

RENAN MARCOS FERREIRA

**Uso de modelos de linguagem de grande escala para decisões
arquiteturais de desenvolvimento de software**

São Paulo
2024

RENAN MARCOS FERREIRA

**Uso de modelos de linguagem de grande escala para decisões
arquiteturais de desenvolvimento de software**

Versão Original

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Engenharia de Software.

Área de Concentração: Engenharia de Software

Orientador: Prof. Dr. Fábio Levy Siqueira

São Paulo
2024

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

Catalogação-na-publicação

Ferreira, Renan

Uso de modelos de linguagem de grande escala para decisões arquiteturais de desenvolvimento de software / R. Ferreira -- São Paulo, 2024.
66 p.

Monografia (MBA em Engenharia de Software) - Escola Politécnica da Universidade de São Paulo. PECE – Programa de Educação Continuada em Engenharia.

1.TECNOLOGIA DA INFORMAÇÃO 2.ENGRENHARIA DE SOFTWARE
3.ARQUITETURA DE SOFTWARE 4.INTELIGÊNCIA ARTIFICIAL
I.Universidade de São Paulo. Escola Politécnica. PECE – Programa de Educação Continuada em Engenharia II.t.

Nome: FERREIRA, Renan

Título: Uso de modelos de linguagem de grande escala para decisões arquiteturais de desenvolvimento de software

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Engenharia de Software.

Aprovado em:

Banca Examinadora

Prof.(a). Dr.(a). _____

Instituição: _____

Julgamento: _____

Prof.(a). Dr.(a). _____

Instituição: _____

Julgamento: _____

Prof.(a). Dr.(a). _____

Instituição: _____

Julgamento: _____

AGRADECIMENTOS

Agradeço a Amanda Dantas Ferreira, minha querida e amada esposa, por todo o apoio em diversas etapas de minha vida e pela motivação que sempre me traz.

Agradeço aos meus pais José Lazilote Ferreira e Maria Aparecida Ferreira, por apoiarem meus estudos em computação e dedicarem parte de suas vidas me ensinando.

Agradeço aos professores do PECE-USP pela dedicação no ensino e todo o conhecimento compartilhado, especialmente ao Prof. Dr. Fábio Levy Siqueira por me orientar e apoiar durante a redação deste trabalho.

Aos amigos e colegas de trabalho e a todas as pessoas que, direta ou indiretamente, me apoiaram durante o período de estudos no PECE-USP, especialmente aos meus amigos Arthur Machado, Fernando Godoy, Luan Sales e Pedro Vieira.

RESUMO

FERREIRA, Renan. Uso de modelos de linguagem de grande escala para decisões arquiteturais de desenvolvimento de software. 2024. 66 p. Monografia (MBA em Engenharia de Software). Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo. São Paulo. 2024.

Avaliar decisões arquiteturais não é um processo simples durante o desenvolvimento de um software, pois exige a consideração de múltiplos *trade-offs*. Este trabalho aborda o uso de Modelos de Linguagem de Grande Escala (*LLMs*) para apoiar esse processo, com foco na extração de decisões arquiteturais a partir de requisitos textuais, atributos de qualidade e modelos de domínio. Foi realizado um estudo de caso em um sistema real de gestão de inadimplência no setor de *fintechs*, comparando as soluções geradas pelos *LLMs* com a arquitetura efetivamente implementada, seguindo o método proposto por Eisenreich, Speth e Wagner (2024). Para avaliação da qualidade de decisões e dos *trade-offs* envolvidos, utilizou-se uma versão adaptada do método *ATAM*.

Os resultados mostram que o modelo GPT-4o1 gerou decisões arquiteturais plausíveis e alinhadas a práticas comuns no desenvolvimento de software, com poucas alucinações sobre padrões ou estratégias inexistentes. Além disso, o LLM identificou *trade-offs* relevantes e estruturou atributos de qualidade de forma similar à solução real. A análise evidencia que os *LLMs* não apenas aceleram o processo de tomada de decisões, mas também oferecem novas ideias durante o processo.

Embora os resultados sejam promissores, a eficácia das respostas depende de uma boa estruturação dos *prompts* e de intervenções humanas para assegurar a assertividade, evidenciando a necessidade de métodos mais maduros e ferramentas que facilitem sua adoção em ambientes corporativos e acadêmicos.

Palavras-chave: Arquitetura de Software. IA. *LLM*. *ATAM*. *ADRs*. Gestão do conhecimento arquitetural.

ABSTRACT

FERREIRA, Renan. **Use of Large Language Models for architectural decisions in software development.** 2024. 66 p. Monografia (MBA em Engenharia de Software). Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo. São Paulo. 2024.

Evaluating architectural decisions is not a straightforward process in software development, as it demands the consideration of multiple trade-offs. This study investigates the use of Large Language Models (LLMs) to support this process, focusing on the extraction of architectural decisions from textual requirements, quality attributes, and domain models. A case study was conducted on a real-world delinquency management system in the fintech sector, comparing the solutions generated by LLMs with the effectively implemented architecture, using the method proposed by Eisenreich, Speth, and Wagner (2024). To assess decision quality and the trade-offs involved, an adapted version of the ATAM method was applied.

The results indicate that the GPT-4o1 model generated plausible architectural decisions aligned with common software development practices, demonstrating minimal hallucinations regarding non-existent patterns or strategies. Furthermore, the LLM identified relevant trade-offs and structured quality attributes in a manner similar to the actual solution. The analysis shows that LLMs not only help the decision-making process but also offer new architectural insights.

Although the findings are promising, the effectiveness of the responses depends on well-structured prompts and human intervention to ensure accuracy, highlighting the need for more mature methods and tools to facilitate their adoption in both corporate and academic environments.

Keywords: Software Architecture, AI, LLM, ATAM, ADRs, Architectural Knowledge Management.

LISTA DE ILUSTRAÇÕES

Figura 1 - Diagrama do modelo de domínio dos requisitos funcionais	34
Figura 2 - Prompt criado para o LLM	39
Figura 3 - Primeira resposta do LLM ao prompt inicial enviado	42
Figura 4 - Segunda resposta do LLM para o modelo de domínio	43
Figura 5 - Processo de Chain of Thought interno do modelo de LLM para fornecer a segunda resposta.....	44

LISTA DE TABELAS

Tabela 1 - Requisitos funcionais textuais do sistema de inadimplência.....	31
Tabela 2 - Atributos de qualidade do sistema de inadimplência.....	33
Tabela 3 - Decisões arquiteturais do sistema de inadimplência	36
Tabela 4 - Relação entre quais decisões arquiteturais atendem à quais atributos de qualidade no sistema de inadimplência.....	38
Tabela 5 - Resposta final do LLM com as decisões arquiteturais.....	45
Tabela 6 - Resposta final do LLM com as decisões arquiteturais (continuação)	46

LISTA DE ABREVIATURAS E SIGLAS

ATAM	Architecture Trade-off Analysis Method
LLM	Large Language Model
IA	Inteligência Artificial
GPT	Generative Pre-trained Transformer
NLP	Natural Language Processing
LLaMA	Large Language Model Meta AI
CQRS	Command Query Responsibility Segregation
ADR	Architecture Decision Records
DDD	Domain-Driven Design
MobSTr	Model-Based Safety Assurance and Traceability
FR	Functional Requirement
NFR	Non-Functional Requirement
I/O	Input/Output
UML	Unified Modeling Language
ACID	Atomicity, Consistency, Isolation, Durability
Pub/Sub	Publisher-Subscriber Pattern
CSV	Comma-separated Values
DSL	Domain-Specific Language

SUMÁRIO

1	INTRODUÇÃO.....	12
1.1	Objetivo	13
1.2	Metodologia.....	14
1.3	Justificativa.....	15
1.4	Estrutura do trabalho	16
2	MODELOS DE LINGUAGEM DE GRANDE ESCALA.....	17
3	ARQUITETURA DE SOFTWARE.....	20
3.1	Estilos e padrões arquiteturais de software	21
3.2	Método de avaliação de candidatos à Arquitetura de Software	24
3.3	Gestão do conhecimento arquitetural de software.....	25
3.4	Geração semiautomática de candidatos à Arquitetura de Software.....	26
4	CONTEXTO DO PROBLEMA E SOLUÇÃO REAL	30
4.1	Requisitos funcionais.....	31
4.2	Atributos de qualidade	32
4.3	Modelo de domínio	34
4.4	Solução real implementada	36
4.5	Instruções para o modelo de linguagem de grande escala	38
5	RESULTADOS OBTIDOS DO MODELO DE LINGUAGEM DE GRANDE ESCALA ..	41
5.1	Discussão.....	47
6	CONSIDERAÇÕES FINAIS.....	50
	REFERÊNCIAS	52
	APÊNDICE A – PROMPT E SAÍDA COMPLETA DO LLM	53

1 INTRODUÇÃO

Os modelos de linguagem de grande escala (LLM, do inglês *large language models*) vem democratizando o uso de IA em diferentes áreas do conhecimento, dado que as pessoas não precisam conhecer de maneira profunda sobre aprendizado de máquina ou estatística aplicada para extrair resultados. Isso facilitou tarefas envolvendo o processamento de linguagem natural como resumo de textos, reescrita de textos envolvendo mudanças nos tons de conversas, estilos e gramática, além de uma construção de sua própria base de conhecimento com diferentes fontes para apoio em tomada de decisões.

Na área do conhecimento da Engenharia de Software, os *LLMs* têm se destacado como ferramentas essenciais para diversos processos, além dos já mencionados, incluindo a geração e revisão de código, pesquisa e criação de testes, entre outros. A interação com esses modelos pode ser realizada de várias maneiras, desde prompts em modelos genéricos, como o *GPT* (*Generative Pre-Trained Transformer*), até o uso de modelos específicos treinados para lidar com atividades da Engenharia de Software, como o *GitHub Copilot*. Este último opera dentro do contexto do código que está sendo editado, reduzindo a necessidade de elaborar prompts explícitos e complexos para atender às solicitações do usuário. Tal abordagem otimiza o fluxo de trabalho, diminuindo barreiras técnicas no uso dessas tecnologias.

Especialmente sobre o tópico de criar e manter sua própria base de conhecimento, existe uma intersecção direta com a área da Arquitetura de Software, dado que a comunidade tem utilizado a gestão de conhecimento arquitetural como um dos fatores-chaves para tomar decisões sobre padrões e estilos arquiteturais desde os anos 2000 (OZKAYA, 2023).

Em geral, engenheiros de software tendem a escolher apenas padrões e estilos arquiteturais com os quais eles já trabalharam em algum momento, descartando decisões que se encaixariam melhor no projeto, sendo que muitas vezes essa decisão leva a custos de manutenção elevados no longo prazo (EISENREICH, SPETH E WAGNER, 2024). Isso pode se agravar ainda mais em cenários ágeis em que múltiplos times de desenvolvimento estão trabalhando em sistemas maiores e

compostos por diversos subsistemas, sendo que de acordo com De Diana, Kon e Gerosa (2010), o conceito de arquitetura emergente começa a ficar ainda mais complicado nessas situações.

De acordo com Ozkaya (2023), a gestão do conhecimento arquitetural inclui diversos elementos: projeto arquitetural, decisões de projeto, contexto, premissas etc. Todos esses elementos caracterizam um software em particular. Enquanto é possível entender o projeto através do código e de outros artefatos de implementação, decisões de projeto requerem conhecimento e contexto sobre os *trade-offs* (que são pontos de vantagens e desvantagens observados em decisões).

Dessa forma, o uso do conhecimento arquitetural em modelos de *LLM* pode contribuir significativamente para decisões mais embasadas e precisas sobre padrões e estilos arquiteturais. Essa abordagem não apenas reduz a dependência exclusiva da experiência individual dos engenheiros de software, mas também promove uma análise mais estruturada e abrangente dos *trade-offs* envolvidos. A ponderação desse conhecimento gerado por *LLMs* pode, portanto, representar uma ferramenta valiosa para a gestão da complexidade arquitetural em contextos de desenvolvimento ágeis, melhorando a qualidade das soluções técnicas, ainda mais em cenários envolvendo múltiplas equipes e subsistemas.

1.1 Objetivo

O objetivo deste trabalho é fazer uma análise exploratória no uso de modelos de linguagem de grande escala para a tomada de decisões arquiteturais. Seguindo o método proposto por Eisenreich, Speth e Wagner (2024), através da especificação de requisitos textuais (com ênfase em atributos de qualidade – também conhecidos como requisitos não funcionais) e modelos de domínio de um software real no escopo de *fintechs*, o modelo de linguagem de grande escala responderá com padrões e estilos arquiteturais junto de uma análise de *trade-offs*. Com isso, é apresentada uma discussão dos resultados obtidos para entender o uso dessa tecnologia para o processo de projeto de software.

1.2 Metodologia

O processo seguido por este trabalho é a aplicação prática do método proposto por Eisenreich, Speth e Wagner (2024) para a geração semiautomática de candidatos a arquitetura de software, a terceira etapa do método. O método completo, apresentado na Seção 3.4, é composto por seis etapas:

1. Gerar o modelo de domínio e cenários de caso de uso com base em requisitos textuais (automático);
2. Refinar o modelo de domínio e os cenários de uso gerados (manual);
3. Derivar múltiplos candidatos à arquitetura e as decisões arquiteturais usando o modelo de domínio, os cenários e os atributos de qualidade (automático);
4. Avaliar e comparar os candidatos à arquitetura (automático);
5. Refinar os candidatos à arquitetura (manual);
6. Selecionar o candidato que mais se encaixa no problema (manual).

Para tal geração de candidatos, primeiramente, junta-se todo o conhecimento arquitetural disponível de um sistema que está em sua fase inicial de modelagem de domínio: a lista de requisitos funcionais e atributos de qualidade, junto de uma primeira versão do modelo de domínio do problema. Depois, cria-se um *prompt* com todo esse conhecimento arquitetural e envia-se esse *prompt* para um modelo de *LLM*, coletando então as respostas geradas pela IA (Inteligência Artificial).

O conhecimento arquitetural do sistema a ser analisado será apresentado no Capítulo 4. Ainda no mesmo capítulo, será apresentada a solução real de estilos e padrões escolhidos para a resolução do problema em um contexto de uma empresa *fintech*, que foi simplificada e anonimizada, seguindo conceitos do *ATAM (Architecture Trade-off Analysis Method)*. Com isso, será possível discutir e comparar os resultados do *LLM* contra a solução real, analisando se essas respostas ajudam na tomada de decisões e debatendo os resultados alcançados.

