Geração de Código DSL para Diagramas C4 Model via LLMs

Murilo Costa[1]

[1] Departamento de Engenharia de Software – Instituto de Ciências Exatas e Informática – Pontifícia Universidade Católica de Minas Gerais (PUC Minas)

murilo.goncalves.1048860@sga.pucminas.br

Abstract. *Translating natural language to architecture DSLs, such as Structurizr, is a core challenge of semantic and structural inference for LLMs. This study evaluates the feasibility and quality of this translation using the Gemini model, comparing 0-shot and one-shot prompting strategies for generating Container (C2) and Component (C3) diagrams. The results indicate two key findings: (1) The one-shot strategy was essential for feasibility (Syntactic Validity Rate - SVR of 67.14%), whereas the 0-shot approach was infeasible (SVR 0.36%); (2) Architectural complexity impacted quality, with high performance on C2 (SVR 91.1%) and an abrupt drop for C3 (SVR 43.2%). This decline was corroborated by a blind qualitative evaluation by experts, which pointed to lower accuracy and completeness for C3. We conclude that artifact complexity is a critical predictor of LLM performance on this task.*

Resumo. *A tradução de linguagem natural para DSLs de arquitetura, como o Structurizr, é um desafio central de inferência semântica e estrutural para LLMs. Este estudo avalia a viabilidade e a qualidade desta tradução usando o modelo Gemini, comparando estratégias de prompting 0-shot e one-shot na geração de diagramas de Contêiner (C2) e Componente (C3). Os resultados indicam dois achados principais: (1) A estratégia one-shot foi essencial para a viabilidade (Taxa de Validade Sintática - TVS de 67,14%), enquanto a 0-shot foi inviável (TVS 0,36%); (2) A complexidade arquitetural impactou a qualidade, com alto desempenho em C2 (TVS 91,1%) e uma queda abrupta em C3 (TVS 43,2%). Esta queda foi corroborada por uma avaliação qualitativa cega de especialistas, que apontou menor acurácia e completude para C3. Conclui-se que a complexidade do artefato é um preditor crítico da performance dos LLMs nesta tarefa.*

Bacharelado em Engenharia de *Software* - PUC Minas
Trabalho de Conclusão de Curso (TCC)

Orientador de conteúdo (TCC I): Joana Souza - joanasouza@pucminas.br
Orientador acadêmico (TCC I): João Pedro Batistele - jp.batisteli@hotmail.com
Orientador de conteúdo (TCC I): Leonardo Vilela - leonardocardoso@pucminas.br
Orientador acadêmico (TCC I): Cleiton Tavares - cleitontavares@pucminas.br
Orientador do TCC II: Danilo de Quadro Maia Filho - 1514571@sga.pucminas.br

Belo Horizonte, 23 de 11 de 2025.

1. Introdução

A Arquitetura de Software define os atributos de qualidade e a evolução de um sistema, tornando sua comunicação clara entre *stakeholders* um fator crítico [Clements et al. 2010, Carducci 2025]. Modelos como C4 e UML facilitam essa comunicação, mas recentemente *Large Language Models* (LLMs) emergiram como ferramentas de apoio, auxiliando na geração de descrições textuais e no design arquitetural [Esposito et al. 2025]. Este trabalho situa-se, portanto, na interseção entre a Arquitetura de Software e a *Generative Artificial Intelligence* (GenAI) aplicada à visualização arquitetural.

Embora LLMs sejam proficientes na geração de texto livre, essa habilidade não se traduz automaticamente na geração de diagramas arquiteturais precisos a partir de descrições em linguagem natural. A lacuna reside na complexidade dessa tradução: a prosa é informal e ambígua [Dhar et al. 2024b], enquanto a linguagem do diagrama de arquitetura é formal e estruturada. O desafio exige, simultaneamente, inferência estrutural, para aderir à notação, e fidelidade semântica, para resolver ambiguidades e intenções implícitas [Esposito et al. 2025, Jahić and Sami 2024]. A falha nesse processo leva a inconsistências e omissões que comprometem o valor do diagrama como ferramenta de comunicação. Dessa forma, o problema central desta pesquisa é a **dificuldade dos LLMs em traduzir descrições de arquitetura formuladas em linguagem natural para uma linguagem estruturada de diagramas arquiteturais**.

A automação da geração de diagramas arquiteturais a partir de linguagem natural promete ganhos significativos. Os benefícios centrais incluem: a redução do esforço manual (custo e tempo) pela geração rápida e repetível de artefatos; e a mitigação de inconsistências, comuns em processos manuais [Jahić and Sami 2024, Dhar et al. 2024b]. Tais ganhos facilitam a comunicação e apoiam a análise da arquitetura. Portanto, avaliar a viabilidade e a qualidade desta automação via LLMs é um passo fundamental para integrá-los ao fluxo de trabalho de Arquitetura de Software, expandindo seu uso além da produção puramente textual [Dhar et al. 2024a].

Para endereçar esse problema, o presente estudo define o seguinte objetivo geral: **avaliar a capacidade de LLMs em traduzir descrições de arquitetura de software em linguagem natural para linguagem estruturada, analisando a viabilidade e a qualidade dos diagramas gerados.** Este objetivo geral se desdobra nos seguintes objetivos específicos: I) Analisar a capacidade dos modelos em gerar código *Domain-Specific Language* (DSL) que seja sintaticamente válido e aderente às especificações a partir de descrições em linguagem natural; II) Avaliar a qualidade dos diagramas gerados em termos de sintática e semântica; III) Investigar o impacto das técnicas de *prompting 0-shot* e *one-shot* na validade e na qualidade dos artefatos gerados.

Como resultados esperados, este estudo prevê a quantificação da viabilidade e da qualidade da geração de diagramas C4 via LLMs. A hipótese central é que técnica *in context-learning* resultará em maior qualidade e viabilidade em relação à técnica sem contexto. Espera-se, no entanto, que mesmo os melhores resultados apresentem limitações, como uma correlação negativa entre a complexidade do diagrama e a performance do modelo. Projeta-se também que a taxonomia de erros identificados aponte para uma maior frequência de falhas lógicas e semânticas em comparação com erros de sintaxe da DSL.

Este trabalho está estruturado da seguinte forma. A Seção 2 apresenta a

fundamentação teórica sobre C4 Model, LLMs, engenharia de *prompt* e o uso de DSLs em arquitetura de software. A Seção 3 revisa os trabalhos relacionados à geração de código estruturado por LLMs. A Seção 4 detalha a metodologia do estudo exploratório, incluindo a seleção de casos e as métricas de avaliação. A Seção 5 apresenta e analisa os resultados. A Seção 6 conclui o trabalho, sumarizando os achados, implicações e limitações. Por fim, a Seção 7 aponta trabalhos futuros.

2. Fundamentação Teórica

Esta seção apresenta os quatro eixos conceituais que fundamentam o estudo: (i) o C4 *Model* como referência hierárquica de representação; (ii) o uso de DSLs na prática de *Architecture as Code*; (iii) as capacidades e limitações dos LLMs em geração estruturada; e (iv) o papel da Engenharia de *Prompts* como técnica de controle contextual.

