Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

A IA Generativa na Engenharia de Software

Um estudo de caso

Cássio Azevedo Cancio

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Paulo Roberto Miranda Meirelles
Cossupervisor: Arthur Pilone Maia da Silva
Cossupervisor: Carlos Eduardo Santos

Aos meus pais, que sempre incentivaram meus estudos.
Aos meus professores, que tornaram este trabalho possível.

Resumo

Cássio Azevedo Cancio. **A IA Generativa na Engenharia de Software: *Um estudo de caso***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.

Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave). Deve ser precedido da referência do documento. Texto texto. Texto texto.

Palavras-chave: Palavra-chave1. Palavra-chave2. Palavra-chave3.

Abstract

Cássio Azevedo Cancio. **Generative AI in Software Engineering: *A case study***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo.

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa. De acordo com o Regimento da Pós-Graduação da USP (Artigo 99), deve ser redigido em inglês para fins de divulgação. É uma boa ideia usar o sítio www.grammarly.com na preparação de textos em inglês. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text.

Keywords: Keyword1. Keyword2. Keyword3.

Lista de abreviaturas

IA	Inteligência Artificial (*Artificial Intelligence*)
IME	Instituto de Matemática e Estatística
LLM	Modelo de Linguagem de Grande Escala (*Large Language Model*)
USP	Universidade de São Paulo

Lista de figuras

Lista de tabelas

Lista de programas

Sumário

Introdução

Contexto

A engenharia de *software* é um campo da computação que surgiu e se desenvolveu através da crescente demanda da sociedade do fim do século XX até a atualidade por sistemas computacionais cada vez mais complexos. Neste contexto, ferramentas, métodos e processos foram criados para possibilitar o atendimento dessa demanda.

Nas últimas décadas, os estudos em inteligência artificial (IA) avançaram rapidamente, de modo que um novo paradigma em IA surgiu, a IA generativa. Diferentemente da IA tradicional, a IA generativa pode criar conteúdos novos e originais baseados no que aprendeu, em vez de apenas copiar, imitar e reproduzir algo que já existe.

Dada a flexibilidade e a abertura de diversas possibilidades com essa nova tecnologia, é natural que uma de suas aplicações fosse a engenharia de *software*. Nos últimos anos, diversos estudos foram publicados de modo a analisar essas aplicações, suas consequências e propor diferentes abordagens para tais aplicações. (Johnson e Menzies, 2024 e Terragni *et al.*, 2025)

Desta forma, este projeto se propõe a levantar os resultados observados em diversos artigos que tratam do estudo do impacto da IA generativa na engenharia de *software*. Além de realizar um estudo de caso sobre a aplicação da IA generativa ao longo das fases do desenvolvimento de *software*, avaliando sua utilidade, limitações e impacto na qualidade e produtividade.

Como estudo de caso, foi realizado o desenvolvimento de uma aplicação web na área de investimentos, com banco de dados, *backend* em *Java Spring Boot* e *frontend* em *Angular*, utilizando ferramentas de IA generativa nas diferentes fases do desenvolvimento de um sistema. As fases analisadas foram: coleta e análise de requisitos, estudo de viabilidade, *design* de *software*, codificação e testes.

Motivação

Este trabalho se faz relevante no contexto em que o uso de ferramentas de IA generativa vem crescendo com o passar dos anos, desde o surgimento de ferramentas como *ChatGPT* e *GitHub Copilot*. Segundo dados da *Stack Overflow 2024 Developer Survey* (Stack Overflow, 2024), 63,2% dos desenvolvedores profissionais já utilizam ferramentas de IA no seu processo de desenvolvimento, enquanto que 13.5% desse mesmo grupo planeja utilizá-las em

breve. Além disso, entre os desenvolvedores que responderam usar inteligência artificial, 82% a utiliza para escrever código.

Desta maneira, é evidente que uma nova tecnologia com amplo uso no mercado de *software* e que abre possibilidade para diversas aplicações, terá impactos sobre como os desenvolvedores escrevem seus códigos. Assim, é de suma importância buscar avaliar e compreender melhor de que maneira esses impactos vêm ocorrendo nas bases de código, inclusive através de um estudo de caso.

Objetivos

Os principais objetivos do trabalho são:

- Compreender de que maneira a IA generativa tem impactado na produção de código das empresas de *software*, através de um levantamento de dados disponíveis em outros artigos;
- Desenvolver um sistema com todo seu processo voltado ao uso de ferramentas de IA generativa ao longo das suas diferentes fases;
- Documentar os resultados gerados pela IA durante o processo de desenvolvimento do sistema, incluindo os *prompts* utilizados;
- Analisar os resultados obtidos, a fim de mensurar a qualidade das respostas geradas.

