estruturas abstratas, conforme descrito por Schmidt[26] e Brambilla et al.[3] ou linguagens específicas de domínio (DSLs), que permitem escrever descrições declarativas que depois são traduzidas automaticamente para código, tal como discutido por Mernik et al.[20]. Contudo, estas técnicas dependem de regras de transformação rígidas e gramáticas estritamente definidas, o que as torna eficazes apenas em domínios muito controlados e pouco adaptáveis a requisitos mais abertos ou expressos em linguagem natural.

Os modelos de IA generativa proporcionaram uma evolução significativa na geração automática de código ao permitirem interpretar instruções menos estruturadas, como texto em linguagem natural, conforme discutido por Chen et al.[9]. Esta capacidade resulta do facto de estes modelos serem treinados em grandes volumes de dados heterogéneos, que incluem código fonte, documentação técnica, exemplos de implementação e descrições presentes em fóruns de programação. A exposição simultânea a linguagem natural e a linguagens de programação permite que os modelos aprendam padrões sintáticos, estruturas típicas de implementação e relações entre diferentes componentes de um programa.
Uma parte importante desta evolução deve-se à arquitetura *Transformer*, proposta em 2017 por Vaswani et al.[28]. Os *Transformer* utilizam mecanismos de atenção que identificam, dentro de uma sequência de texto, as partes mais relevantes para cada token, capturando dependências longas e relações complexas entre elementos da instrução. Esta arquitetura revelou-se altamente eficiente para tarefas de processamento de linguagem natural e tornou possível, nos anos seguintes, treinar modelos de grande escala capazes de compreender instruções complexas e gerar código coerente e funcional[9].
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.

Ciclo de Vida do Desenvolvimento de Software

Embora os avanços nos LLMs demonstrem o seu potencial para a geração de código, a construção de um produto de software envolve um conjunto mais amplo de fases. Para contextualizar o impacto e o papel que a automatização pode assumir, é necessário compreender o ciclo de vida completo do desenvolvimento de software[27], identificando as fases onde é possível introduzir automatização. Antes de existir código, é necessário definir o que deve ser construído e como o sistema deve funcionar. Após a implementação, existem ainda outros passos a cumprir para que se garanta a qualidade e a continuidade do produto.

Estas etapas incluem, em primeiro lugar, a elicitação e análise de requisitos, que visam garantir que as necessidades do cliente sejam adequadamente compreendidas, estruturadas e documentadas. O processo de elicitação de requisitos envolve diversas técnicas, incluindo entrevistas e revisões literárias, como evidenciado por Lim et al.[15], que destacam a importância de métodos eficazes para alcançar uma compreensão abrangente das necessidades dos utilizadores, uma vez que os requisitos devidamente capturados constituem a base sobre a qual todo o ciclo de vida do software é construído.
Com os requisitos definidos, a modelação funcional e estrutural do sistema surge como etapa subsequente, frequentemente suportada por linguagens formais de representação como a UML (Unified Modeling Language). A UML permite traduzir requisitos textuais em representações visuais, facilitando a análise, validação e comunicação entre equipas técnicas e não técnicas. Berenbach[2] destaca que os modelos UML desempenham um papel crítico na documentação e validação de requisitos, servindo como elementos centrais de coordenação entre diferentes perspetivas do sistema. Esta modelação inclui diagramas de casos de uso, classes, atividades ou sequência, que complementam a descrição funcional do sistema e ajudam a antecipar o comportamento desejado. A conceção de interfaces e fluxos de interação beneficia igualmente destas representações estruturadas.
A definição de APIs (Application Programming Interfaces) e dos componentes arquiteturais constitui outra etapa fundamental, pois estabelece os contratos de comunicação entre as diferentes partes do sistema. Especificações claras de APIs permitem uma implementação mais consistente, facilitam a integração entre módulos e suportam estratégias de teste orientadas a componentes e serviços.
Depois da definição dos componentes e das interfaces do sistema, segue-se a fase de implementação, onde o código é desenvolvido de acordo com as especificações definidas. Esta etapa envolve não apenas a programação propriamente dita, mas também a integração entre módulos e a resolução de dependências técnicas, garantindo que os componentes funcionem de forma coesa. A implementação é acompanhada por atividades de verificação e validação, onde se aplicam

diferentes níveis de testes, com o objetivo de identificar defeitos, avaliar comportamentos e assegurar que o software cumpre os requisitos estabelecidos. Por fim, a documentação técnica desempenha um papel transversal a todas estas fases, registando decisões, especificações, arquiteturas, APIs e procedimentos de utilização, facilitando a manutenção futura e promovendo uma compreensão consistente do sistema entre programadores e stakeholders.