2.1. O C4 *Model* como Estrutura de Representação Hierárquica

A comunicação arquitetural constitui um problema persistente na engenharia de software [Len Bass 2012]. O C4 *Model*, proposto por Simon Brown [Brown 2022], introduz uma hierarquia de abstrações (Contexto, Contêiner, Componente e Código) que permite visualizar o sistema em diferentes profundidades cognitivas. Essa estrutura torna o modelo não apenas uma ferramenta de documentação, mas também um referencial empírico para mensurar complexidade arquitetural, dado que cada nível adiciona granularidade e dependências semânticas. Essa propriedade o torna particularmente adequado para investigar limitações de inferência hierárquica em LLMs, pois oferece um gradiente controlado de dificuldade — do macro (Contêiner) ao micro (Componente).

2.2. DSLs e o Paradigma *Architecture as Code*

A prática contemporânea de *Architecture as Code* formaliza a representação arquitetural por meio de DSLs, que diferem das *General-Purpose Language*s (GPLs) por encapsularem semânticas restritas e terminologias do domínio [Fowler 2010]. No contexto arquitetural, as DSLs permitem que diagramas sejam definidos programaticamente e versionados como artefatos de código — prática conhecida como *Diagram as Code* (DaC). O *Structurizr* DSL, utilizado neste estudo, traduz os elementos e relações do C4 *Model* para sintaxe textual controlada. Assim, a tarefa de gerar diagramas arquiteturais via LLMs pode ser formulada como uma tradução semântica entre dois espaços linguísticos: um informal (descrições em linguagem natural) e outro formal (DSL). Essa tradução requer não apenas correção sintática, mas também fidelidade semântica — isto é, a capacidade de preservar a estrutura conceitual da arquitetura subjacente.

2.3. LLMs e a Geração de Estruturas Formais

Os LLMs baseiam-se na arquitetura *Transformer* [Vaswani et al. 2023], sendo capazes de capturar padrões linguísticos em larga escala. Essa capacidade os torna promissores para a geração de texto estruturado, incluindo código [Fan et al. 2023]. Entretanto, a tradução de linguagem natural para DSLs arquiteturais requer inferência simbólica e estrutural, competências nas quais LLMs ainda demonstram limitações. Estudos recentes [Esposito et al. 2025, Dhar et al. 2024a] evidenciam que, embora esses modelos dominem a sintaxe e coerência local, falham em manter consistência hierárquica e relações de dependência entre elementos arquiteturais. Portanto, a hipótese teórica subjacente a este

estudo é que a complexidade estrutural do artefato — expressa pelo nível do diagrama C4 — constitui um fator crítico de degradação de desempenho para LLMs, revelando sua incapacidade de raciocínio composicional em domínios formais.

2.4. Engenharia de *Prompts* como Controle de Contexto

A geração de DSLs por LLMs depende fortemente da formulação dos *prompts* — instruções textuais que delimitam o espaço de inferência do modelo. A Engenharia de *Prompts* busca sistematizar esse processo, utilizando abordagens de *in-context learning* como *zero-shot*, *one-shot* e *few-shot prompting* [Brown et al. 2020]. No primeiro caso, o modelo é testado em sua capacidade de generalização sem suporte contextual; no segundo, exemplos explícitos orientam a geração por analogia. Em tarefas que exigem precisão sintática e mapeamento estrutural, como a geração de DSLs arquiteturais, o *one-shot prompting* tende a atuar como uma forma de transferência estrutural local, fornecendo ao modelo padrões gramaticais e semânticos que compensam lacunas de aprendizado simbólico. Assim, comparar essas duas estratégias permite investigar até que ponto o desempenho do LLM depende de orientação contextual explícita.

3. Trabalhos Relacionados

A investigação sobre o uso de LLMs em tarefas de engenharia e modelagem arquitetural tem crescido rapidamente, abrangendo desde o suporte a decisões de design até a geração automatizada de diagramas. Entretanto, a literatura ainda carece de avaliações empíricas quantitativas e replicáveis que mensurem a capacidade dos LLMs de gerar código DSL arquitetural — um tipo de representação formal cuja validade depende tanto da correção sintática quanto da coerência semântica.

3.1. Geração de Diagramas Arquiteturais a partir de Linguagem Natural

Jahić e Sami (2024) investigaram o uso do ChatGPT 3.5 em tarefas de design arquitetural, combinando uma *survey* com 15 empresas e 50 experimentos sobre cinco projetos *open-source*. O objetivo foi gerar representações textuais de diagramas C4 a partir de descrições arquiteturais. A pesquisa revelou que, embora os LLMs aumentem a produtividade de codificação, os diagramas gerados apresentaram baixa reprodutibilidade e inconsistência lógica, com 80% das empresas relatando preocupação com a qualidade. O estudo contribui ao demonstrar a potencialidade exploratória dos LLMs na representação arquitetural, mas limita-se à avaliação qualitativa e à ausência de uma métrica objetiva de validade. Em contraste, o presente trabalho propõe uma avaliação quantitativa sistemática, utilizando métricas de validade sintática e fidelidade estrutural para mensurar o desempenho de geração.

3.2. Geração de DSLs Arquiteturais a partir de Código-Fonte

Shehata et al. (2024) investigaram a geração automática de diagramas UML (PlantUML) a partir de código Java usando ChatGPT 3.5. A precisão foi alta para entidades básicas (90% para classes e atributos), mas degradou fortemente nas relações (66%), especialmente em sistemas complexos. O trabalho evidencia que a complexidade estrutural é um fator crítico de degradação semântica em LLMs, corroborando a hipótese central deste artigo. Contudo, como a entrada foi código-fonte — e não linguagem natural —, o estudo não avalia a etapa mais desafiadora da tradução semântica: a interpretação de requisitos

textuais. Assim, a contribuição do presente estudo está em transpor essa avaliação para o domínio textual, onde o modelo precisa inferir estrutura a partir de descrições informais, o que amplia o desafio cognitivo.

3.3. Interação Humano–LLM no Design Arquitetural

Ahmad et al. (2023) exploraram a geração de diagramas *Unified Modeling Language* (UML) por meio de um processo colaborativo humano-*bot*, no qual arquitetos humanos interagiam iterativamente com o ChatGPT 3.5 durante o desenvolvimento do sistema *CampusBike*. Os autores observaram que os LLMs são úteis como assistentes criativos, mas suas respostas apresentaram superficialidade e inconsistência, exigindo refinamento manual constante. Embora relevante para compreender o potencial de colaboração, o estudo não isola o desempenho autônomo do modelo. Em contraste, o presente trabalho remove a intervenção humana, permitindo mensurar de forma controlada a capacidade intrínseca do LLM em traduzir texto livre em código DSL formal.

3.4. Modelagem Arquitetural Formal e MBSE

Von Heissen et al. (2024) propuseram um método para gerar rascunhos de arquiteturas de sistema a partir de requisitos textuais, integrando LLMs como plugin do *Cameo Systems Modeler* para criar diagramas SysML. A avaliação qualitativa, conduzida por especialistas em *Model-Based Systems Engineering* (MBSE), indicou qualidade limitada e heterogênea: apenas 20% consideraram os componentes funcionais adequados; 60% classificaram a complexidade como parcialmente satisfatória; e 80% avaliaram a rastreabilidade como incompleta. Esses resultados sugerem que LLMs possuem capacidade parcial de gerar modelos coerentes, atuando mais como suporte heurístico do que como ferramenta de geração confiável. O presente estudo difere ao buscar mensurar quantitativamente essa limitação, avaliando a fidelidade estrutural com métricas objetivas, em vez de depender apenas de percepções humanas.