Capítulo 1

Referencial Teórico

1.1 Engenharia de Software

Segundo a definição do *Institute of Electrical and Electronics Engineers* (IEEE) (1990), a engenharia de *software* é a aplicação de uma sistemática, disciplinada e quantificável abordagem para o desenvolvimento, operação e manutenção de um *software*. Para que a engenharia de *software* seja viável, dada a complexidade dos sistemas demandados na atualidade, foram desenvolvidas etapas, metodologias e ferramentas que dessem suporte aos atores envolvidos no projeto, como desenvolvedores, analistas, investidores, clientes, entre outros.

1.1.1 Etapas do Desenvolvimento de Software

O Ciclo de Vida de Desenvolvimento de Software (SDLC) consiste numa sequência de processos pelos quais o desenvolvimento de um *software* ocorre, de modo a produzir um resultado eficaz e de alta qualidade. Existe alguma variação no número de passos descritos por diferentes fontes, mas, em geral, há sete fases essenciais: planejamento, análise de requisitos, *design*, codificação, testes, implantação e manutenção.

Planejamento

A fase inicial envolve definir o propósito e o escopo do *software*. Durante esta etapa, a equipe de desenvolvimento deve levantar as tarefas necessárias, elaborar estratégias para cumpri-las e colaborar de modo a compreender as necessidades dos usuários finais. Neste processo, os objetivos do *software* e qual problema ele se propõe a resolver precisam ficar claros a todos os envolvidos.

Além disso, nesta fase também ocorre o estudo de viabilidade, ou seja, desenvolvedores e outros atores do projeto avaliam desafios técnicos e financeiros que possam impactar a evolução ou o sucesso do *software*. Ao fim desta fase, um plano de projeto é criado, com o intuito de detalhar as funções do sistema, os recursos necessários, possíveis riscos e o cronograma do projeto. Ao definir papéis, responsabilidades e expectativas claras, o planejamento estabelece uma base sólida para um processo eficiente de desenvolvimento.

Análise de Requisitos

A segunda fase do SDLC visa identificar e registrar os requisitos dos usuários finais. Nesta etapa, a equipe de projeto realiza o levantamento dos requisitos, por meio da coleta de informações das partes interessadas, como analistas, usuários e clientes. São empregadas técnicas como entrevistas, pesquisas e grupos de foco para compreender as necessidades e expectativas dos usuários.

Após a coleta, os dados são analisados, diferenciando os requisitos essenciais dos desejáveis. Essa análise possibilita a definição das funcionalidades, desempenho, segurança e interfaces do *software*. Neste momento, são definidos os requisitos funcionais e não funcionais. Os requisitos funcionais especificam as funções que o *software* deve realizar, ou seja, o que o sistema deve fazer. Já os requisitos não funcionais tratam de como o sistema deve se comportar, incluindo aspectos como desempenho, segurança, usabilidade e escalabilidade.

O resultado desse processo é o Documento de Especificação de Requisitos (DER), que descreve o propósito, as funcionalidades e características do *software*, servindo como guia para a equipe de desenvolvimento e fornecendo estimativas de custo, quando necessário. O êxito desta fase é crucial para o sucesso do projeto, pois assegura que a solução desenvolvida atenda às expectativas dos usuários.

Design

A fase de *design* é responsável pela definição da estrutura do *software*, abrangendo sua funcionalidade e aparência. A equipe de desenvolvimento detalha a arquitetura do sistema, a navegação, as interfaces de usuário e a modelagem do banco de dados, assegurando que o *software* tenha boa usabilidade e seja eficiente.

Entre as atividades desta fase, destaca-se a elaboração de diagramas de fluxo de dados, de entidade-relacionamento, de classes, protótipos de interface e diagramas arquiteturais. O objetivo é garantir que as estruturas projetadas sejam suficientes para dar suporte a todas as funcionalidades do sistema. Também são identificadas dependências, pontos de integração e eventuais restrições, como limitações do equipamento físico e requisitos de desempenho.

O resultado desta fase é o Documento de *Design* de Software (DDS) que estrutura formalmente as informações do projeto e trata preocupações de *design*. Neste documento, são adicionados os artefatos produzidos, servindo como guia estável para coordenar equipes grandes e garantir que todos os componentes do sistema funcionem de maneira integrada.