Para além das fases que compõem o desenvolvimento de software, importa considerar também as metodologias que organizam estas atividades ao longo do ciclo de vida. Os processos tradicionais, como o modelo em cascata (Waterfall)[23], estruturavam o desenvolvimento de forma sequencial, com transições rígidas entre etapas. Com o tempo, esta abordagem revelou limitações em cenários de requisitos dinâmicos, particularmente em contextos de mudança frequente ou incerteza elevada, conduzindo à adoção de metodologias ágeis, que promovem ciclos iterativos, entregas incrementais e adaptação contínua às necessidades do utilizador[12].

Limitações dos Modelos de Linguagem e Técnicas de Refinamento

Apesar dos avanços significativos alcançados pelos modelos de linguagem de grande escala, estes sistemas apresentam limitações que afetam diretamente a sua aplicação no desenvolvimento de software. Entre os desafios mais comuns encontram-se a geração de código incorreto ou incompleto, inconsistências lógicas, alucinações factuais e dificuldades em interpretar requisitos ambíguos ou instruções pouco estruturadas. Estas limitações resultam, em grande parte, da natureza estatística dos modelos, da variabilidade dos dados de treino e da ausência de mecanismos internos de verificação semântica.

Para mitigar estas limitações, diversas técnicas de refinamento têm sido exploradas. Uma abordagem consiste no treino adicional (fine-tuning) sobre dados específicos de um domínio, permitindo ajustar o comportamento do modelo às necessidades de uma organização ou tarefa concreta. No entanto, esta estratégias implica custos elevados, sobretudo quando envolvem retrainar modelos de larga escala. Cottier et al.[5] afirma que o custo associado aos modelos computacionalmente mais intensivos, tem crescido a uma taxa de 2.4 vezes ao ano desde 2016.

Neste contexto, a engenharia de prompts (prompt engineering) tornou-se uma alternativa particularmente relevante. Estas técnicas procuram otimizar as instruções fornecidas ao modelo, estruturando-as de modo a orientar o comportamento do LLM sem necessidade de treino adicional. Estudos recentes têm demonstrado que a forma como o prompt é construído influencia significativamente a qualidade do código gerado, a clareza das explicações e a capacidade de o modelo seguir passos complexos[19]. Ao estabelecer padrões, estratégias e boas práticas de formulação de prompts[18], é possível alcançar resultados mais fiáveis e reduzir ambiguidades inerentes à interpretação do modelo. Para além destas boas práticas gerais, a literatura identifica ainda um conjunto variado de técnicas específicas, cada uma concebida para atingir objetivos distintos, esde a melhoria do raciocínio e lógica do modelo, até à redução de alucinações, conforme sistematizado por Sahoo et al.[25].

Dado que esta dissertação visa explorar a automatização de diferentes etapas do desenvolvimento de software, a engenharia de prompts assume um papel central enquanto mecanismo de controlo e refinamento do comportamento dos modelos utilizados. O baixo custo operacional e a ausência de necessidade de treino especializado tornam esta abordagem particularmente adequada ao contexto estudado, permitindo adaptar modelos generalistas às tarefas específicas que compõem o pipeline de geração de software.

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.