3.5. Geração Progressiva de Artefatos a partir de Requisitos

Wei (2024) introduziu o conceito de *prompting* progressivo, empregando um LLM customizado para gerar design e código orientado a objetos a partir de casos de uso textuais. O método demonstrou ganhos de coerência incremental, mas não abordou a geração de DSLs arquiteturais nem aplicou métricas de desempenho estruturado. O trabalho de Wei é relevante por indicar o potencial de raciocínio iterativo e contextual, aspecto que pode ser estendido à geração hierárquica de diagramas C4. Este estudo complementa essa visão ao propor um protocolo experimental replicável, que quantifica o impacto da complexidade arquitetural sobre o desempenho do LLM.

A literatura converge em duas observações centrais. Primeiro, LLMs demonstram potencial de suporte à engenharia arquitetural, mas carecem de mecanismos de raciocínio hierárquico para manter coerência estrutural à medida que cresce a complexidade. Segundo, as avaliações existentes são majoritariamente qualitativas e não permitem comparações reprodutíveis entre modelos e estratégias de *prompting*. O presente trabalho diferencia-se, portanto, por propor a primeira avaliação empírica sistemática e quantitativa da geração de código DSL arquitetural a partir de linguagem natural, estabelecendo métricas objetivas de validade e qualidade. Essa contribuição posiciona o estudo como um marco metodológico na investigação da inferência estrutural de LLMs no domínio da *Architecture as Code*.

4. Materiais e Métodos

Dada a escassez de trabalhos empíricos sobre a geração de diagramas arquiteturais por LLMs [Esposito et al. 2025], esta pesquisa caracteriza-se como exploratória. Adota-se uma abordagem de métodos mistos (quantitativos e qualitativos) [Runeson et al. 2012]. Esta combinação permite não apenas quantificar objetivamente o desempenho do LLM, mas também interpretar qualitativamente as características e limitações do fenômeno. A metodologia segue o processo estruturado ilustrado na Figura 1 e detalhado nas subseções seguintes.

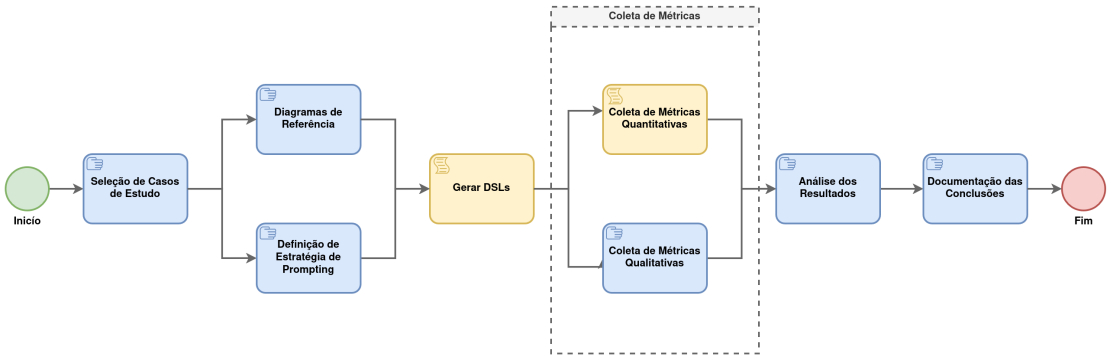

Figura 1. Procedimentos e Etapas da Metodologia

4.1. Questões de Pesquisa e Métricas de Avaliação

Para guiar esta investigação e estruturar a análise dos resultados, foram definidas seis Questões de Pesquisa, cada uma associada a proposições ou hipóteses específicas e a um conjunto de métricas de avaliação. A Tabela 1 consolida o desenho do estudo, conectando os objetivos da análise às métricas quantitativas e qualitativas que serão utilizadas para responder a cada questão.

ID	*Research Question*	Métricas	Hipóteses
RQ1	Qual a viabilidade da geração de DSL sintaticamente válidos por LLMs?	M1 e Taxonomia de Erros	A viabilidade geral será baixa, e a TVS da *0-shot* será inferior à da *one-shot*
RQ2	Qual a fidelidade semântica e estrutural dos diagramas válidos?	M2, M3 e Avaliação Qualitativa	A qualidade dos artefatos gerados são de alta variabilidade
RQ3	Qual a consistência dos diagramas válidos gerados?	M3	Não há diferença significativa na consistência entre os tipos de diagrama
RQ4	Como a complexidade da arquitetura influencia os diagramas gerados?	M1, M2, M3	Não há correlação entre a complexidade e as métricas de performance

Tabela 1. Operacionalização de questões de pesquisa

4.2. Ambiente de Execução

O experimento utilizou o modelo *gemini-2.5-flash*, acessado via *Application Programming Language* (API) oficial. Como o protocolo é executado via API, os requisitos de *hardware* local são mínimos, exigindo apenas Python 3.9 ou superior. A ferramenta *Structurizr CLI* foi empregada para a validação sintática dos códigos DSL gerados e para a renderização dos diagramas.

4.3. Seleção dos Estudo de Casos

A seleção dos casos de estudo iniciou-se com o *dataset* de blogs de Soliman et al. (2023), mas este foi descartado. Os artigos, embora conceitualmente ricos, careciam de especificações técnicas (elementos, interações, escopo) necessárias para a criação de um *ground truth*. Isso representava uma ameaça direta à validade de construção do estudo. Para mitigar este risco, optou-se pelo *Archi Dataset* [Della Pelle 2024], composto por projetos acadêmicos com descrições textuais de requisitos e soluções. Dos 15 projetos disponíveis (V1 e V2), 14 foram selecionados; um (”Nestigo”) foi excluído por possuir documentação em formato binário inacessível. O conjunto final de 14 casos foi, então, classificado por nível de complexidade, resultando em 5 casos de Baixa, 4 de Média e 5 de Alta complexidade.

4.4. Diagramas de Referência

O pesquisador elaborou manualmente os diagramas *Structurizr* de referência. Para cada um dos 14 casos de estudo, foram criados os diagramas de Contêiner (C2) e Componente (C3), totalizando 28 diagramas de referência. Para mitigar o viés do pesquisador nesta etapa, foi adotado um protocolo: os diagramas foram desenvolvidos a partir da identificação sistemática de elementos e relações nas descrições textuais, seguindo os princípios do *C4 Model* [Brown 2022]. O *checklist* oficial do *C4 Model* [c4model.com] foi então utilizado para validar a conformidade dos diagramas de referência com as boas práticas do modelo.