1.3 Justificativa

O método proposto por Eisenreich, Speth e Wagner (2024), ainda não chegou na etapa de geração semiautomática de candidatos à Arquitetura de Software, tendo feito apenas uma análise exploratória com os modelos *LLaMA2* e *GPT-3.5*, para gerar modelos de domínio a partir dos requisitos funcionais textuais. Além disso, o método utiliza como base o estudo de caso do *MobSTR (Model-Based Safety Assurance and Traceability)*, um sistema de condução automatizada com foco em segurança e rastreabilidade baseado no desafio industrial *WATERS 2019 (Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems)*. O trabalho dos autores também propõe o uso de *LLM* para diversos processos da Arquitetura de Software, gerando então uma visão mais ampla da Arquitetura.

Este trabalho tem os seguintes complementos em relação ao trabalho de Eisenreich, Speth e Wagner (2024):

- Tem ênfase nas especificações textuais dos atributos de qualidade (também conhecidos como requisitos não funcionais) com foco no uso de IA para tomada de decisões sobre padrões e estilos arquiteturais;
- É a aplicação prática de uma etapa que o trabalho de Eisenreich, Speth e Wagner (2024) ainda não chegou, a etapa de geração semiautomática de candidatos à Arquitetura de Software
- São análises de resultados em cima de uma solução real no escopo de *fintech*, mais especificamente em um contexto de gestão de inadimplência, sendo diferente do setor industrial automotivo. Enquanto o setor automotivo tem maior foco em sistemas críticos de segurança e estabilidade, os sistemas financeiros possuem maior foco em integridade dos dados;
- Uso de um modelo mais recente de *LLM*, o *GPT-4o1*, que foi treinado com mais parâmetros, é capaz de simular cadeias de pensamento e possui uma janela de contexto mais ampla do que alguns modelos como *GPT-3.5* e *LLaMA2* apresentados no trabalho dos autores.

O principal motivo deste trabalho ter o foco na extração de estilos e padrões arquiteturais a partir dos atributos de qualidade é que, como mencionado anteriormente, esse é geralmente o ponto mais complicado de arquitetura emergente

em cenários ágeis com múltiplas equipes e subsistemas (DE DIANA, KON E GEROZA, 2010), além de que engenheiros de software tendem a escolher apenas padrões e estilos arquiteturais que eles já trabalharam em algum momento (EISENREICH, SPETH E WAGNER, 2024), sendo que o uso de *LLMs* podem ajudar os engenheiros de software a ponderarem outros padrões e estilos arquiteturais, além de ajudar a simplificar a manter a agilidade em múltiplas equipes e a manter a arquitetura emergente, que é um dos pontos fundamentais para o desenvolvimento ágil de software.

O uso de um modelo mais recente de *LLM* com as características mencionadas permite que o modelo possa iterar melhor em documentos extensos, além de promissoramente lidar com tarefas mais complexas, podendo então tomar decisões mais assertivas. Foi escolhido o *ChatGPT* pois na data de desenvolvimento do trabalho era o modelo mais conhecido e o único que oferecia recursos de simulação de cadeias de pensamento dentro do modelo.

1.4 Estrutura do trabalho

Nos Capítulos 2 e 3, são apresentadas as referências acadêmicas que explicam sobre elementos e conceitos de Arquitetura de Software utilizados neste texto, como requisitos funcionais e atributos de qualidade, decisões arquiteturais de software, gestão do conhecimento arquitetural e a geração semiautomática de candidatos à Arquitetura de Software. No Capítulo 4, são apresentados todo o conhecimento arquitetural do sistema real de uma *fintech* (generalizado e simplificado) e o *prompt* gerado para o *LLM* com o objetivo de gerar os candidatos à arquitetura junto das decisões arquiteturais. Já nos Capítulo 5 e 6 é possível visualizar a resposta do *LLM* para o *prompt* gerado e a discussão dos resultados apresentados, incluindo sugestões e ideias para trabalhos futuros.

2 MODELOS DE LINGUAGEM DE GRANDE ESCALA

De acordo com Dhar, Vaidhyanathan e Varma (2024), um Modelo de Linguagem é um modelo de IA probabilístico de Processamento de Linguagem Natural (do inglês, *NLP*, *Natural Language Processing*) que é projetado para gerar probabilidades associadas a uma sequência de palavras. Os autores diferenciam os Modelos de Linguagem de Grande Escala (do inglês *LLMs*, *Large Language Models*), enfatizando que são modelos probabilísticos de inteligência artificial similares, porém treinados com um volume extenso de dados, permitindo uma compreensão e geração de texto avançadas. Eles também observam que *LLMs* são amplamente utilizados em tarefas diversas de *NLP*, possuem bilhões de parâmetros e grandes janelas de contexto (*Context Lengths* e *Parameters*, que geralmente são características importantes para diferenciar modelos de *LLMs*). *Tokens* representam unidades de texto que podem corresponder a uma palavra completa ou partes de palavras, enquanto o *Context Length* em um *LLM* refere-se ao número de *tokens* considerados pelo modelo ao processar ou gerar texto (DHAR, VAIDHYANATHAN E VARMA, 2024).

Os autores ainda explicam que *LLMs* se baseiam na arquitetura de *Transformers*, que utiliza mecanismos de atenção como base de seu funcionamento. Os *Transformers* são compostos por dois componentes principais: um *Encoder* (codificador), responsável por processar o texto de entrada; um *Decoder* (decodificador), que gera o novo texto a partir das informações processadas pelo *Encoder* (DHAR, VAIDHYANATHAN E VARMA, 2024).

Kneusel (2024) explica que a interação com *LLMs* é feita através de *prompts*, que é a entrada fornecida por um usuário, geralmente em formato de texto. Alguns modelos multimodais podem ter *prompts* em formatos de arquivos, imagens, áudio, entre outros. Neste trabalho, o foco são os *prompts* em texto e utilizando o modelo GPT-4o1, da empresa OpenAI, que introduz conceitos de raciocínio dentro dos *LLMs*, utilizando técnicas internas de *Chain of Thought* (cadeias de pensamento), que é uma técnica de *prompt engineering* (OPENAI, 2024). O modelo da OpenAI foi escolhido para uso neste trabalho pois era o único capaz de simular cadeias de pensamento e também o mais conhecido no momento de escrita deste texto.

A técnica de cadeia de pensamento (*Chain of Thought - CoT*), conforme descrita pela OpenAI (2024), é um método que permite os *LLMs* melhorarem sua capacidade de raciocínio em tarefas complexas, promovendo uma abordagem iterativa para a solução de problemas. De acordo com a empresa, essa técnica envolve a geração de uma sequência lógica de passos intermediários antes de apresentar uma resposta final, permitindo ao modelo dividir problemas difíceis em partes menores, explorar abordagens alternativas quando necessário e refinar estratégias com base nos resultados obtidos. A empresa enfatiza que com o treinamento baseado em aprendizado por reforço, os modelos aprendem a identificar e corrigir seus próprios erros e a adaptar suas respostas para questões desafiadoras. Aproveitando desses recursos, neste trabalho é utilizado o modelo GPT-4o1 para a avaliação e tomada de decisões arquiteturais.

Além da técnica de *Chain of Thought*, a estruturação do *prompt* que será apresentada no Capítulo 4 utiliza-se de diversas outras técnicas mencionadas na documentação oficial da OpenAI, no site intitulado de “*Prompt engineering*” (OPENAI, 2024). No site, é mencionado por exemplo o uso de táticas como “incluir detalhes de contexto”, “pedir ao modelo para atuar como uma persona”, “especificar passos necessários para completar uma tarefa”, “dividir tarefas maiores em subtarefas” e “formatar a resposta esperada”. Todas essas técnicas são autoexplicativas, sendo que de acordo com a OpenAI (2024), elas melhoram a assertividade do modelo.

Kneusel (2024) observa que uma das propriedades dos *LLMs* é sua capacidade de aprendizado de contexto, sendo que esse é provavelmente um dos principais motivos pela empolgação de entusiastas na área de IA. O autor explica que o aprendizado de contexto diz respeito ao modelo que aprende dinamicamente a partir das informações que lhe são fornecidas, sem modificar seus pesos. Ele também enfatiza que o aprendizado de contexto é diferente do ajuste fino de um modelo, sendo que no ajuste fino, um modelo previamente treinado é adaptado a uma tarefa, atualizando os pesos com dados novos de treinamento. Ou seja, Kneusel (2024) observa que o aprendizado de contexto adiciona informações novas ao *LLM* como parte do *prompt*, enquanto mantém fixos os pesos do modelo. Isso conecta com a visão de que o conhecimento arquitetural de software pode contribuir para o processo de tomada de decisões

arquiteturais, já que se pode utilizar aprendizado de contexto dos LLMs com o conhecimento arquitetural para decisões mais assertivas.

3 ARQUITETURA DE SOFTWARE

Segundo Fairbanks e Garlan (2010), a Arquitetura de um Software geralmente está associada com uma visão de alto nível sobre o *design* de uma solução, de forma que se entende os elementos e componentes dessa solução junto das relações e características deles, aplicando uma ótica sobre como as decisões influenciam os atributos de qualidade e as restrições, analisando também como podem afetar a implementação ou manutenção de novas funcionalidades. Os autores também enfatizam que existe uma linha tênue entre arquitetura e *design* detalhado, em que o *design* detalhado seria decisões menores em baixo nível que estão diretamente relacionados à implementação de código dos sistemas, como por exemplo a escolha de nomes de variáveis ou uso padrões de projeto (*design patterns*). Entretanto, os autores observam que se essas decisões de baixo nível possuírem a característica de intencionalidade e gerarem impactos nos atributos de qualidade, ainda seria uma decisão arquitetural. Neste trabalho, serão explicados padrões e estilos arquiteturais na Seção 3.1.

Ainda de acordo com Fairbanks e Garlan (2010), é possível observar que pensar sobre Arquitetura de Software é importante porque ela impacta os sistemas em diferentes aspectos e pode-se reduzir riscos de falha em projetos. Eles comentam que todo software possui uma arquitetura, independentemente se foi investido muito tempo pensando sobre as decisões ou não; e se faz necessário entender qual é o risco que se deseja assumir ao pensar pouco sobre decisões arquiteturais. Eles citam por exemplo que talvez para sistemas de pouco risco ou que são menores, pode-se não ter a necessidade de pensar intencionalmente sobre todas as decisões arquiteturais. Neste trabalho, será feito uma análise exploratória no uso de modelos de linguagem de grande escala para ajudar a tomar melhores decisões arquiteturais de maneira intencional.

De acordo com Bass, Clements e Kazman (2013), as restrições em um projeto referem-se a decisões de *design* que já foram previamente estabelecidas, frequentemente associadas a condições externas como políticas organizacionais, limitações tecnológicas ou acordos comerciais, e que delimitam as escolhas disponíveis no processo de tomada de decisões arquiteturais. Os autores também

explicam que os requisitos funcionais determinam ações e comportamentos que o sistema deve executar, enquanto os atributos de qualidade especificam diferentes critérios como desempenho, segurança e adaptabilidade, que qualificam como essas funcionalidades devem ser implementadas. Eles também discutem que arquitetura de software conecta esses elementos ao organizar componentes e suas interações para alcançar os objetivos funcionais e de qualidade dentro das restrições impostas.

Neste trabalho, explora-se como o uso de técnicas de IA generativa pode contribuir para simplificar a identificação e priorização dessas decisões, promovendo alinhamento entre requisitos e restrições para facilitar a tomada de decisões arquiteturais.

3.1 Estilos e padrões arquiteturais de software

Estilos e padrões arquiteturais são conceitos fundamentais no *design* de software, embora possuam escopos distintos. Segundo Fairbanks e Garlan (2010), um estilo arquitetural refere-se a um conjunto de convenções de alto nível que define tipos de elementos arquiteturais como componentes, conectores e suas interações, além de impor restrições para orientar o *design* do sistema. Os autores mencionam exemplos que incluem estilos como *client-server*, *pipe-and-filter*, *map-reduce*, cada um influenciando atributos de qualidade específicos como escalabilidade ou manutenibilidade. Ainda de acordo com os autores, os padrões arquiteturais operam em um nível mais granular, focando em soluções reutilizáveis para problemas recorrentes em partes do *design*, como o uso do padrão REST para mensagens em sistemas com o estilo arquitetural *client-server*. Portanto, enquanto os estilos arquiteturais estruturam o sistema como um todo, padrões contribuem com detalhes de implementação dentro dessa estrutura maior, ambos sendo elementos-chave para alcançar qualidade e consistência no *design*, sendo que são considerados decisões arquiteturais se forem intencionalmente escolhidos, como citado na introdução do Capítulo 3.

Como observado por Fairbanks e Garlan (2010), a distinção entre padrões arquiteturais e estilos arquiteturais não é facilmente demarcada, especialmente em sistemas de grande escala. Os autores destacam que, à medida que surgem sistemas

compostos por outros sistemas menores, essa diferenciação torna-se ainda mais desafiadora. Como exemplo, eles citam que quando um sistema independente é incorporado a um sistema maior, o estilo arquitetural original do sistema menor pode ser considerado um padrão no contexto do sistema maior. Sendo que eles reforçam que essa ambiguidade leva a uma abordagem mais prática, em que padrões e estilos muitas vezes são tratados como conceitos intercambiáveis, dependendo do contexto e do nível de abstração em que são analisados. Por isso, neste trabalho, serão apresentadas diversas decisões, mas sem afirmar de maneira direta se é um padrão ou estilo arquitetural, pois a ênfase está em entender como as decisões foram tomadas deliberadamente para influenciar a qualidade da solução.

Serão mencionadas neste trabalho as seguintes decisões arquiteturais, na discussão do sistema real do escopo de inadimplência, no Capítulo 4:

- *Aggregate Pattern (DDD);*
- *Money Pattern;*
- *CQRS Pattern;*
- *Cache Write-Through Pattern;*
- Banco de dados com propriedades *ACID*;
- *Pub/Sub Pattern.*

De acordo com (FOWLER, 2013), o *Aggregate Pattern*, no contexto de *Domain-Driven Design (DDD)*, define um grupo de objetos de domínio tratados como uma unidade única. O autor menciona que esse padrão estabelece a raiz do agregado como a única abstração acessível externamente, garantindo a integridade do conjunto. O autor também observa que geralmente esse padrão é usado para limitar transações ao nível do agregado e organizar operações de armazenamento e recuperação de dados, reforçando a coesão e consistência no *design* de sistemas.

O *Money Pattern* propõe a criação de uma classe dedicada para representar e manipular valores monetários, considerando que linguagens de programação costumam não tratar dinheiro como um tipo de dado nativo (FOWLER, 2003). O autor destaca que esse padrão é especialmente útil em sistemas que lidam com múltiplas moedas, evitando erros como somar valores em dólares e ienes sem considerar suas

diferenças. Além disso, ele observa que o padrão ajuda a resolver problemas sutis, como erros de arredondamento em cálculos monetários, garantindo maior precisão e consistência em operações financeiras.

