

Pipeline Automatizado para Avaliação de Acessibilidade de Interfaces Android Geradas por LLMs

Leonardo Quezado
Universidade Federal do Ceará
Fortaleza, Brazil
leonardoquezado@proton.me

Ribamar de Souza Martins
Universidade Federal do Ceará
Fortaleza, Brazil
ribamarop@gmail.com

Windson Viana
Universidade Federal do Ceará
Fortaleza, Brazil
windson@virtual.ufc.br

ABSTRACT

Research evaluating the quality of artifacts generated by Large Language Models (LLMs), including code, user interfaces, and media, has expanded considerably. However, most evaluation remains manual and time-consuming, making it difficult to keep pace with rapidly released model versions. Recent work also suggests that newer models can regress relative to their predecessors, reinforcing the need for continuous and repeatable evaluation. Within this broader challenge, assessing the accessibility of LLM-generated mobile interfaces is particularly relevant. This paper presents an end-to-end automated pipeline focused on structural and semantic properties for Android Jetpack Compose interfaces, integrating code extraction, deterministic repair rules, Android compilation, custom Lint checks, and dynamic Accessibility Test Framework (ATF) execution with minimal human intervention. We evaluated the pipeline with three LLMs (Claude Sonnet 4.6, ChatGPT 5.5, and Gemini 3.1 Pro) across eight representative mobile interface types. Across 48 evaluations, the pipeline achieved 100% compilation success and reported 409 accessibility findings across static and runtime checks (some of which may correspond to the same underlying defect). These preliminary results show that end-to-end automation can support feasible, repeatable, and scalable accessibility assessment of LLM-generated Compose code while substantially reducing the manual effort required to evaluate rapidly evolving models.

KEYWORDS

LLM, Jetpack Compose, Accessibility, Android, Accessibility Test Framework, Pipeline

1 INTRODUÇÃO

No desenvolvimento de software e na criação de conteúdo multimídia, ferramentas baseadas em Inteligência Artificial (IA), especialmente os Grandes Modelos de Linguagem (LLMs), têm sido cada vez mais usadas para aumentar a produtividade por meio da geração automática de código, interfaces de usuário, documentos hiperímia, metadados e outros artefatos digitais [4, 5, 12, 19]. Paralelamente, cresce o interesse em avaliar sistematicamente a qualidade desses artefatos, uma vez que estudos indicam que versões mais recentes de LLMs nem sempre superam as anteriores em tarefas específicas [2], reforçando a necessidade de reavaliação contínua. Nesse contexto, a avaliação da acessibilidade de interfaces móveis geradas por LLMs emerge como um desafio particularmente relevante.

Apesar da disponibilidade de diretrizes consolidadas de acessibilidade para o desenvolvimento de aplicações móveis e Web, como o WCAG e a ABNT NBR 17060/2022, garantir que aplicações móveis sejam efetivamente acessíveis continua sendo um desafio recorrente [8, 14, 20]. A acessibilidade é frequentemente negligenciada durante o processo de desenvolvimento, resultando em barreiras de acesso para pessoas com deficiência [1, 3, 13, 23].

LLMs tornaram-se ferramentas amplamente usadas por desenvolvedores e designers de interfaces, tanto como assistentes quanto como agentes capazes de gerar código e interfaces de usuário de forma automática [6, 15]. À medida que sua adoção cresce, aumenta também a necessidade de compreender a qualidade dos artefatos criados por esses modelos. Estudos recentes demonstram que interfaces móveis e páginas Web geradas por LLMs apresentam violações de acessibilidade [16, 18, 21], possivelmente em decorrência da presença desses mesmos problemas nas bases de código utilizadas durante o treinamento. Em contrapartida, outras pesquisas indicam que os próprios LLMs podem ser empregados para complementar mecanismos automáticos de verificação e correção de problemas de acessibilidade [7, 22], evidenciando seu potencial tanto como fonte quanto como solução para esse tipo de problema.

Entretanto, avaliar a capacidade dos LLMs de produzir interfaces acessíveis permanece um desafio prático. Os resultados reportados se tornam rapidamente desatualizados, pois refletem o desempenho de versões específicas de LLMs que são substituídas antes mesmo da publicação ou leitura do trabalho. Além disso, a replicação desses estudos envolve um processo demorado, que inclui a elaboração e execução de prompts, compilação do código gerado, instalação de apps em emuladores ou dispositivos físicos, execução de ferramentas de avaliação de acessibilidade e análise manual dos resultados obtidos [16]. Essa complexidade dificulta a realização de reavaliações e comparações sistemáticas entre modelos.