4.5. Procedimento Experimental de Geração de Código

O procedimento experimental comparou duas técnicas de *in-context learning* — *0-shot* e *one-shot* — na geração de código *Structurizr* para diagramas C2 e C3. Na abordagem *one-shot*, seguindo Brown et al. (2020) , incluiu-se no *prompt* um único exemplo de código DSL válido para orientar o padrão sintático. O *prompt 0-shot* (Apêndice A.1) contém apenas a instrução da tarefa e a descrição do caso de estudo, enquanto o *prompt one-shot* (Apêndice A.2) adiciona o exemplo de DSL para fornecer contexto. A execução foi automatizada por um *script* que enviou ambos os *prompts* à *API* do *Gemini* para cada caso de estudo. Para mitigar a variabilidade dos LLMs e avaliar consistência, cada combinação (caso × técnica) foi repetida 20 vezes, e todos os artefatos gerados foram armazenados. A inclusão do exemplo no *one-shot* buscou avaliar a capacidade do modelo de aprender a sintaxe da DSL com uso mínimo de *tokens*, simulando um cenário real de *prompt engineering* conciso.

4.6. Coleta de Dados e Cálculo das Métricas

A coleta de dados combinou automação e validação por especialistas. Primeiramente, um *script* de avaliação processou todos os artefatos de código DSL gerados para determinar

a Taxa de Validade Sintática (TVS). Em seguida, apenas para os artefatos sintaticamente válidos (renderizáveis), o *script* calculou as métricas de qualidade, como F1-Score e GED (Tabela 2). Em complemento, para a interpretação qualitativa, um painel de 3 especialistas conduziu uma avaliação cega, utilizando um questionário estruturado, apresentado no Apêndice A.3, para avaliar uma amostra representativa dos diagramas válidos.

ID	Métrica	Descrição
M1	Taxa de Validade Sintática (TVS)	Percentual de códigos DSL gerados que são sintaticamente corretos e compiláveis.
M2	F1-Score	Medida de acurácia semântica que calcula a média harmônica entre quão corretos são os elementos gerados e quantos dos elementos esperados foram gerados.
M3	GED	Quantidade de operações necessário para transformar um diagrama em outro

Tabela 2. Métricas de Avaliação dos Diagramas Gerados

5. Resultados

Nível	Técnica	Execuções	Válidos	Métricas Avaliadas
C2	*0-shot*	280	2	TVS, F1-Score, GED
C2	*one-shot*	280	255	TVS, F1-Score, GED, Especialistas
C3	*0-shot*	280	0	TVS
C3	*one-shot*	280	121	TVS, F1-Score, GED, Especialistas

Tabela 3. Panorama das execuções e coleta de dados

A Tabela 3 apresenta um panorama consolidado das execuções realizadas nos diferentes níveis do modelo C4. Cada cenário foi executado 280 vezes por técnica de *prompting*, totalizando 1.120 amostras. A coluna 'Válidos' indica a proporção de saídas sintaticamente corretas, enquanto as métricas F1 e GED foram aplicadas apenas às execuções válidas. Essa consolidação fornece uma visão quantitativa do escopo experimental e da robustez da coleta de dados

A abordagem *0-shot* mostrou-se inviável, com uma TVS de apenas 0,36%. Em contraste, a abordagem *one-shot* foi essencial para o sucesso, alcançando uma TVS geral de 67,14% (correspondendo aos 378 diagramas sintaticamente válidos obtidos no estudo). Dentro desta técnica, o desempenho foi modulado pelo tipo de diagrama: a TVS para diagramas de C2 foi alta (91,1%), mas caiu significativamente para os diagramas de C3 (43,2%). A Figura 2 detalha esta interação. Uma inspeção dos diagramas válidos na estratégia *0-shot* indica dissociação entre validade sintática e fidelidade semântica. O caso de baixa complexidade (Case 01) obteve F1-Score de 0,21 e GED 21,0. Em contrapartida, o caso de alta complexidade (Case 14) registrou F1-Score de 0,02 e GED 117,0, caracterizado pela geração de 123 falsos positivos e precisão de 1,6%. Tais dados sugerem que, na ausência de exemplos de referência (*one-shot*), o processamento de contextos extensos resulta na inserção de elementos espúrios em vez da representação fidedigna da arquitetura.

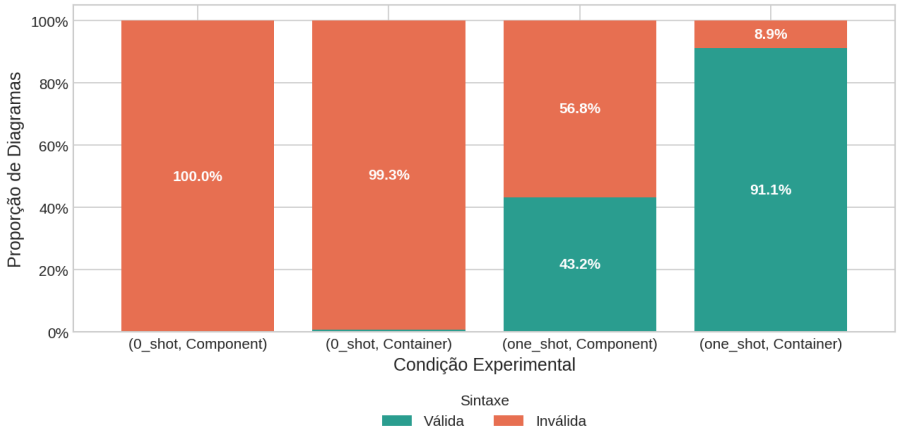

Figura 2. Proporção de Validade Sintática por Condição Experimental

A análise da distribuição proporcional dos tipos de erro, Figura 3, revela perfis de falha distintos entre as técnicas. A abordagem *0-shot* foi dominada por ”Falhas de Gramática Fundamental”, representando 73,4% das falhas em diagramas de Contêiner e 57,5% em Componente. Em contraste, a técnica *one-shot* deslocou o perfil de erro para falhas de natureza semântica. Nos diagramas C2 *one-shot*, a principal causa foi ”Falhas de Duplicidade”(48,0% dos erros), enquanto nos diagramas C3, o perfil foi mais diverso, liderado por ”Falhas de Referência”(35,2%).

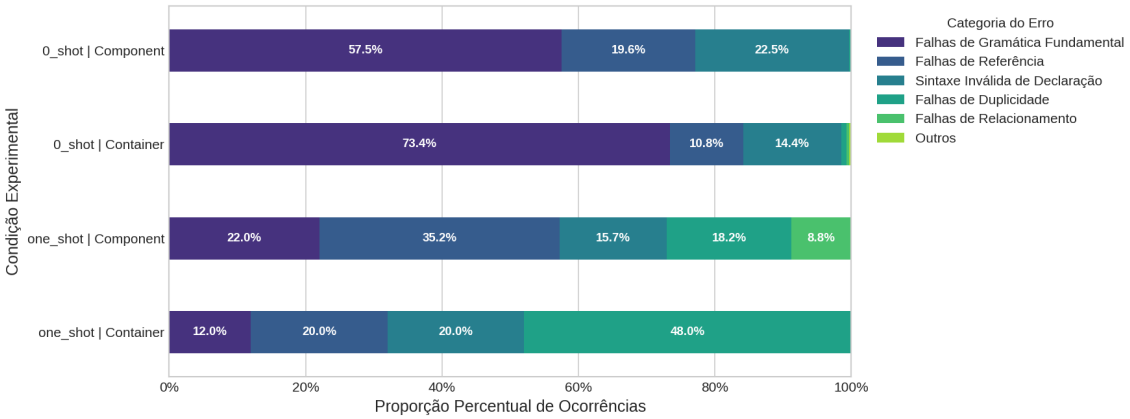

Figura 3. Distribuição de erros por Condição Experimental

A análise quantitativa da qualidade focou nos 376 diagramas válidos da técnica *one-shot*; os 2 diagramas válidos da *0-shot* foram descartados por insuficiência estatística. Os resultados indicam um desempenho superior para diagramas de C2 em relação aos de C3. Na avaliação semântica, a mediana do *F1-Score* foi superior para C2. Similarmente, na avaliação estrutural, os diagramas C2 apresentaram mediana de GED (distância ao gabarito) inferior à dos C3. A análise de consistência (GED Média Intra-grupo) corroborou esta tendência: a geração de C2 foi mais consistente (mediana 10,0) que a de C3 (mediana 15,0), que apresentou maior variabilidade. A mediana geral de consistência foi 12,8.