Fowler também traz uma definição sobre o *CQRS Pattern* (do inglês, *Command Query Responsibility Segregation*). Ele observa que o padrão promove a separação entre os modelos de comando (responsáveis por atualizações/inserções) e consulta (responsáveis por leitura) de um sistema, permitindo otimizações específicas para cada finalidade (FOWLER, 2011). O autor observa que essa abordagem pode ser vantajosa em dois contextos principais: aplicações de alto desempenho, onde a separação permite escalabilidade independente de leituras e escritas, e domínios complexos, onde separar os modelos reduz sobrecarga cognitiva. No entanto, Fowler (2011) ressalta que a aplicação do CQRS deve ser limitada a partes específicas do sistema, como *Bounded Contexts* no *DDD*, pois sua complexidade pode trazer mais riscos do que benefícios em cenários inadequados.

De acordo com AWS (2024), o *Cache Write-Through Pattern* é uma estratégia proativa de cache que atualiza os dados no cache imediatamente após qualquer atualização no banco de dados principal. Diferentemente do padrão *Cache-Aside*, que preenche o cache apenas após uma falha de consulta, o *Write-Through* mantém o cache sincronizado com o banco de dados, reduzindo significativamente a probabilidade de *cache miss* e melhorando o desempenho geral da aplicação, no entanto, esse padrão pode aumentar o custo, já que dados pouco requisitados também são armazenados no cache (AWS, 2024).

A MongoDB (2024) define as propriedades *ACID* (do inglês, *Atomicity, Consistency, Isolation, Durability*) como propriedades que garantem integridade e confiabilidade, mesmo diante de falhas. Atomicidade assegura que uma transação seja completada integralmente ou revertida em caso de erro; Consistência garante que as alterações preservem as regras do banco de dados, evitando estados inválidos; Isolamento protege transações simultâneas de interferências, e Durabilidade, garante que os dados persistam após a conclusão da transação, mesmo em falhas de sistema (MONGODB, 2024). A MongoDB também cita que entre as vantagens do modelo *ACID* estão a confiança em sistemas críticos, como bancos e processamento de

pagamentos, e o cumprimento de requisitos regulatórios; no entanto, o modelo também apresenta desvantagens, como maior complexidade de implementação e impactos de desempenho, especialmente em sistemas distribuídos, devido ao bloqueio de recursos para evitar conflitos durante as transações.

Já a Microsoft (2024) explica o *Pub/Sub Pattern* (do inglês, *Publisher-Subscriber*), descrevendo como uma abordagem que permite que aplicações anunciem eventos para múltiplos consumidores interessados de forma assíncrona, desacoplando o emissor (*publisher*) dos receptores (*subscribers*). Entre as principais vantagens do padrão estão o aumento da escalabilidade e desempenho, já que os emissores não precisam esperar pelas respostas dos receptores, e a capacidade de integrar sistemas heterogêneos usando diferentes plataformas e protocolos. No entanto, ele apresenta desafios como o gerenciamento de assinaturas, garantia de segurança nos canais de mensagens e problemas de ordenação ou duplicação de mensagens (MICROSOFT, 2024).

3.2 Método de avaliação de candidatos à Arquitetura de Software

Como visto nas seções anteriores, quando se fala de decisões arquiteturais, é comum associar o contexto das decisões com o contexto do software, chegando então, nas vantagens e desvantagens que cada decisão arquitetural traz para o contexto do software, que é conhecido também como *trade-offs* arquiteturais. Para estruturar a forma de analisar e avaliar diferentes *trade-offs*, Bass, Clements e Kazman (2013) descrevem o método *ATAM* (*Architecture Tradeoff Analysis Method*) como uma abordagem estruturada para arquitetos e *stakeholders* avaliarem de forma colaborativa escolhas arquiteturais que influenciam aspectos do software, com o objetivo de identificar riscos e oportunidades relacionadas às decisões arquiteturais e seus impactos nos atributos de qualidade do sistema.

Os autores mencionam que o *ATAM* é baseado na criação de cenários de atributos de qualidade, na identificação pontos sensíveis e *trade-offs*, e na documentação de riscos e “não-riscos”, estabelecendo uma conexão clara entre decisões arquiteturais e os objetivos de negócio do projeto. Além disso, a metodologia promove o engajamento dos participantes e gera artefatos valiosos, como árvores de utilidade e

relatórios que servem como referência para guiar a evolução da arquitetura, sendo especialmente relevante em projetos complexos, nos quais os custos de uma má decisão arquitetural podem ser significativos (BASS, CLEMENTS E KAZMAN, 2013).

Ainda segundo os autores, o método é estruturado em quatro fases principais: Preparação e Parceria, onde são definidos os objetivos do processo, organizados os participantes e alinhados os principais elementos da arquitetura a serem apresentados; Análise Inicial, com ênfase na apresentação dos objetivos de negócio, na descrição da arquitetura e na criação de uma árvore de utilidade, em que cenários são detalhados e priorizados para análise; Análise Detalhada, que envolve um grupo ampliado de *stakeholders*, em que são refinados e analisados os cenários mais críticos, avaliando as decisões arquiteturais em relação aos requisitos de qualidade e identificando sensibilidades, riscos e *trade-offs*; Relatório Final, em que são documentados os resultados, incluindo os riscos identificados, recomendações e temas críticos que conectam as decisões arquiteturais aos objetivos de negócio.

Neste trabalho, o *ATAM* será apresentado de maneira simplificada e adaptada, com ênfase principalmente na fase de “Análise Detalhada” para fazer comparações entre o modelo de *LLM* no Capítulo 5 e o sistema real de inadimplência no Capítulo 4.

3.3 Gestão do conhecimento arquitetural de software

Segundo Ozkaya (2023), conhecimento arquitetural em engenharia de software pode ser entendido como um conjunto estruturado de informações que orientam decisões de *design* e planejamento de sistemas, podendo abranger elementos como o projeto da arquitetura; as decisões de *design*; os contextos, restrições e requisitos nos quais o sistema será implantado. Capturar e gerenciar esse conhecimento é essencial para alinhar as escolhas de design às necessidades do projeto, otimizando a evolução contínua dos sistemas frente a mudanças tecnológicas e organizacionais (OZKAYA, 2023).

Ozkaya (2023) também enfatiza que ferramentas baseadas em IA representam uma oportunidade promissora para automatizar a captura e aplicação desse conhecimento,

facilitando a tomada de decisões arquiteturais e permitindo uma abordagem mais eficiente e alinhada às demandas dinâmicas do desenvolvimento de software.

Em resumo, o conhecimento arquitetural não tem exatamente apenas um formato fixo, mas pode ser compreendido como quaisquer informações que ajudam na tomada de decisões arquiteturais, por isso neste trabalho foi escolhido o uso de modelos de domínio como diagrama usando a linguagem *UML* (*Unified Modeling Language*), a descrição textual de requisitos funcionais e atributos de qualidade; sendo todo esse conhecimento usado para construir o *prompt* para o modelo de *LLM*.

3.4 Geração semiautomática de candidatos à Arquitetura de Software

Como observado nas seções anteriores, o processo de tomada de decisões arquiteturais tende a ser trabalhoso na construção e desenvolvimento de um software, embora tenha alto valor para a redução de riscos e escolhas mais assertivas, adequando a qualidade com o custo de desenvolvimento da solução. Por isso, nota-se cada vez mais esforços para automatizar ou facilitar essa atividade, como observa-se no trabalho de Eisenreich, Speth e Wagner (2024) em que a intenção é conectar as recentes pesquisas sobre *LLMs* e IA Generativa com Arquitetura de Software, mais especificamente, a tarefa de avaliação de diferentes decisões arquiteturais. Os autores enfatizam que a Arquitetura de Software cumpre um papel importante na qualidade de todo sistema e propõem um método para gerar arquiteturas de forma semiautomática utilizando IA. Eles mostram que o método é composto pelas seguintes principais etapas: gera-se primeiro os modelos de domínio e cenários de caso de uso usando *LLMs*, depois, derivam-se dos *LLMs* múltiplos candidatos à Arquitetura juntamente de suas respectivas decisões arquiteturais e por fim, avalia-se esses candidatos e toma-se uma decisão para a solução final, que passa a ser escolhida com mais dados por causa do método. Os autores citam que é importante ter iterações constantes e supervisionadas no método para melhorar os resultados devido à forma como os *LLMs* funcionam atualmente, por isso, “semiautomático”. Tal abordagem proposta no método pelos autores, promove uma maior qualidade nas decisões arquiteturais e acelera o processo de avaliação arquitetural, alinhando-se de maneira mais assertiva aos requisitos funcionais, atributos de qualidade e restrições do projeto.

Ainda de acordo com os autores, existem trabalhos relacionados que indicam avanços no uso de automação para apoiar decisões arquiteturais, porém, apresentam limitações que geralmente apresentam muito esforço manual e são focados em partes isoladas do ciclo de *design* – como a geração de modelos de domínio ou somente a avaliação de candidatos à arquitetura. Eisenreich, Speth e Wagner (2024) propõem um fluxo completo, que começa pela geração de modelos iniciais até a decisão final de escolha arquitetural. Essa abordagem se destaca não apenas por apoiar a decisão da solução, mas também por capturar e documentar decisões em forma de registros de decisões arquiteturais (*ADRs*, do inglês, *Architecture Decision Records*), promovendo um contexto histórico das decisões discutidas ao longo do projeto.

Adentrando nos detalhes do método proposto pelos autores, o processo é pensado para ser iterativo durante o ciclo de desenvolvimento de um software, sendo que ele possui seis etapas, onde algumas são automáticas (com auxílio do *LLM*) e outras manuais (com intervenção humana):

1. Automático: Gerar o modelo de domínio e cenários de caso de uso com base em requisitos textuais;
2. Manual: Refinar o modelo de domínio e os cenários de caso de uso gerados;
3. Automático: Derivar múltiplos candidatos à arquitetura e as decisões arquiteturais usando o modelo de domínio, os cenários e os atributos de qualidade;
4. Automático: Avaliar e comparar os candidatos à arquitetura;
5. Manual: Refinar os candidatos à arquitetura;
6. Manual: Selecionar o candidato que mais se encaixa no problema.

Na primeira etapa, espera-se transformar os requisitos e atributos de qualidade fornecidos em uma representação do domínio e em cenários de caso de uso, priorizando nesses artefatos as relações das entidades e omitindo detalhes específicos dessas entidades (como os atributos e comportamentos), já que é mais importante para as decisões arquiteturais entender essas relações pois é através delas que modulariza-se a arquitetura em componentes. Os cenários de caso de uso nessa etapa complementam o modelo de domínio com a descrição do funcionamento

do sistema. Em iterações recorrentes futuras nessa etapa, são incluídas também as decisões arquiteturais e a arquitetura atual para aumentar a assertividade ao gerar novamente modelos de domínio e cenários de caso de uso.

Na segunda etapa, o arquiteto deve entender o modelo de domínio e refiná-lo, fazendo uso inclusive de *LLMs* com *prompts* adicionais para auxiliar no processo de refinamento.

Na terceira etapa, espera-se transformar os requisitos, atributos de qualidade, cenários de caso de uso e modelo de domínio em diferentes candidatos à arquitetura com decisões arquiteturais. O formato exato de resposta e como será a interação com o *LLM* ainda está em discussão no trabalho de Eisenreich, Speth e Wagner (2024), sendo que tem sido levantadas hipóteses para fazer com que o *LLM* faça a divisão de componentes da arquitetura. Inclusive, estão querendo extrair as decisões arquiteturais em formato de *ADRs*.

Na quarta etapa, os autores querem auxiliar o arquiteto a tomar a decisão que melhor se encaixa no cenário através de uma automação de avaliação arquitetural utilizando métodos como o *ATAM*, sendo que a decisão de qual método utilizar também será discutida e caso prove-se ser muito difícil, tornaram essa etapa como semiautomática.

As etapas 5 e 6 consistem em o arquiteto entender as vantagens e desvantagens de cada candidato à decisão arquitetural e escolher uma que melhor se encaixa no problema, podendo então solicitar melhorias com novos *prompts* para melhorar a assertividade das respostas. Por fim, todo o processo é iterativo com a finalidade de suportar a característica iterativa do próprio processo ágil de desenvolvimento de software, assumindo que os requisitos podem mudar e que todo o método poderia ser iterado novamente.

Os autores fizeram uma análise exploratória somente da primeira etapa até então, utilizando *LLaMA2* e *GPT-3.5*, utilizando como base o estudo de caso do *MobSTr*, um sistema de condução automatizada com foco em segurança e rastreabilidade que aconteceu no desafio industrial *WATERS 2019 (Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems)*, um workshop sobre o tema.

Neste trabalho, o objetivo é fazer uma análise e aplicação prática apenas da terceira etapa em uma solução real no escopo de uma *fintech*, etapa que o trabalho de Eisenreich, Speth e Wagner (2024) ainda não chegou, utilizando-se do modelo de *LLM* “*GPT-4o1*” da *OpenAI*.

As etapas 1 a 2 foram retiradas de uma solução real e geradas ainda de maneira manual, tendo sido simplificado e anonimizado, contendo apenas um contexto delimitado da solução e maior ênfase nos atributos de qualidade. Apesar de a etapa 3 mencionar o uso de cenários de caso de uso, a análise exploratória do trabalho de Eisenreich, Speth e Wagner (2024) contém apenas uma lista de requisitos. Será seguido neste trabalho o mesmo formato de requisitos encontrados na análise exploratória, sem casos de uso e com ênfase em atributos de qualidade, mas analisando o comportamento do *LLM* para a geração de candidatos à arquitetura de software e análise de *trade-offs*.

4 CONTEXTO DO PROBLEMA E SOLUÇÃO REAL

Como já mencionado, o objetivo deste trabalho é fazer uma análise exploratória no uso de *LLMs* para a tomada de decisões arquiteturais. Seguindo o método proposto por Eisenreich, Speth e Wagner (2024), através da especificação de requisitos textuais (com ênfase em atributos de qualidade) e modelos de domínio de um software real no escopo de *fintechs*, o modelo de linguagem de grande escala responderá com padrões e estilos arquiteturais junto de uma análise de *trade-offs*. Com isso, serão discutidos os resultados obtidos para entender o uso dessa tecnologia para o processo de projeto de software.

Nesse capítulo, são apresentados alguns conhecimentos arquiteturais e a solução que foi implementada em um ambiente real de uma *fintech*, tendo o contexto do problema sido simplificado e generalizado, de forma a omitir especificidades da empresa e reduzir o escopo. O formato escolhido para a representação dos requisitos e do modelo de domínio foi o mesmo encontrado na análise exploratória do método de Eisenreich, Speth e Wagner (2024).

O sistema em questão trata da gestão de inadimplência, que é um tema recorrente e crítico em diferentes *fintechs*, pois impacta diretamente a saúde financeira da organização e contribui para a continuidade operacional do negócio. Em linhas gerais, inadimplência é o não cumprimento de obrigações financeiras no prazo acordado. Realizar a gestão da inadimplência é um processo composto por diferentes tarefas, como: entender quem são os clientes inadimplentes, quais dívidas eles possuem, como cobrar de forma adequada esses clientes, como entender o comportamento da inadimplência ao longo do tempo para tomada de decisões estratégicas e afins.