Nesta pesquisa, nosso foco é a **checagem rápida da acessibilidade** de aplicações Android e a **replicabilidade/atualização** de estudos com LLMs. Este artigo apresenta um pipeline automatizado composto por cinco estágios: geração de código por LLMs, extração e autocorreção do código gerado, compilação via Gradle, análise estática por meio de uma versão customizada do Android Lint e análise dinâmica utilizando o Accessibility Test Framework (ATF) [11]. O pipeline avalia cada interface em modo claro e modo escuro, permitindo identificar violações de contraste dependentes do tema visual. Com esse pipeline, buscamos responder às seguintes questões de pesquisa: **RQ1**: Qual é a eficácia do pipeline em compilar código Jetpack Compose sem intervenção manual? **RQ2**: Quais diferenças emergem entre as versões atuais do Claude, ChatGPT e Gemini em relação à acessibilidade das interfaces geradas? **RQ3**:

Em que medida as ocorrências identificadas pelo pipeline correspondem às detectadas por avaliações manuais realizadas com o TalkBack e o Accessibility Scanner?

2 BACKGROUND E TRABALHOS RELACIONADOS

2.1 Acessibilidade em Jetpack Compose

No desenvolvimento de interfaces gráficas, o Jetpack Compose consolidou-se como o *framework* moderno para construção de interfaces declarativas e reativas em Android [10]. Ele disponibiliza uma árvore semântica (*semantic tree*) paralela à hierarquia de *views*, cujos nós são enriquecidos por meio de modificadores como `semantics{}` e `contentDescription`. Durante a execução da aplicação, o ATF (Accessibility Test Framework) [11] percorre a árvore de objetos `AccessibilityNodeInfo` e realiza verificações automáticas de acessibilidade. Entre as verificações suportadas destacam-se: (i) ausência de texto acessível em elementos interativos (*Speakable-TextPresentCheck*, ou *ST*); (ii) alvos de toque com dimensões inferiores a 48×48 dp (*TouchTargetSizeCheck*); e (iii) contraste insuficiente entre texto e plano de fundo (*TextContrastCheck*).

2.2 LLMs e Geração de Código de UI

Estudos recentes [16, 18, 22] investigaram o uso de LLMs para gerar código de interfaces com suporte à acessibilidade. Suh et al. [18] compararam o código gerado por LLMs com código escrito por desenvolvedores humanos em projetos Web reais, mostrando que LLMs podem reduzir problemas comuns como contraste insuficiente e ausência de texto alternativo. Contudo, ainda apresentam dificuldades em cenários mais complexos envolvendo atributos ARIA. Padrão similar foi documentado em React Native [21]. Outros autores investigaram a geração de UIs a partir de imagens, mostrando que modelos como o ChatGPT podem produzir código Android a partir de *mockups* visuais, e que iterações guiadas por *prompting* e *feedback* podem reduzir violações de acessibilidade [22]. De forma semelhante, os autores em [16] analisaram a qualidade de acessibilidade em código Android gerado por diferentes versões do ChatGPT e pelo modelo Sabiá, observando que, LLMs frequentemente não produzam código totalmente acessível e que instruções explícitas de acessibilidade nem sempre melhoram os resultados.

3 PIPELINE DE AVALIAÇÃO

Como discutido anteriormente, esta pesquisa tem como objetivo automatizar o processo de avaliação e *benchmarking* de LLMs, de modo a facilitar a reprodutibilidade e acelerar a evolução de estudos e ferramentas de acessibilidade que usam LLMs. Para isso, propõe-se a redução da intervenção manual por meio de mecanismos de integração via API com LLMs, métodos de autocorreção de código e o uso do ATF como avaliador dinâmico de acessibilidade durante a execução das aplicações. O pipeline, implementado em Python, é composto por seis estágios ilustrados na Figura 1.

3.1 Benchmarking de Prompts e Orquestrador

O pipeline é iniciado pelo avaliador, que configura três dimensões: a base de prompts, a estratégia de prompting e a lista de LLMs a avaliar. Em relação à estratégia, o orquestrador suporta prompts

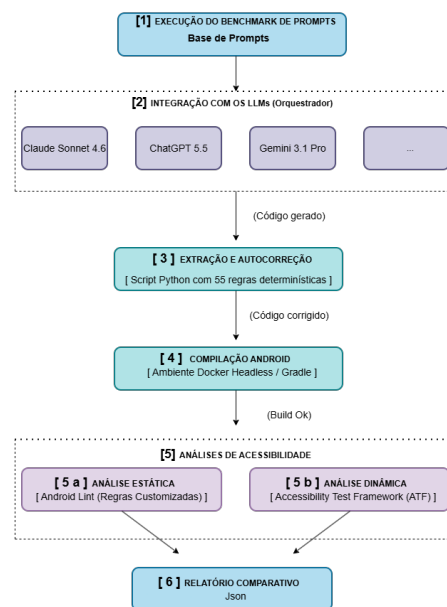


Figura 1: Estágios do pipeline de avaliação automatizada.

zero-shot (sem exemplos, para medir o comportamento padrão dos modelos) e *few-shot* (com exemplos de código correto fornecidos no contexto, para guiar a geração). Cada execução envia ao orquestrador dois blocos: (i) uma descrição de tela específica (e.g., “Generate a complete Android E-Commerce screen”) e (ii) um sufixo fixo idêntico para todas as execuções, com restrições estruturais e de compilação, como o uso do pacote `com.accessibility.test`, Jetpack Compose com Material 3 e Kotlin.