Os resultados (Figura 4) indicam uma percepção de qualidade superior para os diagramas de C2. C2 apresentou proporções maiores de respostas favoráveis (‘Con-

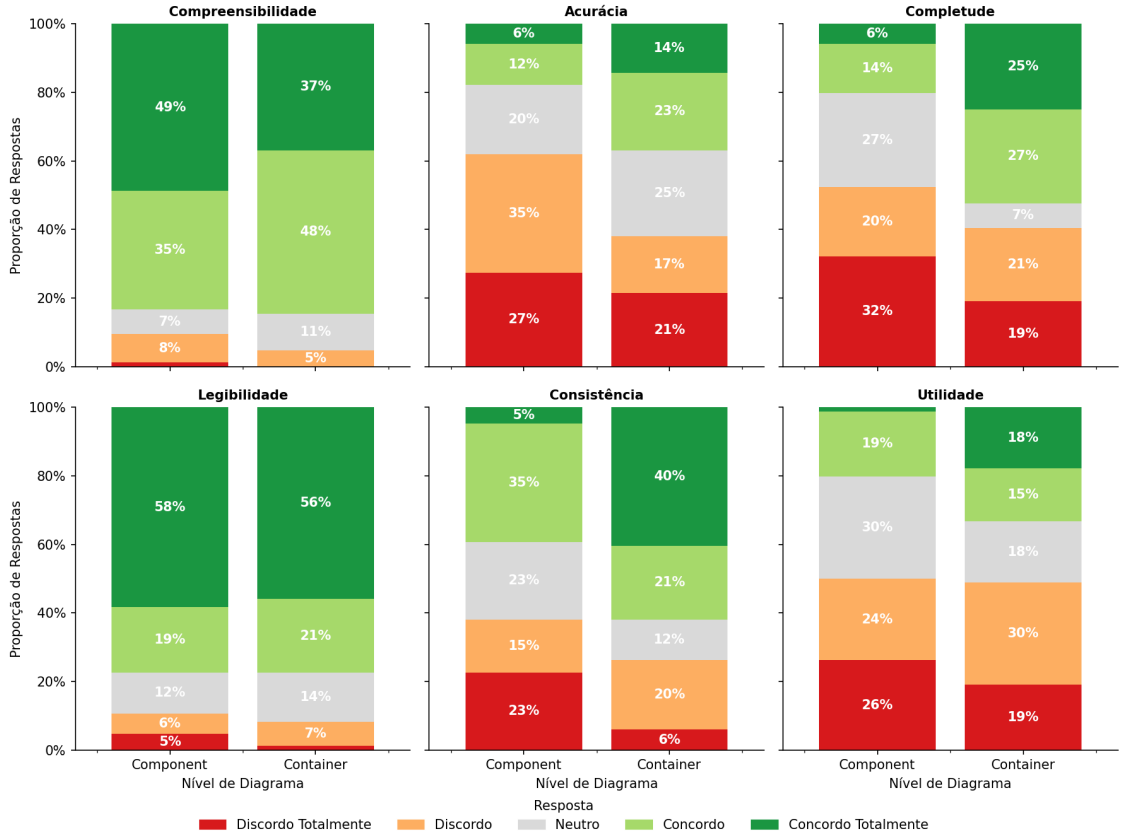

Figura 4. Proporção de Respostas das Avaliações por Propriedades e Nível de Diagrama

cordo'/'Concordo Totalmente') em quase todas as propriedades. As disparidades mais pronunciadas ocorreram em Acurácia, Completude e Utilidade. Em Acurácia, a proporção de respostas negativas ('Discordo'/'Discordo Totalmente') para C3 (62%) foi significativamente maior que para C2 (38%). Em Completude, a proporção negativa também foi maior em C3 (52%) comparado a C2 (40%). A única exceção foi Legibilidade, onde ambos os tipos receberam avaliações positivas similares (77% favoráveis).

6. Discussão

Os resultados deste estudo fornecem evidências empíricas consistentes de que a tradução de linguagem natural para DSLs arquiteturais, como o *Structurizr*, é uma tarefa cuja viabilidade e qualidade dependem fortemente tanto da técnica de *prompting* quanto da complexidade do artefato gerado. Esta seção discute os achados à luz das quatro questões de pesquisa, integrando interpretações quantitativas e qualitativas e relacionando-as à literatura existente sobre geração de artefatos arquiteturais por LLMs.

6.1. RQ1 — Qual a viabilidade de geração de diagramas sintaticamente válidos por LLMs?

Os resultados revelam diferença de TVS entre as abordagens: enquanto o *0-shot* obteve apenas 0,36% de sucesso, o *one-shot* alcançou 67,14%, evidenciando que exemplos contextuais são decisivos para estruturar código formal. A disparidade indica que o modelo não internaliza autonomamente a gramática do *Structurizr* DSL, dependendo fortemente do aprendizado em contexto para reproduzir padrões sintáticos. O modelo demonstrou alta capacidade de adaptação sintática, transferência estrutural, a partir de um único exemplo, validando que a instrução *one-shot* atua como um mecanismo eficiente de transferência local de estrutura [Brown et al. 2020]. O modelo imita a forma apresentada no *prompt* único, sem necessariamente apreender o significado semântico profundo do domínio.

A análise qualitativa reforça essa interpretação. Nos experimentos *0-shot*, 73% dos erros envolveram falhas gramaticais básicas (ausência de chaves, parênteses ou comandos), enquanto no *one-shot* predominaram erros semânticos e referenciais — duplicidade de entidades (48%), referências cruzadas (35%) e inconsistências de escopo (12%). Essa transição demonstra que o modelo, ao receber exemplos, domina a gramática, mas continua limitado em coerência semântica. Em outras palavras, o *one-shot* melhora a forma, não o significado: o modelo compreende a sintaxe da DSL, mas não raciocina sobre a arquitetura que ela expressa. Esses achados convergem com evidências de que LLMs generalistas mantêm déficits de raciocínio simbólico e hierárquico em domínios formais [Dhar et al. 2024a, Esposito et al. 2025]. O aprendizado em contexto corrige a sintaxe superficial, mas não resolve a coerência estrutural, gerando diagramas sintaticamente válidos, porém semanticamente inconsistentes.

Em síntese, a RQ1 demonstra que a geração sintaticamente válida de DSLs por LLMs depende de exemplos explícitos. O *0-shot* revela ausência de competência sintática inata, enquanto o *one-shot* alcança correção formal sem compreensão semântica. Assim, o *prompting* contextual é necessário, mas insuficiente, para garantir validade estrutural plena, refletindo um raciocínio arquitetural simbolicamente frágil: o modelo domina o como escrever, mas não o que representar.