Os conhecimentos arquiteturais incluem: a especificação textual dos requisitos funcionais e dos atributos de qualidade, o modelo de domínio inicial pensado para o problema, a solução arquitetural que foi implementada para esse problema e a identificação de *trade-offs* para essa solução arquitetural utilizando o método *ATAM*. Sendo que o maior foco é na especificação de requisitos de atributos de qualidade, pois esses são os que mais impactam na escolha de decisões arquiteturais. Todos os

artefatos foram mapeados com base na principal funcionalidade: “O sistema deve gerenciar a inadimplência dos clientes”.

4.1 Requisitos funcionais

Na Tabela 1 são apresentados os requisitos que contextualizam o problema, definindo conceitos importantes para o funcionamento do sistema.

Tabela 1 - Requisitos funcionais textuais do sistema de inadimplência

Id	Descrição
FR-1	O sistema deve gerenciar a inadimplência dos clientes
FR-1.1	Um cliente inadimplente possui um documento (CPF ou CNPJ), o conjunto de suas dívidas e seus respectivos pagamentos realizados para essas dívidas
FR-1.2	Uma dívida é caracterizada por: valor bruto, valor líquido (após todos os pagamentos realizados), data de surgimento da dívida, motivo da dívida
FR-1.3	Um pagamento para uma dívida é caracterizado por: valor pago e hora do pagamento
FR-1.4	Saldo de inadimplência é a soma de valor líquido de todas as dívidas de todos os clientes inadimplentes, agrupado por dia, é caracterizado por: saldo, data de referência, documento do cliente
FR-1.5	Inadimplência de um cliente é diferença entre a soma de valor bruto e a soma de valor líquido de todas as dívidas, agrupado por dia e por documento do cliente inadimplente, é caracterizado por: saldo, data de referência, documento do cliente, todas as dívidas do cliente até a data de referência, todos os pagamentos do cliente até a data de referência
FR-1.6	O sistema deve permitir o acesso ao histórico de clientes inadimplentes
FR-1.7	O sistema deve permitir o acesso ao histórico de dívidas para clientes inadimplentes
FR-1.8	O sistema deve permitir o acesso ao histórico de pagamentos de dívidas
FR-1.9	O sistema deve emitir um relatório chamado "Foto da Inadimplência" com o saldo de inadimplência de acordo com o dia solicitado, que é uma data de referência
FR-1.10	O sistema deve emitir um relatório chamado "Inadimplência do Cliente", contendo a inadimplência de um cliente específico e informando de maneira detalhada todas as dívidas e pagamentos desse cliente (com todas as características de dívidas e pagamentos)

Fonte: Elaborado pelo autor.

A coluna “Id” (Identificador) identifica os requisitos para facilitar a interação com o LLM, ao solicitar novos *prompts* ou o modelo precisar citar eles. Os requisitos foram derivados da principal funcionalidade de “gerenciar inadimplência dos clientes”, que é descrita no FR-1; *FR* é sigla para Requisito Funcional (do inglês, *Functional Requirement*). Por isso, todos os requisitos seguem uma hierarquia de numeração derivados do FR-1, como por exemplo, o FR-1.1. A descrição dos requisitos

representa as principais funcionalidades que o sistema deve atender com foco em entender termos de negócio e a relação entre diferentes entidades.

4.2 Atributos de qualidade

A seguir na Tabela 2 são apresentados os atributos de qualidade que contextualizam a solução arquitetural implementada para o problema. Nas colunas da Tabela 2, a Categoria, representa qual é o escopo de problema que o atributo de qualidade se encaixa. As categorias podem ser: “Precisão; Disponibilidade; Confiabilidade; Desempenho; Capacidade; Observabilidade; Auditabilidade”. Já a “Métrica de Sucesso”, quantifica o atributo de qualidade de maneira que fique clara a expectativa de como o sistema atende a esse atributo. Ainda na Tabela 2, alguns atributos de qualidade estão descritos com alguns termos como “D-0” e “D-1”, essa nomenclatura é uma forma de referenciar temporalidade em dias (“D”) relativos à data de ocorrência de geração dos dados. Por exemplo, D-6 de uma data específica seriam 6 dias antes dessa data. A coluna “Id” (Identificador) identifica os requisitos não funcionais (*NFR*, do inglês, *Non-Functional Requirement*, conhecidos também como atributos de qualidade) para facilitar a interação com o LLM, ao solicitar novos *prompts* ou o modelo precisar citar eles.

Tabela 2 - Atributos de qualidade do sistema de inadimplência.

Id	Descrição	Categoria	Métrica de Sucesso
FR-1	O sistema deve gerenciar a inadimplência dos clientes	-	-
NFR-1.1	Erros em cálculos financeiros envolvendo dívidas e pagamentos precisam ser igual ou inferior a um milionésimo (0,000001), principalmente para manter o saldo de inadimplência correto	Precisão	Taxa de erro em operações matemáticas envolvendo decimais $\leq 10^{-6}$
NFR-1.2	O relatório de "Foto da Inadimplência", quando é solicitado em D-0 em relação ao dia atual do sistema como data de referência deve estar disponível em até 2 minutos após solicitado a sua emissão	Disponibilidade	Tempo de resposta ≤ 2 minutos
NFR-1.3	O relatório de "Foto da Inadimplência", quando é solicitado em D-0 em relação ao dia atual do sistema como data de referência pode conter divergências de até 1 milhão em relação ao saldo da inadimplência real final do próprio dia	Confiabilidade	(Saldo da inadimplência real - Saldo da inadimplência reportado) ≤ 1 milhão
NFR-1.4	O relatório de "Foto da Inadimplência", quando é solicitado em D-1 em relação ao dia atual do sistema como data de referência deve estar disponível em até 40 minutos após solicitado a sua emissão	Disponibilidade	Tempo de resposta ≤ 40 minutos
NFR-1.5	O relatório de "Foto da Inadimplência", quando é solicitado em D-1 em relação ao dia atual do sistema como data de referência não pode ter divergências relação ao saldo da inadimplência real final do próprio dia	Confiabilidade	(Saldo da inadimplência real - Saldo da inadimplência reportado) = 0
NFR-1.6	O relatório de "Inadimplência do Cliente", quando é solicitado em D-0 em relação ao dia atual do sistema como data de referência deve estar disponível em até 1 minuto após solicitado a sua emissão	Disponibilidade	Tempo de resposta ≤ 1 minuto
NFR-1.7	O sistema deve suportar a persistência de novas dívidas em até 500 milissegundos	Desempenho	Tempo de resposta $\leq 500ms$
NFR-1.8	O sistema deve permitir o acesso ao histórico de novos pagamentos em até 400 milissegundos	Desempenho	Tempo de resposta $\leq 400ms$
NFR-1.9	O sistema deve suportar a persistência concorrente de até 50 novas dívidas	Capacidade	Throughput ≥ 50 transações simultâneas
NFR-1.10	O sistema deve suportar o registro persistente simultâneo de até 600 novos pagamentos	Capacidade	Throughput ≥ 600 transações simultâneas
NFR-1.11	O sistema deve emitir métricas sobre Throughput, Tempo de resposta, Latência de todas as dependências de I/O	Observabilidade	Ter dashboard de Observabilidade
NFR-1.12	O sistema deve manter o histórico de dívidas, pagamentos e clientes inadimplentes por 6 anos para fins de auditoria	Auditabilidade	Dados recuperáveis por até 6 anos
NFR-1.13	O histórico solicitado dentro desse período de 6 anos para fins de auditoria pode ser emitido em até 5 dias	Disponibilidade	Tempo de resposta ≤ 5 dias

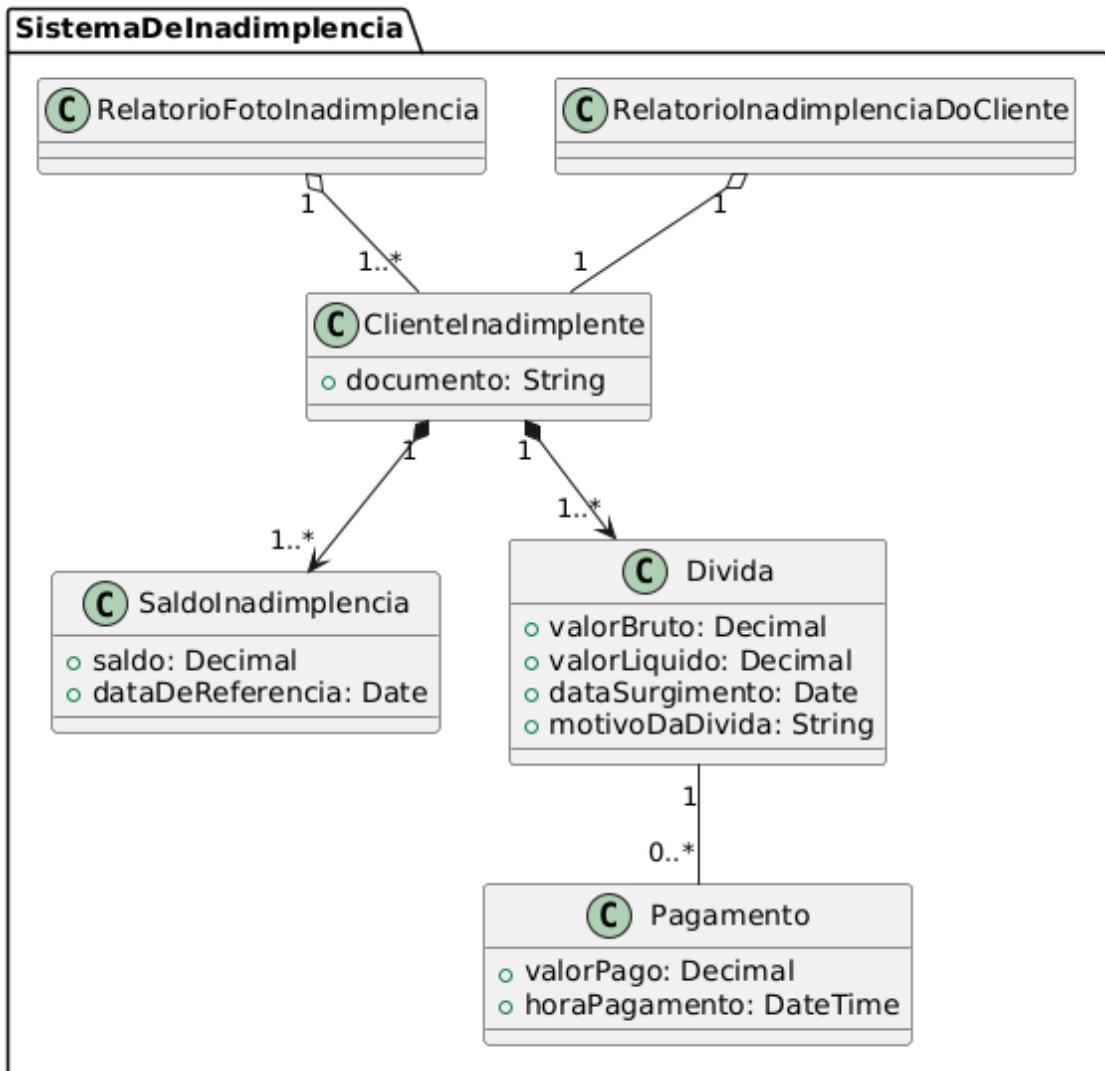
Fonte: Elaborado pelo autor.

Os atributos de qualidade foram derivados da principal funcionalidade de “gerenciar inadimplência dos clientes”, que é descrita no FR-1. Por isso, todos os atributos de qualidade seguem uma hierarquia de numeração derivados do FR-1, como por exemplo, o NFR-1.1. Esses requisitos possuem um contexto sobre tamanho da companhia, justificando os números apresentados na tabela.

4.3 Modelo de domínio

A Figura 1 apresenta o diagrama de domínio para os requisitos mencionados na Seção 4.1, utilizando o formato de representação de diagrama de classes na linguagem *UML (Unified Modeling Language)*.

Figura 1 - Diagrama do modelo de domínio dos requisitos funcionais.



Fonte: Elaborado pelo autor.

No diagrama, o Cliente Inadimplente é composto por suas respectivas dívidas. Nesse contexto, são criados somente Clientes quando eles já estão inadimplentes (ou seja, não cumpriram obrigações financeiras no prazo acordado), portanto, a Dívida precisa existir e isso explica a relação de um Cliente Inadimplente que é composta por no mínimo uma ou mais dívidas.

Essas dívidas podem ser pagas parcialmente ou totalmente, então uma dívida tem uma relação de associação com nenhum ou diversos pagamentos.

O Saldo da Inadimplência é gerado diariamente diversas vezes ao longo do dia, sempre que uma nova dívida é criada, aumentando o saldo de inadimplência do cliente ao longo do tempo, por isso o Saldo está em relação de composição com o Cliente Inadimplente, sendo que o Cliente Inadimplente é o objeto-todo responsável por criar e destruir suas respectivas partes, tanto na relação com a Dívida quanto a relação do Saldo da Inadimplência.

Como apresentado na Seção 4.1, existem dois relatórios: “Foto da Inadimplência” e “Inadimplência do Cliente”, em que o primeiro é uma visão geral de todas as dívidas de todos os clientes inadimplentes agrupados por dia; e o segundo, é o detalhamento da inadimplência de um cliente específico, contendo também o saldo de inadimplência desse cliente junto com os detalhes das dívidas. Por isso, no diagrama da Figura 1, observa-se que os relatórios possuem uma relação de agregação com o Cliente Inadimplente e o que muda entre eles é exatamente a relação de multiplicidade, que pode estar relacionado com muitos clientes ou apenas um.

4.4 Solução real implementada

A partir dos requisitos funcionais, modelos de domínio e atributos de qualidade, as seguintes decisões arquiteturais foram escolhidas como uma solução real para o problema, usado por uma empresa *fintech*, descritos na Tabela 3. A coluna “Id” (Identificador) identifica os requisitos para facilitar a interação com o *LLM*, ao solicitar novos *prompt* e analisar as respostas do modelo.