Codificação

Na fase de codificação, os engenheiros e desenvolvedores transformam o *design* do *software* em código executável. O objetivo é produzir um *software* funcional, eficiente e com boa usabilidade. Para isso, utilizam-se linguagens de programação adequadas, seguindo o DDS e diretrizes de codificação estabelecidas pela organização e pela legislação local.

Durante esta fase, são realizadas revisões de código, nas quais os membros da equipe examinam o trabalho uns dos outros para identificar erros ou inconsistências, garantindo

elevados padrões de qualidade. Além disso, testes preliminares internos são conduzidos para garantir que as funcionalidades básicas do sistema foram atendidas.

Ao final da fase de codificação, o *software* passa a existir como um produto funcional, representando a materialização dos esforços das etapas anteriores, mesmo que ainda sejam necessários refinamentos e ajustes subsequentes. O resultado desta fase é o código-fonte.

Testes

A fase de testes consiste em verificar a qualidade e a confiabilidade do *software* antes de sua entrega aos usuários finais. Seu objetivo é identificar falhas, erros e vulnerabilidades, assegurando que o sistema atenda aos requisitos especificados.

Inicialmente, são definidos parâmetros de teste alinhados aos requisitos do *software* e casos de teste que contemplem diferentes cenários de uso. Em seguida, são conduzidos testes de diversos níveis e tipos, incluindo testes de unidade, de integração, de sistema, de segurança e de aceitação, permitindo a avaliação tanto de componentes individuais quanto da operação do sistema na sua totalidade.

Quando um erro é identificado, ele é registrado detalhadamente, incluindo seu comportamento, métodos de reprodução e impacto sobre o sistema. As falhas são encaminhadas para correção e o *software* retorna à fase de testes para validação. Este ciclo de teste e correção se repete até que o sistema esteja conforme os critérios previamente estabelecidos. O resultado desta fase é um código-fonte mais robusto e menos propenso a falhas.

Implantação

A fase de implantação ou *deployment* consiste em disponibilizar o *software* aos usuários finais, garantindo sua operacionalidade no ambiente de produção. Este processo ocorre tanto no primeiro lançamento do sistema, quanto quando ele já está em uso pelos usuários e passando por atualizações. Por isso, o processo deve minimizar interrupções e impactos negativos nos acessos dos usuários.

A escolha da estratégia de implantação é feita conforme as características do sistema e de seus usuários. As estratégias mais comuns são:

- *Rolling*: a atualização ocorre de forma gradual, substituindo instâncias antigas por novas até que todo o sistema esteja atualizado;
- *Blue-Green*: dois ambientes paralelos são mantidos, um em produção e outro em preparação, permitindo a troca imediata entre eles;
- *Canary*: a nova versão é liberada primeiramente para um grupo de usuários, monitorando o comportamento do sistema antes de expandir a implantação para todos.

Além de colocar o *software* em operação, esta fase envolve assegurar que os usuários compreendam seu funcionamento. Para isso, podem ser fornecidos manuais, treinamentos e suporte técnico. Desta maneira, a fase de implantação marca a transição do *software* de projeto para produto, iniciando efetivamente o cumprimento de seus objetivos e a entrega de valor ao usuário.

Manutenção

A fase de manutenção é caracterizada por suporte contínuo e por melhorias incrementais, de modo a garantir que o *software* mantenha seu funcionamento adequado, acompanhe as necessidades dos usuários e as demandas de mercado. Nesta fase, são realizadas atualizações, correções de falhas e suporte ao usuário.

Considerando o horizonte de longo prazo, a manutenção inclui estratégias de modernização ou substituição do *software*, buscando manter sua relevância e adequação às evoluções tecnológicas.

1.1.2 Metodologias de Desenvolvimento

As metodologias de desenvolvimento de *software* consistem em abordagens sistemáticas para organizar, planejar e executar projetos de engenharia de *software*. Cada metodologia apresenta vantagens e limitações, sendo a escolha dependente do tamanho do projeto, grau de complexidade, maturidade da equipe e expectativa de mudanças nos requisitos.

Método de Cascata

A metodologia cascata é uma abordagem sequencial em que cada fase do desenvolvimento de *software* deve ser concluída antes de iniciar a seguinte. Ela é adequada para projetos com requisitos muito definidos e pouca probabilidade de mudanças durante o desenvolvimento. O modelo enfatiza documentação detalhada e planejamento prévio, garantindo controle rígido sobre prazos e entregas. Embora simples de aplicar, pode se mostrar inflexível diante de alterações nos requisitos ou no ambiente de negócios.