6.2. RQ2 — Qual a fidelidade semântica e estrutural dos diagramas válidos gerados?

Os resultados quantitativos mostram desempenho superior nos diagramas de C2 em comparação aos de C3, com maior F1 e menor GED. Essa diferença indica que o modelo preserva coerência estrutural em representações de nível macro, mas perde precisão à medida que aumenta a granularidade e o número de entidades. Assim, o LLM mantém integridade estrutural apenas enquanto a complexidade combinatória permanece baixa.

A análise qualitativa reforça esse padrão. Avaliadores observaram que os diagramas C2, embora incompletos, preservam a intenção arquitetural geral, enquanto os C3 apresentam três inconsistências recorrentes: (1) omissão de entidades críticas, (2) criação de elementos genéricos sem correspondência semântica e (3) agrupamentos e dependências incorretas. Esses desvios explicam a disparidade entre métricas estruturais e fidelidade semântica, evidenciando que alta similaridade formal não implica compreensão arquitetural. Os especialistas também relataram uma "fidelidade superficial": o mo-

delo replica padrões visuais e nomenclaturas plausíveis, mas perde nuances semânticas, fenômeno consistente com o *semantic drift* descrito por Esposito et al. (2025) , em que LLMs preenchem lacunas de modo coerente, porém incorreto.

A convergência entre análises quantitativas e qualitativas revela que o LLM domina padrões estruturais, mas carece de raciocínio hierárquico e entendimento contextual para preservar relações semânticas coerentes. Assim, fidelidade estrutural e semântica configuram dimensões parcialmente independentes. Em síntese, a RQ2 demonstra que LLMs são formalmente precisos, mas semanticamente incompletos: reproduzem a forma, não o significado. Os resultados reforçam a necessidade de abordagens que integrem raciocínio simbólico ou supervisão humana para garantir consistência semântica em artefatos arquiteturais.

6.3. RQ3 - Qual a consistência dos diagramas válidos gerados?

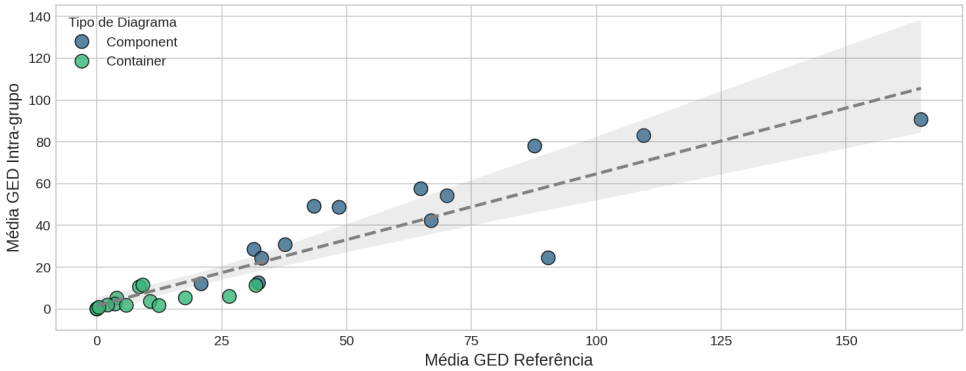

Figura 5. Relação entre Consistência e Acurácia Estrutural

Esta questão examinou a consistência intra-grupo das gerações *one-shot*, avaliando a estabilidade estrutural sob condições idênticas. Foram utilizadas duas métricas: o GED de referência, que mede o desvio em relação ao *ground truth*, e o GED intra-grupo, que capta a variação entre múltiplas execuções. Na Figura 5, o F1-Score correlaciona-se negativamente com o GED intra-grupo (p –0,78): menor consistência implica menor fidelidade semântica. Diagramas de Contêiner mantêm baixa variabilidade, enquanto os de Componente exibem dispersão ampla, refletindo instabilidade crescente com a complexidade. Esses resultados indicam que o modelo alterna entre interpretações estruturais distintas, comprometendo a reprodutibilidade—a mesma instabilidade destacada por Jahić e Sami (2024) e Esposito et al. (2025) .

A Figura 6 reforça o padrão ao mostrar correlação positiva entre GED de referência e GED intra-grupo (p +0,82): quanto maior o desvio estrutural, maior a variabilidade entre execuções. A análise qualitativa confirma essa tendência: em C2, variações são superficiais (ordem ou nomes), enquanto em C3 surgem mudanças de agrupamento, omissões e inversões hierárquicas, alterando a semântica do sistema. Assim, a complexidade intensifica o comportamento estocástico do modelo.

Em conjunto, as Figuras 5 e 6 demonstram que a consistência intra-grupo não é uma propriedade do LLM, mas um efeito da simplicidade da tarefa. A variabilidade cresce com a complexidade e associa-se diretamente à perda de fidelidade, sugerindo coerência

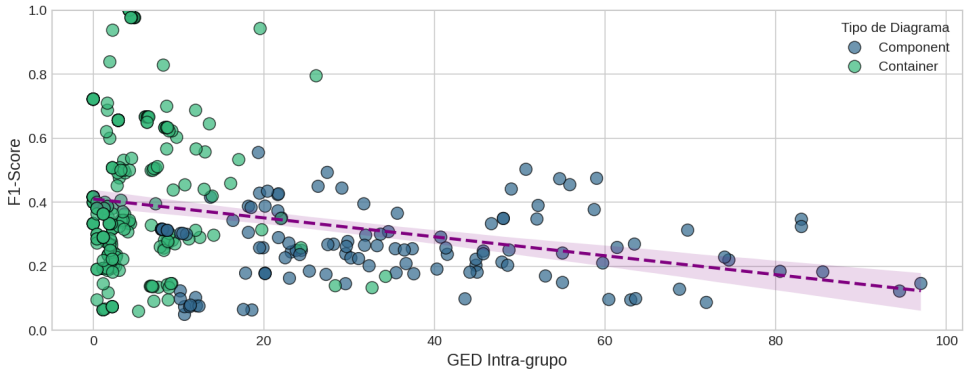

Figura 6. Relação entre Consistência da Geração e Qualidade Semântica

apenas local e ausência de rastreamento hierárquico entre execuções. A RQ3 confirma que LLMs generalistas não apresentam reprodutibilidade determinística em geração arquitetural, mesmo sob *prompting* controlado, limitando seu uso em *Architecture as Code*. Trabalhos futuros devem explorar mecanismos de controle determinístico (como *seed fixing* e *beam search* restrito) ou abordagens híbridas humano–*bot* para estabilizar o raciocínio estrutural.