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[9], uma variante do GPT-3[11] especializada em código, 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[24], 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[8], são disponibilizados em três variantes distintas especificadas abaixo e na Figura 1:

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

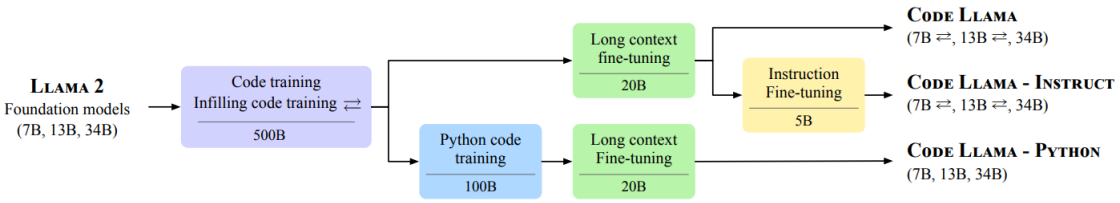

Figura 1. Pipeline de especialização do Code Llama[24].

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[10] e a sua evolução mais recente, StarCoder2[7], 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. 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.[32] estudaram empiricamente a ferramenta analisando o comportamento da mesma e o seu impacto no processo de 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.[30] 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.

A crescente integração destas ferramentas diretamente nas IDEs mostra que os LLMs estão a ser efetivamente adotados no desenvolvimento de software do dia a dia, sobretudo em cenários de completamento de código. Esta adoção prática tem contribuído para acelerar tarefas rotineiras e aumentar a produtividade dos programadores[21].

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. Estas especificações assumem-se como a fonte central de orientação do processo, podendo incluir listas de requisitos, componentes, tarefas e decisões tecnológicas que, em abordagens tradicionais, estariam distribuídas por diversos artefactos de análise e planeamento. Assim, uma parte significativa das atividades que antecedem a implementação, é formalizada diretamente nestes ficheiros. 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

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

[2] https://aws.amazon.com/pt/q/developer/

[3] https://github.com/github/spec-kit

[4] https://kiro.dev

se criam fluxos automáticos que partem de especificações estruturadas para produzir componentes coerentes.
Além destas ferramentas, têm surgido também propostas que abordam este problema recorrendo a especificações formais. Patil et al.[22] 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.[17], 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[1] e avalia a sua consistência e verificabilidade utilizando verificadores formais. O SpecGen demonstra o papel crescente dos LLMs na automatização de etapas tradicionalmente manuais.

A Figura 2 ilustra a cronologia 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.

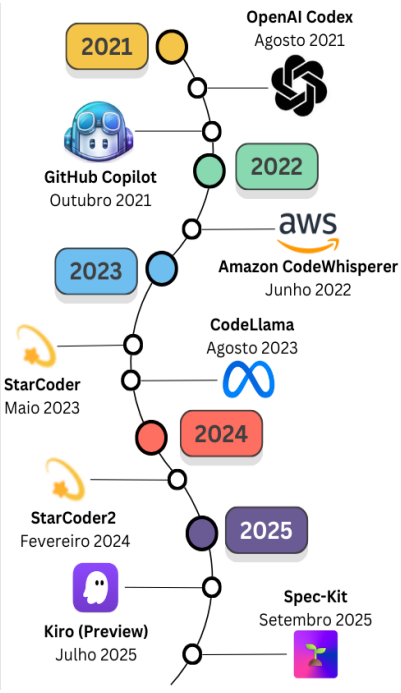

Figura 2. Linha temporal dos modelos e ferramentas de geração de código analisados nesta secção.

Engenharia de Prompts

Como já foi referido na secção 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. 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.[16], 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, um conjunto de tarefas padronizadas para programação que inclui geração, tradução, refatoração e completamento de código, 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 que o desempenho do ChatGPT pode melhorar significativamente. Esta conclusão foi alcançada através de métricas como o BLEU que avalia a semelhança entre o código gerado e a solução de referência através da contagem de n-gramas, isto é, sequências contíguas de n tokens que permitem medir a sobreposição entre as duas versões de código e o CodeBLEU, que amplia a abordagem anterior ao incluir informação específica de programas, analisando não apenas n-gramas mas também a estrutura sintática representada por árvores AST e as relações de fluxo de dados.

Adicionalmente Wang et al.[29], 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.[14], 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.[6] 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.[31] 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 Processo de Desenvolvimento de Software da Trust Systems

5 Pipeline Proposta

6 Planeamento do Trabalho

O trabalho a desenvolver na fase seguinte da dissertação organiza-se em três grandes etapas.