Metodologias Ágeis

As metodologias ágeis consistem em práticas iterativas e incrementais, voltadas à entrega contínua de valor ao usuário e à adaptação rápida às mudanças nos requisitos. As abordagens ágeis valorizam colaboração, flexibilidade e melhoria contínua, sendo especialmente adequada para projetos dinâmicos e de alta complexidade.

Além disso, elas são geralmente complementadas por práticas de Integração Contínua e Entrega Contínua (CI/CD), que automatizam a construção, teste e implantação do *software*. Desta maneira, a utilização de CI/CD permite que novas funcionalidades e correções sejam disponibilizadas rapidamente, mantendo a qualidade do sistema e reduzindo o tempo de reposta entre a equipe de desenvolvimento e os usuários.

Alguns exemplos de métodos ágeis incluem:

- **Extreme Programming (XP)**: enfatiza práticas de programação colaborativa, como *pair programming*, integração contínua, testes automatizados e feedback constante do cliente.
- **Lean Software Development**: baseado nos princípios de eliminação de desperdício, entrega rápida e melhoria contínua, priorizando valor para o cliente.
- *Scrum*: método iterativo que organiza o trabalho em ciclos curtos, denominados *sprints* (iterações), com duração de uma a quatro semanas. Define papéis específicos,

como o responsável pelo produto (*product owner*), o facilitador do processo (*scrum master*) e a equipe de desenvolvimento. Inclui ainda reuniões regulares, como o planejamento da iteração (*sprint planning*), reuniões diárias de acompanhamento (*daily*), a revisão da iteração e a retrospectiva, que garantem planejamento, monitoramento e melhoria contínua do processo.

- *Kanban*: método visual de gestão do fluxo de trabalho, baseado na utilização de quadros divididos em colunas que representam etapas do processo (por exemplo, "A Fazer", "Em Progresso" e "Concluído"). Cada atividade é representada por um cartão que se movimenta conforme avança nas etapas, promovendo transparência, fluxo contínuo e foco na identificação e eliminação de gargalos.
- Programação extrema (*extreme programming* — XP): metodologia ágil que enfatiza práticas de desenvolvimento colaborativo e de qualidade, como programação em par, integração contínua, desenvolvimento orientado a testes e refatoração frequente. Valoriza o envolvimento próximo do cliente, a simplicidade do código e a adaptação rápida às mudanças de requisitos.
- Desenvolvimento enxuto (*lean software development*): abordagem inspirada nos princípios da produção enxuta, que busca eliminar desperdícios, reduzir custos e acelerar entregas. Essa metodologia prioriza a criação de valor para o cliente, a melhoria contínua e a capacitação das equipes para tomadas de decisão mais eficientes.

1.1.3 Ferramentas de Desenvolvimento

As ferramentas de desenvolvimento de software oferecem suporte às etapas do ciclo de vida, desde planejamento até manutenção. Elas incluem:

- **Sistemas de Controle de Versão** (como Git e SVN): permitem gerenciar alterações no código-fonte, possibilitando colaboração entre desenvolvedores e rastreamento de histórico.
- **Ambientes de Desenvolvimento Integrados (IDEs)** (como Visual Studio, IntelliJ IDEA e Eclipse): fornecem recursos como edição de código, depuração, testes e integração com sistemas de controle de versão.
- **Ferramentas de Gerenciamento de Projeto** (como Jira, Trello e Asana): auxiliam na organização de tarefas, acompanhamento de progresso e comunicação entre equipes.
- **Ferramentas de Integração e Entrega Contínua (CI/CD)** (como Jenkins, GitHub Actions e GitLab CI): automatizam testes, builds e implantações, promovendo qualidade e agilidade.
- **Ferramentas de Teste** (como Selenium, JUnit e Postman): permitem a realização de testes automatizados e manuais para validar funcionalidades, desempenho e segurança do software.

O uso combinado dessas ferramentas contribui para maior produtividade, qualidade e confiabilidade no desenvolvimento de software.

1.2 Inteligência Artificial

1.2.1 Conceitos Básicos

Inteligência Artificial (IA) é o campo da ciência da computação que se dedica a criar sistemas capazes de executar tarefas que normalmente exigem inteligência humana, como reconhecimento de padrões, tomada de decisão e aprendizado a partir de dados. A IA pode ser classificada em:

- **IA Fraca (Narrow AI)**: sistemas projetados para executar tarefas específicas, sem consciência ou compreensão geral do mundo.
- **IA Forte (General AI)**: hipotéticos sistemas capazes de realizar qualquer tarefa cognitiva que um humano consegue executar.
- **IA Baseada em Aprendizado de Máquina**: sistemas que melhoram seu desempenho por meio da análise de grandes volumes de dados e ajuste de parâmetros internos.