6.4. RQ4 — Como a complexidade da arquitetura impacta os diagramas válidos gerados?

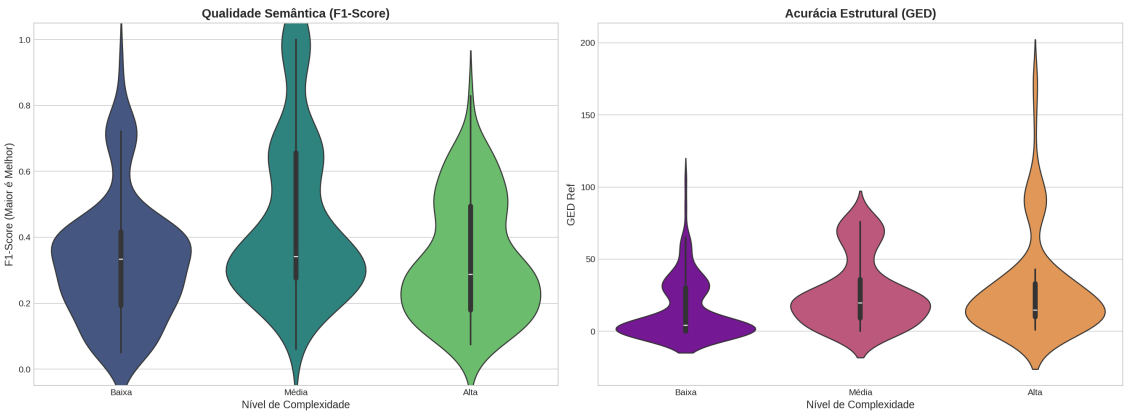

Figura 7. Análise de Qualidade por Nível de Complexidade

Esta questão avaliou o impacto da complexidade na fidelidade semântica (F1) e na acurácia estrutural (GED). A Figura 7 mostra queda progressiva do F1 e aumento da variabilidade intra-grupo, enquanto o GED cresce em mediana e dispersão. Diagramas simples mantêm melhor cobertura semântica e menor erro estrutural; já níveis médio e alto exibem degradação acentuada. Assim, quanto maior a complexidade, menor a consistência e maior o acúmulo de erros.

Essa tendência decorre de limitações dos LLMs em raciocínio hierárquico e consistência topológica. Cinco mecanismos explicam o padrão: (1) explosão combinatória, que amplia o espaço de soluções e o GED; (2) fragilidade hierárquica, com mistura

de níveis arquiteturais; (3) competição por contexto entre descrição e exemplo *one-shot*; (4) generalização genérica diante de topologias inéditas; e (5) estocasticidade da decodificação, que amplia variações em grafos densos. Esses fatores resultam em queda simultânea do F1 e aumento da dispersão do GED, alinhados ao colapso semântico reportado por Dhar et al. (2024) em tarefas estruturais de alta densidade.

Os resultados sugerem relação não linear, com possível ponto de inflexão em que acréscimos de complexidade levam a degradação acelerada. Entre as implicações práticas estão: reduzir carga cognitiva via geração hierárquica, usar *prompts* estruturados para restringir variabilidade e aplicar mecanismos de controle determinístico para diminuir instabilidade. Em síntese, a complexidade é o principal fator de degradação no desempenho de LLMs em geração arquitetural: à medida que aumenta a granularidade, cresce a instabilidade semântica e estrutural. O modelo domina a sintaxe da DSL, mas não o raciocínio estrutural necessário para escala e confiabilidade.

6.5. Ameaças à Validade

No presente trabalho, as ameaças foram avaliadas em quatro dimensões principais, permitindo uma análise sistemática dos riscos e das medidas adotadas para reduzir seus efeitos sobre os resultados.

1. **Validade de Construto:** métricas quantitativas podem não refletir completamente a percepção humana de qualidade; mitigada pelo uso de métricas multifacetadas e avaliação qualitativa.
2. **Validade Interna:** risco de viés do pesquisador; mitigado por validação cega com especialistas externos.
3. **Validade Externa:** escopo restrito (diagramas C4 *Model*, 14 casos) limita a generalização; limitação reconhecida explicitamente.
4. **Validade da Conclusão:** variabilidade dos LLMs pode afetar a confiabilidade; mitigada por execuções múltiplas e análise estatística descritiva.

7. Conclusão

A tradução de linguagem natural para DSLs arquiteturais, como o *Structurizr*, mostrou-se uma tarefa cuja viabilidade e qualidade dependem fortemente tanto da técnica de *prompting* quanto da complexidade estrutural do artefato. Os resultados demonstraram que a abordagem *0-shot* é essencialmente inviável para gerar código sintaticamente válido, enquanto o *one-shot* estabelece um patamar mínimo de viabilidade ao fornecer ao modelo padrões estruturais necessários. Ainda assim, essa melhoria sintática não se traduz em compreensão plena da semântica arquitetural.

A análise quantitativa e qualitativa convergiu para um padrão consistente: diagramas de C2 apresentam maior fidelidade semântica, menor distância estrutural e maior consistência intra-grupo, enquanto diagramas de C3 sofrem degradação acentuada. Esses achados sustentam a hipótese de que a performance dos LLMs é inversamente proporcional à complexidade do artefato, refletindo a dificuldade desses modelos em manter a coesão hierárquica e a rastreabilidade conceitual exigidas por representações arquiteturais mais densas. O estudo, portanto, contribui ao estabelecer um protocolo experimental replicável, baseado em métricas objetivas, e ao evidenciar empiricamente as limitações estruturais dos LLMs generalistas no domínio de *Architecture as Code*.

Apesar de fornecer evidências, o estudo apresenta limitações que circunscrevem sua generalização: análise restrita a um único modelo (*Gemini*), foco em 14 casos de estudo acadêmicos e escopo centrado exclusivamente em diagramas C4. Essas restrições não invalidam os achados, mas indicam a necessidade de ampliar a diversidade de modelos, casos e DSLs em investigações futuras. Em síntese, os resultados mostram que LLMs podem atuar como assistentes contextuais na geração arquitetural, acelerando tarefas de representação estrutural, mas ainda não substituem o julgamento humano na inferência semântica de sistemas complexos. Avanços substanciais em raciocínio hierárquico, técnicas de *prompting* e mecanismos avaliativos serão necessários para que a automação arquitetural atinja maturidade prática.

8. Trabalhos Futuros

Os desdobramentos da pesquisa incluem, primeiro, a ampliação da generalização empírica. Pretende-se replicar e expandir o estudo com outros LLMs (como GPT, Claude e Mistral) e em cenários industriais ou híbridos, investigando se as limitações observadas são específicas do modelo Gemini ou inerentes à tarefa. Também se propõe avaliar DSLs alternativas (como PlantUML e SysML textual) para examinar a escalabilidade e a adaptação do método.

Outro vetor de investigação envolve o aprimoramento da engenharia de *prompt* e de mecanismos de raciocínio explícito. O "gargalo da complexidade" observado nos diagramas C3 sugere a necessidade de decompor a tarefa em etapas menores. Técnicas como *Chain-of-Thought*, *ReAct* e *Tree-of-Thoughts* podem apoiar raciocínio incremental e manutenção de estado hierárquico, enquanto estratégias humano-no-*loop* e *fine-tuning* com exemplos arquiteturais podem reduzir erros semânticos e melhorar a coerência entre camadas.

Por fim, há a agenda de reprodutibilidade e *benchmarking*. Propõe-se consolidar o protocolo metodológico deste estudo em um *benchmark* público para geração de DSLs arquiteturais, incluindo um corpus validado de descrições textuais e diagramas *ground truth*, um protocolo de avaliação padronizado com métricas quantitativas e qualitativas (TVS, F1-Score, GED) e a estratificação por níveis de complexidade, dado seu papel como preditor de desempenho. A criação de um *leaderboard* público possibilitaria comparações objetivas entre modelos e fortaleceria a transparência e a replicabilidade na comunidade de *Architecture as Code*.