A primeira etapa consiste no desenvolvimento do protótipo. Nesta fase será implementada a *pipeline* proposta, definindo, integrando e articulando os diferentes módulos que serão necessários para a geração de artefactos das diferentes etapas de desenvolvimento de software. Esta fase culminará numa primeira versão funcional da solução.

Segue-se uma etapa de avaliação e testes, durante a qual serão analisados os resultados produzidos pela *pipeline*. Serão comparadas diferentes estratégias de prompting e avaliados diversos modelos de linguagem. Esta fase inclui também o refinamento e melhoria da solução sempre que necessário, garantindo maior coerência entre os artefactos gerados e maior robustez no comportamento da pipeline.

Por fim, terá lugar a redação e revisão do documento final, que incorpora a consolidação dos resultados obtidos, a descrição detalhada da metodologia seguida e a discussão das limitações e contributos do trabalho.

A Tabela 1 apresenta o cronograma das atividades previstas, permitindo visualizar a sequência das fases do trabalho e o tempo alocado a cada etapa do projeto.

Descrição das atividades	Jan 26	Fev 26	Mar 26	Abr 26	Mai 26	Jun 26
Desenvolvimento do protótipo	■	■	■	■		
Avaliação e testes				■	■	
Análise e ajuste do protótipo					■	
Redação do documento final				■	■	■
Revisão final da tese e preparação da defesa						■

Tabela 1. Cronograma de atividades

Referências

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

[2] Brian Berenbach. 2004. Comparison of UML and text based requirements engineering. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 247–252. doi:10.1145/1028664.1028766

[3] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers. doi:10.2200/S00441ED1V01Y201208SWE001

[4] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *CoRR* abs/2303.12712 (2023). arXiv:2303.12712 doi:10.48550/ARXIV.2303.12712

[5] 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

[6] 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

[7] Anton Lozhkov 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

[8] Hugo Touvron et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). arXiv:2307.09288 doi:10.48550/ARXIV.2307.09288

[9] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[10] Raymond Li et al. 2023. StarCoder: may the source be with you! *Trans. Mach. Learn. Res.* 2023 (2023). https://openreview.net/forum?id=KoFOg41haE

[11] Tom B. Brown et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[12] Marta Fernández-Diego, Erwin R. Méndez, Fernando González-Ladrón-de-Guevara, Silvia Abrahão, and Emilio Insfrán. 2020. An Update on Effort Estimation in Agile Software Development: A Systematic

Literature Review. *IEEE Access* 8 (2020), 166768–166800. doi:10.1109/ACCESS.2020.3021664

[13] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8 (2024), 220:1–220:79. doi:10.1145/3695988

[14] 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

[15] Sachiko Lim, Aron Henriksson, and Jelena Zdravkovic. 2021. Data-Driven Requirements Elicitation: A Systematic Literature Review. *SN Comput. Sci.* 2, 1 (2021), 16. doi:10.1007/S42979-020-00416-4

[16] 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

[17] 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

[18] 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.

[19] 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

[20] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4 (2005), 316–344. doi:10.1145/1118890.1118892

[21] Nettur, Suresh Babu, Karpurapu, Shanthi, Nettur, Unnati, Gajja, Likhit Sagar, Myneni, Sravanthy, Dusi, and Akhil. 2025. The Role of GitHub Copilot on Software Development: A Perspective on Productivity, Security, Best Practices and Future Directions. *arXiv preprint arXiv:2502.13199* (2025).

[22] 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

[23] W. W. Royce. 1987. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings, 9th International Conference on Software Engineering, Monterey, California, USA, March 30 - April 2, 1987*, William E. Riddle, Robert M. Balzer, and Kouichi Kishida (Eds.). ACM Press, 328–339. http://dl.acm.org/citation.cfm?id=41801

[24] 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

[25] 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

[26] Douglas C. Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. *Computer* 39, 2 (2006), 25–31. doi:10.1109/MC.2006.58

[27] Ian Sommerville. 2015. *Software Engineering* (10 ed.). Pearson.

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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