Tabela 3 - Decisões arquiteturais do sistema de inadimplência

Id	Decisão	Vantagens	Desvantagens
ADR-1	DDD Aggregate Pattern	Promove abstração de transações na camada de domínio e garante consistência interna, reduz impactos de operações concorrentes	Pode aumentar a curva de aprendizado sobre o sistema e pode gerar problemas de desempenho se não for bem pensado o escopo dos agregados
ADR-2	Money Pattern	Promove coesão e semântica de valores financeiros, garante abstração de arredondamentos por todo o código	Pode gerar alocações excessivas na memória e aumenta curva de aprendizado sobre o sistema
ADR-3	CQRS Pattern	Otimiza operações de leitura e escrita, melhorando desempenho e incentiva a escalabilidade horizontal	Consistência eventual pode aumentar complexidade de manutenção e detecção de erros
ADR-4	Cache Write-Through Pattern	Promove redução no tempo de resposta, alivia carga de banco de dados operacional e promove a sensação de disponibilidade percebida por clientes do sistema	Pode aumentar complexidade de manutenção por ter que lidar com expiração do cache
ADR-5	Banco de dados com propriedades ACID	Reduz problemas relacionados a concorrência, reduz problemas em cenários de falha	Pode aumentar complexidade para escalabilidade em alto volume de transações simultâneas
ADR-6	Pub/Sub Pattern	Promove processamento assíncrono, facilitando a geração de novos dados em cenários de alto volume de dados	Pode aumentar complexidade de manutenção e ser difícil de investigar problemas, dependendo onde é aplicado

Fonte: Elaborado pelo autor.

As decisões arquiteturais contêm diversos padrões escolhidos intencionalmente para atender aos requisitos específicos. Como observado no Capítulo 3 sobre Arquitetura de Software, existe uma linha tênue entre design de alto nível e baixo nível, então os padrões descritos na Tabela 3 podem ter diferentes pontos de vista: visão de código, armazenamento de dados, relação entre componentes do software e afins. A tabela mostra como essas decisões afetam os diferentes atributos de qualidade e quais *trade-offs* essas decisões carregam.

A seguir, na Tabela 4, observa-se a relação de quais decisões arquiteturais afetam quais atributos de qualidade mencionados na Seção 4.2.

Ao analisar a relação entre a Tabela 3 e a Tabela 4, observa-se que as vantagens encontradas na decisão arquitetural (ADR) afetam múltiplos atributos de qualidade. Isso acontece porque os atributos de qualidade apesar de muitas vezes apresentarem propostas específicas, ainda fazem parte de uma categoria de problemas que são comuns entre eles. Por exemplo, os atributos de qualidade “NFR-1.3; 1.5; 1.9 e 1.10” envolvem métricas de sucesso que são sobre transações simultâneas (concorrência, que está relacionado a capacidade) e divergências de números (confiança desses números, relacionado à confiabilidade); pontos em que o Aggregate Pattern tratam através de abstrações sobre uso de transações na camada de domínio.

Outro ponto importante é que existem algumas decisões que até poderiam afetar ainda mais categorias diferentes, como a “ADR-4” que fala sobre um padrão de Cache; esse padrão até poderia ser uma vantagem para a categoria de “Desempenho”, além da categoria “Disponibilidade” e “Auditabilidade” que são referenciadas na Tabela 4, porém, no sistema real foi tomada a decisão de uso de Cache somente nos fluxos dos atributos “NFR-1.2; 1.4; 1.6 e 1.13”, isso porque pelo contexto do sistema, não teriam ganhos de usar cache no momento de inserções de novas dívidas no escopo transacional. Portanto, a relação na Tabela 4 mostra também quais ADRs foram utilizadas em atributos específicos considerando outros contextos fora do escopo aqui.

Tabela 4 - Relação entre quais decisões arquiteturais atendem à quais atributos de qualidade no sistema de inadimplência

Id	Atributo(s) atendido(s)	Categorias de atributos
ADR-1	NFR-1.3; NFR-1.5; NFR-1.9; NFR-1.10	Confiabilidade; Capacidade
ADR-2	NFR-1.1	Precisão
ADR-3	NFR-1.2; NFR-1.4; NFR-1.6; NFR-1.7; NFR-1.8; NFR-1.9; NFR-1.10; NFR-1.12; NFR-1.13	Disponibilidade; Desempenho; Auditabilidade; Capacidade
ADR-4	NFR-1.2; NFR-1.4; NFR-1.6; NFR-1.13	Disponibilidade; Auditabilidade
ADR-5	NFR-1.3; NFR-1.5; NFR-1.9; NFR-1.10	Confiabilidade; Desempenho; Capacidade
ADR-6	NFR-1.2; NFR-1.3; NFR-1.11	Disponibilidade; Confiabilidade; Observabilidade

Fonte: Elaborado pelo autor.

4.5 Instruções para o modelo de linguagem de grande escala

Através de todo o conhecimento arquitetural citado nas Seções 4.1, 4.2 e 4.3 (requisitos funcionais, atributos de qualidade e modelagem inicial do domínio), foi estruturado um *prompt* para interagir com o modelo de *LLM*, utilizando algumas das diversas técnicas encontradas no site da OpenAI, que são mais detalhadas no Capítulo 2. O conhecimento arquitetural foi condensado em formatos textuais para facilitar o uso do *LLM*, sendo que foi utilizado a linguagem específica de domínio (*DSL*, do inglês, *Domain-Specific Language*) *PlantUML* para o diagrama de classes e

arquivos CSV (do inglês, *Comma-separated values*, valores separados por vírgula) para os requisitos textuais e atributos de qualidade. A Figura 2 mostra o *prompt* que foi criado.

Figura 2 - Prompt criado para o LLM

Você é um arquiteto de software. Sua tarefa é me fazer enxergar pontos que talvez eu não iria ponderar na hora de tomar decisões sobre um projeto. Pense passo a passo antes de responder:

- 1- Colete quais são os estilos e padrões arquiteturais de software baseado em referências acadêmicas importantes da área (utilize livros clássicos e modernos conhecidos, como Software Architecture In Practice, Fundamentals of Software Architecture, Patterns of Enterprise Application Architecture e afins), sabendo que existe diferença entre padrão e estilo arquitetural de software. Seja criativo na busca.
- 2- Colete as vantagens e desvantagens de cada um desses estilos e padrões, focando principalmente em quais atributos de qualidade (também chamado de requisito não funcional) cada um desses estilos e padrões ajudam (ou atrapalham).
- 3- Leia o modelo de domínio que vou te mandar em formato de PlantUML em texto para entender o contexto do problema.
- 4- Leia os requisitos que vou te mandar em formato de CSV, onde tem as seguintes colunas, a coluna mais importante é a de "Métrica de sucesso para atingir o Atributo de Qualidade (Fitness Function)", que é um objetivo metrificando um pouco mais o requisito não funcional, que está na coluna "Categoria".
- 5- Baseado em todo o contexto prévio que você adquiriu em nossa conversa, me traga estilos e padrões que eu poderia usar ou ponderar na hora de tomar decisões sobre a solução técnica para esse problema.
- 6- Formate a resposta utilizando a fase 4 do método ATAM (Architecture Tradeoff Analysis Method), focando somente em uma etapa: identificação de trade-offs. Você dirá qual é o estilo/padrão e tradeoffs tenho para o meu contexto.
- 7- Diga de forma resumida se entendeu e nesse prompt, foque em buscar contexto para os pontos 1 e 2. Vou te enviar o CSV e o modelo de domínio em PlantUML no próximo prompt, só me dê a resposta formatada do passo 6 após eu te enviar CSV e modelo em PlantUML.

Fonte: Elaborado pelo autor.

É possível perceber na Figura 2 que uma das principais técnicas empregadas na construção do *prompt* é o aproveitamento da funcionalidade de cadeia de pensamento (*Chain of Thought*) do modelo GPT-4o1 da OpenAI para quebrar um problema grande em subproblemas e deixar bem especificado todas as expectativas que se tem de resposta, incluindo o pedido para que o modelo atue com o papel de arquiteto de software, aplicando a técnica também de atribuição de personagens no *prompt*.

Outra técnica importante aplicada é a de “dar espaço para o modelo pensar”, no qual esse primeiro *prompt* é enviado, aguarda-se a resposta do modelo para ele entender as instruções e só depois envia-se o diagrama de classes do modelo inicial de domínio em *PlantUML*. Novamente, aguarda-se o feedback do *LLM* para ela “entender o conteúdo” e então, envia-se por fim, os requisitos textuais em formato CSV. No total,

foram três interações de *prompts* com o modelo, sendo que as duas últimas não serão apresentadas em figuras pois são basicamente os mesmos tópicos apresentados nas seções 4.1, 4.2 e 4.3, só que em formatos específicos. Serão discutidos os resultados no capítulo seguinte.

Durante a fase de concepção do *prompt*, foi testado diversas outras técnicas diferentes e percebeu-se que nem todas trouxeram tanta assertividade. Quanto menos contexto e mais “solicitações ao mesmo tempo” um *prompt* contém, a sensação é que o modelo acaba sendo menos assertivo. Essas “solicitações ao mesmo tempo” seriam múltiplas tarefas sem especificar qual é a ordem a ser seguida ou sem deixar claro o formato e resultado esperado. Por exemplo, pedir ao modelo algo muito genérico como “Traga trade-offs baseado nos requisitos que vou te enviar” acaba sendo pouco efetivo, o modelo começa a dar respostas que não fazem sentido ou que pouco ajudam (termo geralmente conhecido como alucinações de modelos de *LLM*).

5 RESULTADOS OBTIDOS DO MODELO DE LINGUAGEM DE GRANDE ESCALA

Após o envio do primeiro *prompt* estruturado apresentado na seção 4.5, um fenômeno interessante já pode ser notado logo na primeira resposta que o *LLM* apresenta, em que é visto um processo de “pensamento” mesmo, buscando entender justamente quais referências pode-se utilizar da literatura e sugerindo diversos padrões e estilos arquiteturais diferentes. A primeira resposta é bem longa por essa característica, então a Figura 3 apresenta apenas parte dessa resposta. A iteração completa com o modelo de *LLM* está no APÊNDICE A – PROMPT E SAÍDA COMPLETA DO *LLM*.

É possível perceber na Figura 3 que todas as técnicas de *prompt* empregadas fazem a diferença, pois antes do modelo tentar responder de maneira direta, nota-se a coleta de todo o contexto necessário para preparar a resposta final, claramente isso é o aprendizado de contexto mencionado no Capítulo 2 que está em execução, sendo notável uma das características mais interessantes desse tipo de Inteligência Artificial. Não foi preciso ter conhecimento sobre algoritmos complexos de *Deep Learning* ou entendimento profundo sobre estatística aplicada para induzir o modelo a fazer uma tarefa avançada.

Figura 3 - Primeira resposta do LLM ao prompt inicial enviado

Entendido. Abaixo segue meu entendimento e a busca de contexto para os pontos 1 e 2 (ainda não respondendo o passo 6, conforme solicitado):

Contexto Buscado – Passo 1 (Coletar Estilos e Padrões Arquiteturais):

Em termos de literatura clássica e moderna sobre Arquitetura de Software, algumas obras de referência incluem:

- “Software Architecture in Practice” (Bass, Clements, Kazman)
- “Fundamentals of Software Architecture” (Mark Richards, Neal Ford)
- “Patterns of Enterprise Application Architecture” (Martin Fowler)
- “Documenting Software Architectures: Views and Beyond” (Clements, et al.)
- “Enterprise Integration Patterns” (Gregor Hohpe, Bobby Woolf)

A partir dessas referências, podemos destacar diferentes estilos e padrões arquiteturais:

Estilos Arquiteturais (Architectural Styles):

1. Camadas (Layered Architecture)
2. Hexagonal/Ports and Adapters
3. Arquitetura em Microservices
4. Arquitetura Monolítica
5. Arquitetura Orientada a Serviços (SOA)
6. Arquitetura Baseada em Eventos (Event-Driven Architecture - EDA)
7. Arquitetura Orientada a Mensagens (Message-Oriented Middleware - MOM)
8. Arquitetura Pipeline/Filter
9. Arquitetura Baseada em Componentes (CBA)

Padrões Arquiteturais (Architecture Patterns):

Do ponto de vista de padrões, muitas vezes são vistos como soluções para problemas recorrentes de design dentro de um estilo. Alguns exemplos:

1. Model-View-Controller (MVC)
2. Model-View-Presenter (MVP)
3. Model-View-ViewModel (MVVM)
4. Domain-Driven Design (DDD) patterns (Aggregate, Domain Services, Bounded Context, Repository)
5. Padrões de Integração Empresarial (como o “Saga Pattern”, “Event Sourcing”, “CQRS”, “Publish/Subscribe”, “Anti-corruption Layer”, “Strangler Fig”, etc.)
6. Padrões de Otimização de Performance (Caching, Circuit Breaker, Bulkhead, etc.)

Contexto Buscado – Passo 2 (Vantagens, Desvantagens, Atributos de Qualidade):

Abaixo seguem vantagens e desvantagens de alguns estilos e padrões, bem como quais atributos de qualidade impactam:

Depois, ao enviar o modelo inicial de domínio em formato de *PlantUML*, o modelo responde da seguinte forma, na Figura 4.

Figura 4 - Segunda resposta do LLM para o modelo de domínio

Thought about domain model structure por 4 segundos

Entendido. A partir do diagrama de domínio em PlantUML, vejo um contexto voltado para um sistema de inadimplência, onde temos entidades como **ClienteInadimplente**, **Dívida**, **Pagamento**, **SaldoInadimplencia**, além de relatórios específicos para acompanhar a situação desses clientes. Isso sugere um domínio voltado a gestão financeira, tracking de dívidas, pagamentos e geração de relatórios.

Focando ainda nos pontos 1 e 2 (levantamento de contexto sobre estilos e padrões arquiteturais e seus impactos em atributos de qualidade), comprehendo que o domínio apresentado pode vir a se beneficiar de padrões e estilos que facilitem a manutenção, escalabilidade e testabilidade, principalmente considerando que o domínio lida com regras de negócio sensíveis (cálculos de saldo, valores pagos, surgimento de dívidas ao longo do tempo). Por exemplo:

- Estilos mais modulares (como Arquitetura em Camadas ou Hexagonal) podem ajudar na manutenibilidade e testabilidade.
- Padrões como DDD podem ajudar a capturar as nuances do domínio de inadimplência de forma clara, reduzindo complexidade a longo prazo.
- Caso haja necessidade de escalabilidade e distribuição, padrões de microservices ou event-driven podem ser considerados, embora aumentem a complexidade operacional.