9. Pacote de Replicação

O pacote de replicação deste trabalho encontra-se disponível em: https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2025-1-tcci-0393100-pes-murilo-costa

Referências

[Ahmad et al. 2023] Ahmad, A., Waseem, M., Liang, P., Fahmideh, M., Aktar, M. S., and Mikkonen, T. (2023). Towards human-bot collaborative software architecting with chatgpt. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, page 279–285, New York, NY, USA. Association for Computing Machinery.

[Brown 2022] Brown, S. (2022). *The C4 model for visualising software architecture*.

[Brown et al. 2020] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

[c4model.com] c4model.com. Software architecture diagram review checklist. https://c4model.com/diagrams/checklist. Acesso em: 18 maio 2025.

[Carducci 2025] Carducci, M. (2025). *Mastering Software Architecture: A Comprehensive New Model and Approach*. Apress.

[Clements et al. 2010] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J. (2010). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, Upper Saddle River, NJ, 2nd edition.

[Della Pelle 2024] Della Pelle, G. (2024). Archi dataset: a dataset of software engineering projects. https://10.5281/zenodo.14238664.

[Dhar et al. 2024a] Dhar, R., Vaidhyanathan, K., and Varma, V. (2024a). Can LLMs Generate Architectural Design Decisions? - An Exploratory Empirical Study . In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, pages 79–89, Los Alamitos, CA, USA. IEEE Computer Society.

[Dhar et al. 2024b] Dhar, R., Vaidhyanathan, K., and Varma, V. (2024b). Leveraging generative ai for architecture knowledge management. In *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*, pages 163–166.

[Esposito et al. 2025] Esposito, M., Li, X., Moreschini, S., Ahmad, N., Cerny, T., Vaidhyanathan, K., Lenarduzzi, V., and Taibi, D. (2025). Generative ai for software architecture. applications, trends, challenges, and future directions.

[Fan et al. 2023] Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53.

[Fowler 2010] Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley Professional.

[Jahić and Sami 2024] Jahić, J. and Sami, A. (2024). State of practice: Llms in software engineering and software architecture. In *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*, pages 311–318.

[Len Bass 2012] Len Bass, Paul Clments, R. K. (2012). *Software Architecture in Practice*. Prentice Hall.

[Runeson et al. 2012] Runeson, P., Höst, M., Rainer, A. W., and Regnell, B. (2012). *Case Study Research in Software Engineering - Guidelines and Examples*.

[Shehata et al. 2024] Shehata, M., Lepore, B., Cummings, H., and Parra, E. (2024). Creating uml class diagrams with general-purpose llms. In *2024 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 157–158.

[Soliman et al. 2023] Soliman, M., Gericke, K., and Avgeriou, P. (2023). Where and What do Software Architects blog? : An Exploratory Study on Architectural Knowledge in Blogs, and their Relevance to Design Steps . In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, pages 129–140, Los Alamitos, CA, USA. IEEE Computer Society.

[Vaswani et al. 2023] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.

[Von Heissen et al. 2024] Von Heissen, O., Hanke, F., Mpidi Bita, I., Hovemann, A., Dumitrescu, R., et al. (2024). Toward intelligent generation of system architectures. *DS 130: Proceedings of NordDesign 2024, Reykjavik, Iceland, 12th-14th August 2024*, pages 504–513.

[Wei 2024] Wei, B. (2024). Requirements are All You Need: From Requirements to Code with LLMs . In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pages 416–422, Los Alamitos, CA, USA. IEEE Computer Society.

A. Apêndice

A.1. 0-Shot Prompt Template

You are a software architecture expert and a master in the C4 Model and Structurizr DSL. Your task is to generate a {diagram_type} (C4 Model) diagram in Structurizr DSL based on the architecture proposal below. The diagram must follow the official C4 Model conventions (Person, Software System, Container, Component, Deployment). Use Structurizr DSL syntax for the code.

STRICT RULES:

1. The result must contain ONLY Structurizr DSL code.
2. The code must start with workspace opening brackets and ends with closing brackets
3. DO NOT include explanations, comments, or markdown formatting (e.g., no "'dsl).
4. Use snake_case naming convention for all elements.
5. Backend containers must have names ending with _ms.
6. If no architecture layers are explicitly defined, assume a layered architecture with components: controller, service, and repository.
7. Frontend containers must be named webapp or mobileapp. - If multiple frontends exist, their names must end with _webapp or _mobileapp. - All frontend components must have names ending with _page.
8. Databases must have the same name as the backend container and end with _db.

ARCHITECTURE PROPOSAL:

{architecture_proposal_text}

A.2. One-shot Prompt Template

You are a software architecture expert and a master in the C4 Model and Structurizr DSL. Your task is to generate a {diagram_type} (C4 Model) diagram in Structurizr DSL based on the architecture proposal below. The diagram must follow the official C4 Model conventions (Person, Software System, Container, Component, Deployment). Use Structurizr DSL syntax for the code.

STRICT RULES:

1. The result must contain ONLY Structurizr DSL code.
2. The code must start with workspace opening brackets and ends with closing brackets
3. DO NOT include explanations, comments, or markdown formatting (e.g., no "'dsl).
4. Use snake_case naming convention for all elements.
5. Backend containers must have names ending with _ms.
6. If no architecture layers are explicitly defined, assume a layered architecture with components: controller, service, and repository.
7. Frontend containers must be named webapp or mobileapp. - If multiple frontends exist, their names must end with _webapp or _mobileapp. - All frontend components must have names ending with _page.
8. Databases must have the same name as the backend container and end with _db.

EXAMPLE OF A GOOD {diagram_type} DIAGRAM:

{example_code}

ARCHITECTURE PROPOSAL:

{architecture_proposal_text}

A.3. Questionário de Avaliação

1. Compreensibilidade
   (a) A mensagem geral do diagrama é fácil de assimilar.
   (b) Os elementos individuais e seus respectivos papéis são fáceis de identificar e compreender.
2. Acurácia
   (a) O diagrama identifica corretamente os principais elementos (e.g., contêineres, sistemas, pessoas) descritos no texto.
   (b) O diagrama representa corretamente as interações e os relacionamentos entre os elementos, conforme descrito no texto.
3. Completude
   (a) Não há elementos arquiteturais importantes do texto que foram omitidos no diagrama.
   (b) Não há interações importantes entre os elementos que foram omitidas no diagrama.
4. Legibilidade
   (a) A disposição geral dos elementos no diagrama é lógica e bem estruturada.
   (b) As linhas de relacionamento são diretas e evitam cruzamentos excessivos, facilitando o rastreamento das conexões.

5. Consistência
    (a) O diagrama mantém um nível de abstração consistente, sem misturar detalhes de níveis C4 diferentes.
    (b) A terminologia (nomes de elementos, tecnologias, descrições) é usada de forma consistente em todo o diagrama.
6. Utilidade
    (a) Este diagrama seria uma ferramenta eficaz para comunicar a arquitetura a outros membros da equipe ou stakeholders.
    (b) Este diagrama serve como uma base sólida para analisar o design e suportar a tomada de decisões arquiteturais.