1.2.2 Redes Neurais

Redes neurais artificiais são modelos computacionais inspirados na estrutura do cérebro humano. Elas são compostas por camadas de nós (neurônios artificiais) interconectados, capazes de processar informações e aprender padrões a partir de exemplos. Redes neurais são amplamente utilizadas em tarefas como classificação, reconhecimento de imagens e processamento de linguagem natural.

1.2.3 Aprendizado de Máquina

Aprendizado de Máquina (Machine Learning) é um subcampo da IA que se concentra em criar algoritmos capazes de aprender e fazer previsões a partir de dados, sem serem explicitamente programados para cada tarefa. Os principais tipos de aprendizado incluem:

- **Supervisionado**: o algoritmo aprende a partir de exemplos rotulados, tentando prever saídas corretas para novas entradas.
- **Não Supervisionado**: o algoritmo identifica padrões ou estruturas em dados não rotulados.
- **Por Reforço**: o algoritmo aprende tomando ações em um ambiente e recebendo recompensas ou penalidades, buscando maximizar o desempenho ao longo do tempo.

1.3 IA Generativa

1.3.1 Modelos de Linguagem

Modelos de linguagem são algoritmos capazes de compreender, gerar e manipular texto em linguagem natural. Eles aprendem padrões estatísticos e semânticos a partir de

grandes corpora de texto e são a base para sistemas de tradução automática, assistentes virtuais e chatbots avançados.

1.3.2 Transformers

Transformers são uma arquitetura de redes neurais introduzida em 2017, projetada para lidar eficientemente com sequências de dados, como texto. Utilizam mecanismos de atenção que permitem processar relações entre palavras de forma paralela, superando limitações de modelos recorrentes tradicionais.

1.3.3 LLMs (*Large Language Models*)

Large Language Models (LLMs) são modelos de linguagem treinados com enormes volumes de dados textuais, contendo bilhões de parâmetros. Eles são capazes de gerar texto coerente, resumir informações, responder perguntas e realizar tarefas complexas de processamento de linguagem natural. Exemplos incluem GPT, BERT e T5.

Capítulo 2

Metodologia

2.1 Abordagem de Pesquisa

2.1.1 Tipo de Pesquisa

2.1.2 Procedimentos Metodológicos

2.2 Coleta de Dados

2.2.1 Fontes de Dados

2.2.2 Instrumentos de Coleta

2.2.3 Processo de Coleta

2.3 Análise de Dados

2.3.1 Métodos de Análise

2.3.2 Ferramentas Utilizadas

2.3.3 Critérios de Avaliação

Capítulo 3

Resultados

3.1 Análise dos Dados

3.2 Avaliação do Sistema

3.2.1 Desempenho

3.2.2 Eficiência

3.2.3 Usabilidade

3.3 Discussão

3.3.1 Limitações Identificadas

3.3.2 Melhorias Propostas

Capítulo 4

Conclusão

4.1 Resumo dos Resultados

4.1.1 Principais Descobertas

4.1.2 Objetivos Alcançados

4.1.3 Contribuições

4.2 Trabalhos Futuros

4.2.1 Direções de Pesquisa

4.2.2 Melhorias Propostas

4.2.3 Desafios Identificados

4.3 Considerações Finais

Referências

["IEEE Standard Glossary of Software Engineering Terminology" 1990] "Ieee standard glossary of software engineering terminology". *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064 (citado na pg. 3).

[Johnson e Menzies 2024] Brittany Johnson e Tim Menzies. " AI Over-Hype: A Dangerous Threat (and How to Fix It) ". *IEEE Software* 41.06 (nov. de 2024), pp. 131–138. ISSN: 1937-4194. DOI: 10.1109/MS.2024.3439138. URL: https://doi.ieeecomputersociety.org/10.1109/MS.2024.3439138 (citado na pg. 1).

[Stack Overflow 2024] Stack Overflow. *Stack Overflow Developer Survey 2024*. 2024. URL: https://survey.stackoverflow.co/2024/ (acesso em 19/08/2025) (citado na pg. 1).

[Terragni *et al.* 2025] Valerio Terragni, Annie Vella, Partha Roop e Kelly Blincoe. "The future of ai-driven software engineering". *ACM Trans. Softw. Eng. Methodol.* 34.5 (mai. de 2025). ISSN: 1049-331X. DOI: 10.1145/3715003. URL: https://doi.org/10.1145/3715003 (citado na pg. 1).

Índice remissivo