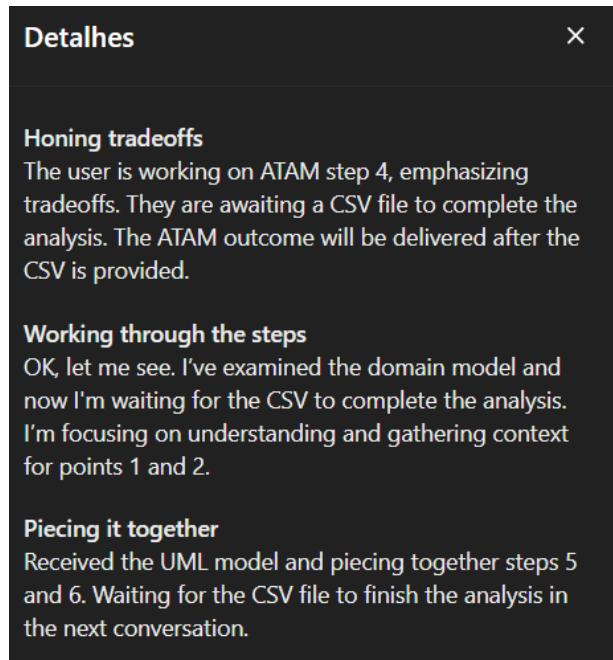
Em resumo, comprehendi o contexto do domínio e continuo focado em entender como as diferentes abordagens arquiteturais (pontos 1 e 2) podem se alinhar com os requisitos não funcionais, preparando o terreno para, após receber o CSV (com requisitos e métricas de sucesso), propor uma análise de trade-offs conforme solicitado.

⏮ ⏴ ⏵ ⏵ ⏵ ⏵

Fonte: Elaborado pelo autor.

É visível que o modelo já começa a afunilar a resposta conectando o conteúdo que ele coletou na Figura 3 com o novo conhecimento arquitetural (o diagrama inicial do modelo de domínio), apesar de ainda não ser a resposta final do modelo, ele começa a citar pontos como “talvez trazer estilos modulares como a Arquitetura em Camadas ou Arquitetura Hexagonal possa ajudar no longo prazo”. O ChatGPT permite também visualizar quais foram os processos executados internamente do *Chain of Thought*, reforçando como ele estava afunilando a resposta, como pode ser visto na Figura 5.

Figura 5 - Processo de Chain of Thought interno do modelo de LLM para fornecer a segunda resposta



Fonte: Elaborado pelo autor.

Na Figura 5, o processo de *Chain of Thought* fica muito evidente, ele começa a “raciocinar” de forma sequencial: “*Preciso formatar a resposta utilizando ATAM e avaliar trade-offs*”; “*Vou examinar o modelo de domínio agora e focar em entender o contexto por completo*”; “*Quando eu receber o arquivo CSV com os requisitos, vou juntar tudo para emitir a resposta final*”.

Por fim, a resposta final para o modelo com a análise de *trade-offs* é apresentada nas Tabelas 5 e 6.

Tabela 5 - Resposta final do LLM com as decisões arquiteturais

Decisão	Vantagens	Desvantagens
Arquitetura em Camadas + Domain-Driven Design (DDD)	O DDD combinado com uma arquitetura em camadas (Aplicação, Domínio, Infraestrutura) melhora a manutenibilidade e clareza do domínio, facilitando garantir a precisão (NFR-1.1) nos cálculos financeiros e mantendo regras de negócio coerentes. Também torna mais simples a auditabilidade (NFR-1.12), já que o domínio é bem definido e rastreável.	Pode aumentar a complexidade inicial de desenvolvimento e o tempo de resposta (por adicionar camadas), comprometendo parte das metas de desempenho (NFR-1.7 e NFR-1.8) se não houver otimizações. Além disso, o acréscimo de camadas pode afetar a latência e dificultar atingir metas de tempo estritas para relatórios (NFR-1.2, NFR-1.4, NFR-1.6).
Arquitetura Hexagonal (Ports and Adapters)	Ajuda na flexibilidade e substituição de fontes de dados ou serviços externos, facilitando a observabilidade (NFR-1.11) e o cumprimento de requisitos de auditabilidade. Permite manter a precisão no domínio central, isolado de infraestruturas que podem introduzir erros.	Exige maior esforço de configuração inicial e abstrações adicionais, o que pode afetar desempenho em cenários de alto throughput (NFR-1.9, NFR-1.10), já que cada porta/adaptador pode adicionar overhead.
Microservices + Event-Driven Architecture (EDA)	Pode facilitar a escalabilidade horizontal (NFR-1.9 e NFR-1.10) e aumentar a capacidade do sistema de lidar com grandes volumes de transações simultâneas, além de permitir resiliência e disponibilidade (NFR-1.2, NFR-1.6, NFR-1.4) distribuindo a carga entre diferentes serviços. A separação de componentes pode também facilitar reportar métricas e atingir requisitos de observabilidade (NFR-1.11).	A comunicação assíncrona e distribuída pode introduzir dificuldades em garantir precisão (NFR-1.1) e confiabilidade estrita nos relatórios (NFR-1.3, NFR-1.5), gerando discrepâncias nos dados em tempo real. Também aumenta a complexidade de coordenação, debug e testes, o que pode prejudicar manutenibilidade e auditabilidade caso não haja uma boa estratégia de versionamento e rastreamento de eventos.

Fonte: Elaborado pelo autor.

Tabela 6 - Resposta final do LLM com as decisões arquiteturais (continuação)

Decisão	Vantagens	Desvantagens
CQRS (Command Query Responsibility Segregation) + Event Sourcing	Facilita a auditabilidade (NFR-1.12) e consulta histórica (NFR-1.13) por armazenar o histórico completo de eventos. Pode ajudar a gerenciar o alto throughput (NFR-1.9, NFR-1.10) isolando comandos de leituras, melhorando disponibilidade e escalabilidade na geração de relatórios com dados antigos.	Aumenta a complexidade do modelo de dados e exige infraestrutura adicional. A recomposição do estado pode afetar desempenho se não for bem projetada, dificultando atender metas estritas de tempo para relatórios (NFR-1.2, NFR-1.4, NFR-1.6). Além disso, garantir precisão com cálculos financeiros ao reconstituir estados a partir de eventos pode ser trabalhoso e suscetível a erros se não houver validação rigorosa.
Padrões de Performance (Caching, Circuit Breaker) e Otimização de Latência	Uso de cache pode reduzir tempo de resposta dos relatórios imediatos (NFR-1.2, NFR-1.6), e circuit breakers podem melhorar a disponibilidade sob falhas de serviços externos. Isso ajuda a atingir metas de desempenho.	O cache pode introduzir discrepâncias entre o estado real e o estado retornado, comprometendo a precisão (NFR-1.1) e a confiabilidade (NFR-1.3, NFR-1.5), principalmente em relatórios financeiros sensíveis. Circuit breakers e políticas de fallback podem devolver dados desatualizados ou incompletos, prejudicando a consistência dos relatórios.

Fonte: Elaborado pelo autor.

Após todo o contexto adquirido, o modelo responde atendendo às solicitações exatas feitas no *prompt*, seguindo uma metodologia adaptada do *ATAM* com análise de *trade-offs*, e referenciando de maneira breve quais atributos de qualidade são atendidos por quais decisões. Os resultados são mais bem discutidos na próxima seção.

5.1 Discussão

A primeira observação que é possível retirar nos resultados do *LLM*, é que existe uma solução interessante para o problema levantado por Eisenreich, Speth e Wagner (2024) e que foi citado no Capítulo 1 deste trabalho, em que basicamente os autores enfatizam que os engenheiros de software tendem a escolher apenas padrões e estilos arquiteturais que eles já trabalharam em algum momento, descartando decisões que se encaixariam melhor no projeto. Nos resultados, apareceram diversas decisões arquiteturais novas e diferentes das que foram mostradas na Seção 4.4, que mostra a solução real implementada. Sendo que, todas as decisões arquiteturais existem e geralmente são abordagens consideradas no ambiente real de desenvolvimento de software, não teve nenhuma alucinação da parte do modelo de *LLM* sobre padrões ou estratégias que não existem.

Isso enfatiza o fato de que, podemos utilizar IA tanto para acelerar o processo de tomada de decisão arquitetural, quanto para extrair novas ideias que antes não estavam claras ou que não foram discutidas por um cenário de arquitetura emergente, principalmente em ambientes ágeis de software.

Quanto a qualidade da resposta pensando nos *trade-offs*, é possível perceber que o modelo errou pouco, pois ele conseguiu tanto agrupar diversas categorias de atributos de qualidade de maneira muito parecida de como foi feita na solução real apresentada neste trabalho; quanto citar *trade-offs* que são realmente relevantes. Um exemplo é o padrão CQRS, que o modelo cita que isso facilitaria a auditabilidade e a consulta histórica, podendo também ajudar em disponibilidade e escalabilidade; sendo que esses pontos foram exatamente os mesmos pontos enfatizados na solução real na Seção 4.4, sendo que outro fato surpreendente é que ele cita alguns atributos afetados que também são idênticos aos citados como vantagem na solução real, como o NFR-1.9 e NFR-1.10.

Um exemplo claro da qualidade da resposta é a menção sobre *Caching*, que também cita os mesmos NFR-1.2 e NFR-1.6 como sendo beneficiados por essa decisão arquitetural, tanto no modelo de *LLM*, como na solução real.

Outro ponto que se pode perceber é que nem todas as decisões citadas na solução real aparecem na resposta do modelo, principalmente os padrões menores que ainda afetam decisões arquiteturais. Um exemplo claro é sobre o NFR-1.1 que fala sobre precisão, o modelo acaba citando de maneira genérica que algumas técnicas de *DDD* podem ajudar, mas não fala exatamente quais técnicas. O *Money Pattern* costuma aparecer em estudos relacionados a *DDD*, de fato; mas a solução real apresentada cita esse padrão de forma específica, enquanto o modelo de *LLM* deixa mais genérico. Apesar de ser uma desvantagem, se for pensar que alguns engenheiros podem nem conhecer técnicas de *DDD*, só o fato de o modelo citar de maneira genérica, já é algo que poderia gerar um tipo de *insight* para buscar mais sobre o tema. Inclusive elencando o exemplo sobre *Caching* novamente, o modelo fala de forma genérica sobre adoção dele, enquanto na solução real foi optado um padrão específico de *Cache Write-Through Pattern*.

Durante os experimentos deste trabalho, percebeu-se que o uso de GPT-4o1 fez muita diferença. Como já citado algumas vezes, a capacidade do modelo de utilizar técnicas de *Chain of Thought* internamente acaba sendo essencial para esse tipo de tarefa complexa envolvendo tomada de decisões arquiteturais e o modelo depender de tanto contexto e conhecimento de diferentes fontes para dar respostas assertivas. Quando foi testado durante o desenvolvimento deste trabalho o uso de modelos como o GPT-4 ou GPT-3 para executar as mesmas tarefas, pela falta dessa técnica, as respostas não foram boas.

Um aspecto não tão positivo é que, como mencionado na Seção 4.5, o processo de estruturação de *prompts* requer um conhecimento de técnicas que ainda estão sendo descobertas e moldadas, até porque o assunto de *LLMs* ainda é algo recente. Ou seja, existem poucos conteúdos sobre como estruturar um *prompt* de uma maneira ideal; e quando não se conhecem essas técnicas, as primeiras respostas parecem desanimadoras, podendo então desmotivar engenheiros ou arquitetos a quererem buscar mais sobre o tema e acharem que os *LLMs* ainda são muito imaturas. Para melhorar esse aspecto, seria interessante ver novos produtos ou sistemas que encapsulam toda a dificuldade de ter que aprender como estruturar um *prompt* para tomar decisões arquiteturais, ou então, fazer com que essas técnicas sejam cada vez mais difundidas tanto no ambiente corporativo, como acadêmico.

Inclusive, ainda comentando sobre produtos ou sistemas que poderiam encapsular e automatizar a tarefa de avaliação de decisões arquiteturais, uma coisa que facilitaria muito seria a criação de algum tipo de DSL que também ajudasse a estruturar a forma de pensar e descrever requisitos funcionais e atributos de qualidade, porque em geral, boa parte do motivo pelo qual a qualidade da resposta foi muito boa, é porque os requisitos estavam bem estruturados; algo que acontece poucas vezes em ambientes reais corporativos. Talvez nesse aspecto o método completo proposto por Eisenreich, Speth e Wagner (2024) possa ser efetivo, pois ele propõe uma iteração completa que conta com a ajuda da IA para estruturar esses requisitos. Inclusive usar a própria IA generativa para coletar, juntar e transformar conhecimento arquitetural desestruturado encontrado no ambiente corporativo em informações estruturadas também poderia ser um excelente caminho.

6 CONSIDERAÇÕES FINAIS

O uso de IA generativa, mais especificamente, os *LLMs*, tem empolgado diversos entusiastas na área justamente pelo seu aprendizado de contexto. Algo que neste trabalho, foi possível analisar acontecendo na prática.

Este trabalho é, de certa forma, uma validação e um complemento de uma etapa proposta no método de geração semiautomática de candidatos à Arquitetura de Software, que é o trabalho proposto por Eisenreich, Speth e Wagner (2024), enfatizando que o tema é promissor e pode trazer bons frutos. Quanto aos resultados que esperava-se alcançar fazendo o estudo neste trabalho, eles foram positivos e trazem alguns complementos ao trabalho dos autores, pois utiliza-se de um software real e ainda utiliza um modelo recente de *LLM* que possui técnicas novas na área. Muitas técnicas de *LLM* ainda estão em processo de estruturação e há muito espaço para novos estudos nessa área. Apesar das análises evidenciarem resultados interessantes, a IA vem apenas para facilitar e automatizar parte do trabalho, sendo que intervenções humanas ainda são completamente necessárias para evitar alucinações.

A área de Arquitetura de Software costuma ter muito conhecimento arquitetural e contexto específico aplicado para cada situação, pois é o que diferencia um software do outro. Essa característica impulsiona o uso de *LLMs* na área, mas as respostas devem ser analisadas cautela para tomar decisões assertivas.

Como possíveis trabalhos futuros, pode-se mencionar as seguintes oportunidades:

- Poderia ser testado outras etapas do método do trabalho de Eisenreich, Speth e Wagner (2024), ou testar em diferentes situações e softwares existentes;
- O uso de *LLMs* para Arquitetura de Software parece se encaixar muito bem, pode-se buscar maneiras diferentes sobre como melhorar processos arquiteturais envolvendo essas novas técnicas;
- As técnicas de *LLMs* ainda são muito novas e pouco conhecidas, pode-se existir trabalhos de treinamento no ambiente corporativo e acadêmico ou ferramentas e abstrações que facilitem o uso e aplicação de técnicas, como o

uso de *DSL* para encapsular formatos específicos de escrita textual dos atributos de qualidade.

REFERÊNCIAS

- AWS. Caching patterns - Database Caching Strategies. **AWS Docs**, 2024. Disponível em: <<https://docs.aws.amazon.com/whitepapers/latest/database-caching-strategies-using-redis/caching-patterns.html>>. Acesso em: 05 Dezembro 2024.
- BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. 3^a. ed. [S.I.]: Addison-Wesley Professional, 2013. 45;80-87 p.
- DE DIANA, M. J. D. O.; KON, F.; GEROSA, M. A. Conducting an Architecture Group in a Multi-team. **Department of Computer Science – University of São Paulo (USP)**, São Paulo, 2010.
- DHAR, R.; VAIDHYANATHAN, K.; VARMA, V. Can LLMs Generate Architectural Design Decisions? - An Exploratory Empirical Study. **2024 IEEE 21st International Conference on Software Architecture (ICSA)**, 4 Março 2024.
- EISENREICH, T.; SPETH, S.; WAGNER, S. From Requirements to Architecture: An AI-Based Journey to Semi-Automatically Generate Software Architectures. **Designing '24: 1st International Workshop on Designing Software**, Lisbon Portugal, 25 Julho 2024.
- FAIRBANKS, G.; GARLAN, D. **Just Enough Software Architecture: A Risk-Driven Approach**. [S.I.]: Marshall & Brainerd, 2010. 34-48;292-312 p.
- FOWLER, M. Money. **Martin Fowler**, 2003. Disponível em: <<https://martinfowler.com/eaaCatalog/money.html>>. Acesso em: 12 Dezembro 2024.
- FOWLER, M. CQRS. **Martin Fowler**, 2011. Disponível em: <<https://martinfowler.com/bliki/CQRS.html>>. Acesso em: 12 Dezembro 2024.
- FOWLER, M. DDD Aggregate. **Martin Fowler**, 2013. Disponível em: <https://martinfowler.com/bliki/DDD_Aggregate.html>. Acesso em: 12 Dezembro 2024.
- KNEUSEL, R. T. **Como a Inteligência Artificial Funciona: Da Magia à Ciência**. 1^º. ed. [S.I.]: Novatec Editora, 2024. 207-248 p.
- MICROSOFT. Publisher-Subscriber pattern. **Microsoft Learn**, 2024. Disponível em: <<https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber>>. Acesso em: 12 Dezembro 2024.
- MONGODB. A Guide to ACID Properties in Database Management Systems. **MongoDB**, 2024. Disponível em: <<https://www.mongodb.com/resources/basics/databases/acid-transactions>>. Acesso em: 12 Dezembro 2024.
- OPENAI. Learning to Reason with LLMs. **OpenAI**, 2024. Disponível em: <<https://openai.com/index/learning-to-reason-with-langs/>>. Acesso em: 24 Novembro 2024.
- OPENAI. Prompt engineering. **OpenAI Platform**, 2024. Disponível em: <<https://platform.openai.com/docs/guides/prompt-engineering>>. Acesso em: 12 Dezembro 2024.
- OZKAYA, I. Can Architecture Knowledge Guide Software Development With Generative AI? **IEEE Software**, Outubro 2023.

APÊNDICE A – PROMPT E SAÍDA COMPLETA DO LLM

Você é um arquiteto de software. Sua tarefa é me fazer enxergar pontos que talvez eu não iria ponderar na hora de tomar decisões sobre um projeto. Pense passo a passo antes de responder:

1- Colete quais são os estilos e padrões arquiteturais de software baseado em referências acadêmicas importantes da área (utilize livros clássicos e modernos conhecios, como Software

Architecture In Practice, Fundamentals of Software Architecture, Patterns of Enterprise Application

Architecture e afins), sabendo que existe diferença entre padrão e estilo arquitetural de software.

Seja criativo na busca.

2- Colete as vantagens e desvantagens de cada um desses estilos e padrões, focando

principalmente em quais atributos de qualidade (também chamado de requisito não funcional) cada um desses estilos e padrões ajudam (ou atrapalham).

3- Leia o modelo de domínio que vou te mandar em formato de PlantUML em texto para entender o contexto do problema.

4- Leia os requisitos que vou te mandar em formato de CSV, onde tem as seguintes colunas, a

coluna mais importante é a de "Métrica de sucesso para atingir o Atributo de Qualidade (Fitness Function)", que é um objetivo metrificando um pouco mais o requisito não funcional, que está na coluna "Categoria".

5- Baseado em todo o contexto prévio que você adquiriu em nossa conversa, me traga estilos e padrões que eu poderia usar ou ponderar na hora de tomar decisões sobre a solução técnica para esse problema.

6- Formate a resposta utilizando a fase 4 do método ATAM (Architecture Tradeoff Analysis Method), focando somente em uma etapa: identificação de trade-offs. Você dirá qual é o estilo/padrão e tradeoffs tenho para o meu contexto.

7- Diga de forma resumida se entendeu e nesse prompt, foque em buscar contexto para os pontos

1 e 2. Vou te enviar o CSV e o modelo de domínio em PlantUML no próximo prompt, só me dê a resposta formatada do passo 6 após eu te enviar CSV e modelo em PlantUML.

Thought about software architecture trade-offs por dois segundos

Entendido. Abaixo segue meu entendimento e a busca de contexto para os pontos 1 e 2 (ainda não respondendo o passo 6, conforme solicitado):

Contexto Buscado – Passo 1 (Coletar Estilos e Padrões Arquiteturais):

Em termos de literatura clássica e moderna sobre Arquitetura de Software, algumas obras de referência incluem:

- “Software Architecture in Practice” (Bass, Clements, Kazman)
- “Fundamentals of Software Architecture” (Mark Richards, Neal Ford)
- “Patterns of Enterprise Application Architecture” (Martin Fowler)
- “Documenting Software Architectures: Views and Beyond” (Clements, et al.)
- “Enterprise Integration Patterns” (Gregor Hohpe, Bobby Woolf)

A partir dessas referências, podemos destacar diferentes estilos e padrões arquiteturais:

Estilos Arquiteturais (Architectural Styles):

1. Camadas (Layered Architecture)
2. Hexagonal/Ports and Adapters
3. Arquitetura em Microservices
4. Arquitetura Monolítica
5. Arquitetura Orientada a Serviços (SOA)
6. Arquitetura Baseada em Eventos (Event-Driven Architecture - EDA)
7. Arquitetura Orientada a Mensagens (Message-Oriented Middleware - MOM)
8. Arquitetura Pipeline/Filter
9. Arquitetura Baseada em Componentes (CBA)

Padrões Arquiteturais (Architecture Patterns):

Do ponto de vista de padrões, muitas vezes são vistos como soluções para problemas recorrentes de design dentro de um estilo. Alguns exemplos:

1. **Model-View-Controller (MVC)**
2. **Model-View-Presenter (MVP)**
3. **Model-View-ViewModel (MVVM)**
4. **Domain-Driven Design (DDD) patterns** (Aggregate, Domain Services, Bounded Context, Repository)
5. **Padrões de Integração Empresarial** (como o “Saga Pattern”, “Event Sourcing”, “CQRS”, “Publish/Subscribe”, “Anti-corruption Layer”, “Strangler Fig”, etc.)
6. **Padrões de Otimização de Performance** (Caching, Circuit Breaker, Bulkhead, etc.)

Contexto Buscado – Passo 2 (Vantagens, Desvantagens, Atributos de Qualidade):

Abaixo seguem vantagens e desvantagens de alguns estilos e padrões, bem como quais atributos de qualidade impactam:

Estilos Arquiteturais

1. Camadas (Layered Architecture)

- **Vantagens:**
 - Separação de responsabilidades clara.
 - Facilita a manutenção e substituição de camadas.
 - Bom para aplicações monolíticas tradicionais.
- **Desvantagens:**
 - Pode gerar camadas desnecessárias, aumentando latência e complexidade.
- **Atributos de Qualidade Afetados:**
 - **Manutenibilidade:** melhora, pois cada camada tem responsabilidade específica.
 - **Testabilidade:** melhora, pois cada camada pode ser testada isoladamente.
 - **Performance:** pode piorar pela sobrecarga de chamadas entre camadas.

2. Hexagonal / Ports and Adapters

- **Vantagens:**
 - Desacoplamento entre domínio e infraestrutura.
 - Facilidade de troca de tecnologias (adapters).
- **Desvantagens:**
 - Configuração inicial mais complexa.

- **Atributos de Qualidade Afetados:**
 - **Flexibilidade e Evolutividade:** melhora, pois é fácil substituir componentes externos.
 - **Testabilidade:** melhora, pois o domínio pode ser testado isoladamente.
 - **Complexidade Inicial:** pode aumentar, dificultando o aprendizado.

3. Microservices Architecture

- **Vantagens:**
 - Escalabilidade independente de serviços.
 - Permite deploy e evolução independentes.
- **Desvantagens:**
 - Aumento da complexidade de comunicação e orquestração.
 - Dificuldade de troubleshooting e observabilidade.
- **Atributos de Qualidade Afetados:**
 - **Escalabilidade e Resiliência:** melhoram pela separação em serviços autônomos.
 - **Desempenho:** pode variar; a comunicação em rede pode aumentar latência.
 - **Manutenibilidade:** pode melhorar em serviços bem isolados, mas a complexidade global cresce.

4. Monolítica

- **Vantagens:**
 - Simplicidade de desenvolvimento inicial.
 - Facilidade de testes end-to-end, pois tudo está em um único deploy.
- **Desvantagens:**
 - Escalabilidade limitada, geralmente escalado como uma unidade.
 - Dificuldade em introduzir novas tecnologias e alterar partes isoladas.
- **Atributos de Qualidade Afetados:**
 - **Performance:** pode ser boa internamente (chamadas internas muito rápidas),
 - **Manutenibilidade:** tende a piorar conforme a base cresce e se torna complexa.
 - **Escalabilidade:** limitada, pois escala-se o todo em vez de partes específicas.

5. SOA (Service-Oriented Architecture)

- **Vantagens:**
 - Reuso de serviços.
 - Padronização de interfaces, interoperabilidade.

- **Desvantagens:**
 - Semelhante a microservices, mas pode levar a um “ESB” (Enterprise Service Bus) pesado e complexo.
- **Atributos de Qualidade Afetados:**
 - **Reusabilidade:** alta, serviços padronizados podem ser reutilizados.
 - **Complexidade:** pode aumentar com governança de serviços.
 - **Escalabilidade:** melhor que um monólito, mas potencialmente menos flexível que microservices.

6. Event-Driven Architecture (EDA)

- **Vantagens:**
 - Alta desacoplagem entre produtores e consumidores de eventos.
 - Sistemas reativos, escaláveis, facilitando a implementação de integrações complexas.
- **Desvantagens:**
 - Dificuldade de rastrear fluxos de eventos.
 - Raciocínio mais complexo sobre estado do sistema.
- **Atributos de Qualidade Afetados:**
 - **Escalabilidade e Resiliência:** melhora, pois a comunicação assíncrona é natural.
 - **Testabilidade e Depurabilidade:** pode piorar pela dificuldade em reproduzir cenários complexos de eventos.

7. Message-Oriented Middleware (MOM)

- **Vantagens:**
 - Desacoplamento temporal, sistemas podem se comunicar assincronamente.
 - Tolerância a falhas melhorada (mensagens persistidas).
- **Desvantagens:**
 - Complexidade de gerenciamento de filas e tópicos.
 - Potencial latência acrescida.
- **Atributos de Qualidade Afetados:**
 - **Resiliência e Escalabilidade:** melhoram com filas assíncronas.
 - **Latência e Tempo de Resposta:** podem piorar devido à comunicação assíncrona.

8. Pipeline/Filter

- **Vantagens:**
 - Facilidade de entender o fluxo de dados.
 - Possibilidade de paralelização e substituição de etapas do pipeline.
- **Desvantagens:**
 - Pode não lidar bem com estados complexos.
 - Ajuste fino de performance pode ser complicado.
- **Atributos de Qualidade Afetados:**
 - **Manutenibilidade e Composabilidade:** melhora, pois etapas são modulares.
 - **Desempenho:** pode ser bom se bem projetado, mas cada estágio adiciona overhead.

9. Arquitetura Baseada em Componentes (CBA)

- **Vantagens:**
 - Reuso e substituição de componentes.
 - Modularidade clara.
- **Desvantagens:**
 - Necessidade de padronização clara de interfaces entre componentes.
 - Potencial de sobrecarga caso não haja uma orquestração adequada.
- **Atributos de Qualidade Afetados:**
 - **Reusabilidade e Evolutividade:** melhoram com componentes bem definidos.
 - **Complexidade de Integração:** pode aumentar, demandando mais governança.

Padrões Arquiteturais

1. MVC/MVP/MVVM

- **Vantagens:**
 - Separação de responsabilidades (UI, lógica e dados).
 - Facilidade de teste de lógica separada da interface.
- **Desvantagens:**
 - Pode introduzir sobrecarga com camadas extras se o projeto for simples.
- **Atributos de Qualidade Afetados:**
 - **Manutenibilidade e Testabilidade:** melhoram, pois a lógica não está misturada à UI.
 - **Complexidade:** pode aumentar, dependendo da granularidade.

2. Domain-Driven Design Patterns (Aggregate, Repository, etc.)

- **Vantagens:**
 - Foco no domínio do negócio, modelos ricos e expressivos.
 - Enfatiza clareza do modelo e linguagem ubíqua.
- **Desvantagens:**
 - Curva de aprendizado maior.
 - Pode ser complexo demais para domínios simples.
- **Atributos de Qualidade Afetados:**
 - **Manutenibilidade, Entendimento do Domínio:** melhoraram consideravelmente.
 - **Performance:** pode ser impactada se o modelo for pesado ou mal projetado.

3. Enterprise Integration Patterns (ex: Saga, Event Sourcing, CQRS)

- **Vantagens:**
 - Soluções comprovadas para problemas complexos de integração e consistência de dados.
 - Permitem lidar com eventual consistency, escalabilidade e resiliência.
- **Desvantagens:**
 - Aumentam a complexidade de design e implementação.
 - Podem exigir ferramentas e infraestrutura específicas.
- **Atributos de Qualidade Afetados:**
 - **Escalabilidade, Tolerância a Falhas, Consistência Eventual:** podem melhorar.
 - **Complexidade, Testabilidade:** podem piorar pelo aumento da complexidade.

4. Padrões de Performance (Caching, Circuit Breaker, etc.)

- **Vantagens:**
 - Melhoria de desempenho (caching).
 - Aumento de resiliência frente a falhas transitórias (circuit breaker).
- **Desvantagens:**
 - Gestão do estado do cache e validade dos dados pode ser complexa.
 - Circuit breaker pode introduzir latências adicionais se mal configurado.
- **Atributos de Qualidade Afetados:**
 - **Performance (Latência e Throughput):** melhora com caching.
 - **Resiliência:** melhora com circuit breaker.
 - **Complexidade Operacional:** aumenta, pois requer monitoramento e tuning.

Resumo de Entendimento:

- Neste prompt, estou entendendo que preciso levantar um contexto inicial sobre estilos e padrões de arquitetura, tirando referências da literatura e apontando suas vantagens e desvantagens, bem como quais atributos de qualidade são afetados.
- O objetivo é preparar o terreno para, no próximo prompt, receber o modelo de domínio em PlantUML e o CSV de requisitos, para então formular as recomendações arquiteturais e fazer a análise de trade-offs (passo 6 do método ATAM) no contexto específico do problema.
- Por ora, a tarefa era apenas buscar contexto (1 e 2) e, por fim, dizer de forma resumida se entendi e focar nesses pontos. Sim, entendi: estou preparando a base teórica para a análise posterior.

```

@startuml
skinparam componentStyle uml2

package SistemaDelnadiimplencia {

    class Saldolnadiimplencia {
        + saldo: Decimal
        + dataDeReferencia: Date
    }

    class Clientelnadimplente {
        + documento: String
    }

    class Dvida {
        + valorBruto: Decimal
        + valorLiquido: Decimal
        + dataSurgimento: Date
        + motivoDaDvida: String
    }

    class Pagamento {
        + valorPago: Decimal
        + horaPagamento: DateTime
    }
}

```

```

class RelatorioFotoInadimplencia {
}

class RelatorioInadimplenciaDoCliente {
}
}

Clientelnadimplente "1" *--> "1..*" Divilda
Clientelnadimplente "1" *--> "1..*" Saldolnadimplencia

Divilda "1" -- "0..*" Pagamento

RelatorioFotoInadimplencia "1" o-- "1..*" Clientelnadimplente
RelatorioInadimplenciaDoCliente "1" o-- "1" Clientelnadimplente
@enduml

```

Thought about domain model structure por 4 segundos ^

Honing tradeoffs

The user is working on ATAM step 4, emphasizing tradeoffs. They are awaiting a CSV file to complete the analysis. The ATAM outcome will be delivered after the CSV is provided.

Working through the steps

OK, let me see. I've examined the domain model and now I'm waiting for the CSV to complete the analysis. I'm focusing on understanding and gathering context for points 1 and 2.

Piecing it together

Received the UML model and piecing together steps 5 and 6. Waiting for the CSV file to finish the analysis in the next conversation.

Entendido. A partir do diagrama de domínio em PlantUML, vejo um contexto voltado para um sistema de inadimplência, onde temos entidades como **Clientelnadimplente**, **Divilda**, **Pagamento**, **Saldolnadimplencia**, além de relatórios específicos para acompanhar a situação desses clientes. Isso sugere um domínio voltado a gestão financeira, tracking de dívidas, pagamentos e geração de relatórios.

Focando ainda nos pontos 1 e 2 (levantamento de contexto sobre estilos e padrões arquiteturais e seus impactos em atributos de qualidade), comprehendo que o domínio apresentado pode vir a se beneficiar de padrões e estilos que facilitem a manutenção, escalabilidade e testabilidade, principalmente considerando que o domínio lida com regras de negócio sensíveis (cálculos de saldo, valores pagos, surgimento de dívidas ao longo do tempo). Por exemplo:

- Estilos mais modulares (como Arquitetura em Camadas ou Hexagonal) podem ajudar na manutenibilidade e testabilidade.
- Padrões como DDD podem ajudar a capturar as nuances do domínio de inadimplência de forma clara, reduzindo complexidade a longo prazo.
- Caso haja necessidade de escalabilidade e distribuição, padrões de microservices ou event-driven podem ser considerados, embora aumentem a complexidade operacional.

Em resumo, compreendi o contexto do domínio e continuo focado em entender como as diferentes abordagens arquiteturais (pontos 1 e 2) podem se alinhar com os requisitos não funcionais, preparando o terreno para, após receber o CSV (com requisitos e métricas de sucesso), propor uma análise de trade-offs conforme solicitado.

ID do Requisito;Descrição;Categoria;Métrica de sucesso para atingir o Atributo de Qualidade (Fitness Function);Dependência;.....
 FR-1;O sistema deve gerenciar a inadimplência dos clientes;Funcional;.....
 FR-1.1;Um cliente inadimplente possui um documento (CPF ou CNPJ), o conjunto de suas dívidas e seus respectivos pagamentos realizados para essas dívidas;Funcional;.....
 FR-1.2;Uma dívida é caracterizada por: valor bruto, valor líquido (após todos os pagamentos realizados), data de surgimento da dívida, motivo da dívida;Funcional;.....
 FR-1.3;Um pagamento para uma dívida é caracterizada por: valor pago e hora do pagamento;Funcional;.....
 FR-1.4;Saldo de inadimplência é a soma de valor líquido de todas as dívidas de todos os clientes inadimplentes, agrupado por dia, é caracterizado por: saldo, data de referência, documento do cliente;Funcional;.....
 FR-1.5;Inadimplência de um cliente é diferença entre a soma de valor bruto e a soma de valor líquido de todas as dívidas, agrupado por dia e por documento do cliente inadimplente, é caracterizado por: saldo, data de referência, documento do cliente, todas as dívidas do cliente até a data de referência, todos os pagamentos

do cliente até a data de referência;Funcional;.....
NFR-1.1;Erros em cálculos financeiros envolvendo dívidas e pagamentos precisam ser igual ou inferior a um milionésimo (0,000001), principalmente para manter o saldo de inadimplência correto;Precisão;Taxa de erro em operações matemáticas envolvendo decimais $\leq 10^{-6}$;.....
FR-1.6;O sistema deve permitir o acesso ao histórico de clientes inadimplentes;Funcional;.....
FR-1.7;O sistema deve permitir o acesso ao histórico de dívidas para clientes inadimplentes;Funcional;.....
FR-1.8;O sistema deve permitir o acesso ao histórico de pagamentos de dívidas;Funcional;.....
FR-1.9;"O sistema deve emitir um relatório chamado ""Fotoladimplencia"" com o saldo de inadimplência de acordo com o dia solicitado, que é uma data de referência";Funcional;.....
NFR-1.2;"O relatório de ""Fotoladimplencia""", quando é solicitado em D-0 em relação ao dia atual do sistema como data de referência deve estar disponível em até 2 minutos após solicitado a sua emissão";Disponibilidade;Tempo de resposta ≤ 2 minutos;.....
NFR-1.3;"O relatório de ""Fotoladimplencia""", quando é solicitado em D-0 em relação ao dia atual do sistema como data de referência pode conter divergências de até 1 milhão em relação ao saldo da inadimplência real final do próprio dia";Confiabilidade; (Saldo da inadimplência real - Saldo da inadimplência reportado) ≤ 1 milhão;.....
NFR-1.4;"O relatório de ""Fotoladimplencia""", quando é solicitado em D-1 em relação ao dia atual do sistema como data de referência deve estar disponível em até 40 minutos após solicitado a sua emissão";Disponibilidade;Tempo de resposta ≤ 40 minutos;.....
NFR-1.5;"O relatório de ""Fotoladimplencia""", quando é solicitado em D-1 em relação ao dia atual do sistema como data de referência não pode ter divergências relação ao saldo da inadimplência real final do próprio dia";Confiabilidade; (Saldo da inadimplência real - Saldo da inadimplência reportado) = 0;.....
FR-1.10;"O sistema deve emitir um relatório chamado ""InadimplenciaDoCliente-X""", contendo a inadimplência de um cliente e informando de maneira detalhada todas as dívidas e pagamentos desse cliente (com todas as características de dívidas e pagamentos);Funcional;.....
NFR-1.6;"O relatório de ""InadimplenciaDoCliente-X""", quando é solicitado em D-0 em relação ao dia atual do sistema como data de referência deve estar disponível em até 1 minuto após solicitado a sua emissão";Disponibilidade;Tempo de resposta ≤ 1 minuto;.....
NFR-1.7;O sistema deve suportar a persistência de novas dívidas em até 500 milissegundos;Desempenho;Tempo de resposta $\leq 500ms$;.....
NFR-1.8;O sistema deve permitir o acesso ao histórico de novos pagamentos em até 400 milissegundos;Desempenho;Tempo de resposta $\leq 400ms$;.....
NFR-1.9;O sistema deve suportar a persistência concorrente de até 50 novas dívidas;Capacidade;Throughput ≥ 50 transações

simultâneas;.....
 NFR-1.10;O sistema deve suportar o registro persistente simultâneo de até 600 novos pagamentos;Capacidade;Throughput ≥ 600 transações simultâneas;.....
 NFR-1.11;O sistema deve emitir métricas sobre Throughput, Tempo de resposta, Latência de todas as dependências de I/O;Observabilidade;Métricas exibidas em uma plataforma qualquer de Observabilidade;.....
 NFR-1.12;O sistema deve manter o histórico de dívidas, pagamentos e clientes inadimplentes por 6 anos para fins de auditoria;Auditabilidade;Dados históricos sobre dívidas, pagamentos e clientes inadimplentes disponíveis em qualquer plataforma por até 6 anos;.....
 NFR-1.13;O histórico solicitado dentro desse período de 6 anos para fins de auditoria pode ser emitido em até 5 dias;Disponibilidade;Tempo de resposta ≤ 5 dias;.....

Pronto para análise de trade-offs conforme solicitado. ^

Piecing together trade-offs

Now I'm weaving through the CSV and domain model to pinpoint style and pattern trade-offs, ensuring clarity in how they integrate with the overall architecture.

Mapping architectural choices

I'm aligning high concurrency, precision, and performance constraints with microservices and domain-driven design, while ensuring good observability and auditability. The goal is to pinpoint the optimal setups for these needs.

Mapping architectural paths

OK, let me see. I'm charting microservices and domain-driven design to address high concurrency and performance constraints. The aim is to pinpoint the best setup for these needs.

Weighing options

I'm exploring microservices for scalability and performance. They're complex and have overheads, but simpler monoliths might ensure precision initially with fewer distributed issues.

Navigating architectural challenges

I'm evaluating DDD with hexagonal architecture for precision versus complexity. Also, weighing CQRS with event sourcing for concurrency and auditability against simpler CRUD.

Passo 6 do ATAM – Identificação de Trade-offs

Abaixo seguem alguns estilos e padrões arquiteturais sugeridos para o contexto dado, bem como os principais trade-offs envolvidos considerando os requisitos funcionais e não funcionais identificados no CSV (NFRs relacionadas a precisão, disponibilidade, confiabilidade, desempenho, capacidade, observabilidade, auditabilidade) e o modelo de domínio do sistema de inadimplência:

1. Arquitetura em Camadas + Domain-Driven Design (DDD)

- **Trade-off:**
 - **Prós:** O DDD combinado com uma arquitetura em camadas (Aplicação, Domínio, Infraestrutura) melhora a **manutenibilidade e clareza do domínio**, facilitando garantir a **precisão** (NFR-1.1) nos cálculos financeiros e mantendo regras de negócio coerentes. Também torna mais simples a **auditabilidade** (NFR-1.12), já que o domínio é bem definido e rastreável.
 - **Contras:** Pode aumentar a **complexidade** inicial de desenvolvimento e o **tempo de resposta** (por adicionar camadas), comprometendo parte das metas de **desempenho** (NFR-1.7 e NFR-1.8) se não houver otimizações. Além disso, o acréscimo de camadas pode afetar a **latência** e dificultar atingir metas de tempo estritas para relatórios (NFR-1.2, NFR-1.4, NFR-1.6).

2. Arquitetura Hexagonal (Ports and Adapters)

- **Trade-off:**
 - **Prós:** Ajuda na **flexibilidade** e substituição de fontes de dados ou serviços externos, facilitando a **observabilidade** (NFR-1.11) e o cumprimento de requisitos de **auditabilidade**. Permite manter a **precisão** no domínio central, isolado de infraestruturas que podem introduzir erros.
 - **Contras:** Exige maior esforço de configuração inicial e abstrações adicionais, o que pode afetar **desempenho** em cenários de alto throughput (NFR-1.9, NFR-1.10), já que cada porta/adaptador pode adicionar overhead.

3. Microservices + Event-Driven Architecture (EDA)

- **Trade-off:**
 - **Prós:** Pode facilitar a **escalabilidade horizontal** (NFR-1.9 e NFR-1.10) e aumentar a capacidade do sistema de lidar com grandes volumes de transações simultâneas, além de permitir resiliência e disponibilidade (NFR-1.2, NFR-1.6, NFR-1.4) distribuindo a carga entre diferentes serviços. A separação de componentes pode também facilitar reportar métricas e atingir requisitos de **observabilidade** (NFR-1.11).
 - **Contras:** A comunicação assíncrona e distribuída pode introduzir dificuldades em garantir **precisão** (NFR-1.1) e **confiabilidade** estrita nos relatórios (NFR-1.3, NFR-1.5), gerando discrepâncias nos dados em tempo real. Também aumenta a complexidade de coordenação, debug e testes, o que pode prejudicar **manutenibilidade e auditabilidade** caso não haja uma boa estratégia de versionamento e rastreamento de eventos.

4. CQRS (Command Query Responsibility Segregation) + Event Sourcing

- Trade-off:

- **Prós:** Facilita a **auditabilidade** (NFR-1.12) e **consulta histórica** (NFR-1.13) por armazenar o histórico completo de eventos. Pode ajudar a gerenciar o alto throughput (NFR-1.9, NFR-1.10) isolando comandos de leituras, melhorando disponibilidade e escalabilidade na geração de relatórios com dados antigos.
- **Contras:** Aumenta a **complexidade do modelo de dados** e exige infraestrutura adicional. A recomposição do estado pode afetar **desempenho** se não for bem projetada, dificultando atender metas estritas de tempo para relatórios (NFR-1.2, NFR-1.4, NFR-1.6). Além disso, garantir **precisão** com cálculos financeiros ao reconstituir estados a partir de eventos pode ser trabalhoso e suscetível a erros se não houver validação rigorosa.

5. Padrões de Performance (Caching, Circuit Breaker) e Otimização de Latência

- Trade-off:

- **Prós:** Uso de cache pode reduzir tempo de resposta dos **relatórios imediatos** (NFR-1.2, NFR-1.6), e circuit breakers podem melhorar a **disponibilidade** sob falhas de serviços externos. Isso ajuda a atingir metas de **desempenho**.
- **Contras:** O cache pode introduzir discrepâncias entre o estado **real** e o estado **retornado**, comprometendo a **precisão** (NFR-1.1) e a **confiabilidade** (NFR-1.3, NFR-1.5), principalmente em relatórios financeiros sensíveis. Circuit breakers e políticas de fallback podem devolver dados desatualizados ou incompletos, prejudicando a consistência dos relatórios.

Resumo do Entendimento:

- Há um entendimento claro dos trade-offs entre diferentes estilos/padrões e os requisitos não funcionais levantados. Por exemplo, microservices e EDA melhoram escalabilidade e disponibilidade mas podem prejudicar precisão e confiabilidade imediata.
- DDD com arquiteturas em camadas ou hexagonais melhora manutenibilidade e precisão, mas pode afetar desempenho.
- CQRS e Event Sourcing ajudam na auditabilidade e histórico, mas aumentam complexidade e podem afetar o desempenho de geração de relatórios.
- Padrões de performance (caching) otimizam latência, mas podem comprometer precisão.

Entendido: Esta é a resposta formatada do passo 6 do método ATAM (identificação de trade-offs) considerando o contexto fornecido (modelo de domínio e requisitos do CSV).