Quando habilitado, o orquestrador realiza *self-repair*: em caso de falha de compilação não resolvida pelo módulo de correção, os erros do compilador são reenviados ao mesmo LLM, que tenta corrigir o próprio código gerado em até duas tentativas. Nesta versão, o orquestrador integra sete modelos via APIs proprietárias (Claude Sonnet 4.6/Anthropic, ChatGPT 5.5/OpenAI, Gemini 3.1 Pro/Google, Grok 2/xAI, DeepSeek-Coder-V2) e as plataformas Llama 3.3 70B via Groq e Llama 3.1 8B local via Ollama. Para este estudo, foram avaliados Claude Sonnet 4.6, ChatGPT 5.5 e Gemini 3.1 Pro com base em: (i) desempenho em benchmarks de geração de código; (ii) disponibilidade de API; (iii) adoção entre programadores [15, 17].

3.2 Extração, Auto-Correção e Compilação

Uma das dificuldades reportadas em estudos anteriores [16, 21] é que o código de interface gerado por LLMs ainda possui falhas eventuais em termos de uso de bibliotecas que impediam sua compilação direta. Este módulo utiliza regras determinísticas mapeadas a partir de erros comuns de compilação identificados em testes exploratórios e na literatura correlata. Por exemplo, os LLMs frequentemente importavam `rememberSaveable` de forma incorreta (`runtime.*` em vez de `runtime.saveable.*`). Ao todo, 55 regras e 128 testes de regressão foram consolidados e congelados antes da fase experimental, garantindo que novos reparos não quebrassem casos anteriores e limitando-se estritamente a falhas de compilação.

Listagem 1: Violação detectada pelo pipeline na tela E-Commerce (Claude 4.6, trecho).

```
// Claude 4.6 -- tela E-Commerce (zero-shot, trecho)
IconButton(onClick = { addToCart(product) }) { Icon(
    imageVector = Icons.Filled.ShoppingCart,
    contentDescription = null // ComposeIconNullContentDescription
)}
```

A Listagem 1 ilustra como conformidade sintática não implica conformidade semântica: o Claude incluiu `contentDescription=null`, satisfazendo a assinatura da função mas suprimindo a descrição para leitores de tela e gerando *ComposeIconNullContentDescription*. Essa distinção entre presença sintática do parâmetro e seu valor semântico é central para compreender por que restrições estruturais no prompt não eliminam violações de acessibilidade. Concluída a etapa de correção sintática, o arquivo fonte é inserido em um template de projeto Android pré-configurado (pacote com.accessibility.test, targetSdk 36 / Android 16) e compilado em um APK de depuração via Gradle (`./gradlew assembleDebug`). O APK é instalado em um dispositivo físico conectado via ADB Wi-Fi e a suíte de testes instrumentados é executada com `./gradlew connectedDebugAndroidTest`.

3.3 Análises de Acessibilidade

A ferramenta realiza tanto a análise estática via Android Lint como análise dinâmica via ATF, que opera sobre a árvore de acessibilidade em tempo de execução. Para contornar a limitação do Android Lint nativo em análise semântica de Compose, foram criadas regras customizadas via UAST (*Unified Abstract Syntax Tree*), distribuídas como módulo Gradle (`:lint-rules`). As regras detectam em tempo de compilação: (i) `Icon()` com `contentDescription` ausente ou nulo em componentes interativos ou como única indicação semântica de uma ação (*ComposeIconNullContentDescription*); (ii) `TextField` e `OutlinedTextField` sem parâmetro `label`, heurística estática que sinaliza potencial ausência de nome acessível em tempo de execução (*ComposeTextFieldMissingLabel*).

Durante a execução dinâmica, o framework percorre a árvore de nós de acessibilidade produzida pela camada semântica do Compose, cobrindo o estado inicial da tela e posições subsequentes obtidas por rolagem. Três verificações são realizadas: (i) *SpeakableTextPresentCheck*, que detecta nós clicáveis ou focalizáveis sem texto nem `contentDescription`; (ii) *TouchTargetSizeCheck*, que verifica se alvos interativos possuem dimensões mínimas de 48 dp × 48 dp conforme recomendado pelo Material Design 3; e (iii) *TextContrastCheck*, que captura um bitmap da tela via `UiAutomation.takeScreenshot()` e estima a razão de contraste entre o primeiro plano e o fundo pelo método de amostragem por frequência de pixels, sinalizando valores abaixo de 4,5:1 (WCAG AA). Ao final, o pipeline extrai as violações detectadas processando os documentos XML gerados pelos testes do Android. Para verificar o comportamento do pipeline, foram criadas três telas sintéticas com violações predefinidas (*ground truth*), executadas em dispositivo físico. A Tabela 1 compara os totais esperados e obtidos.

O Lint correspondeu ao *ground truth* nas três telas. Na *GalleryGridScreen*, o Lint detectou 1 ocorrência estática de `contentDescription=null`, enquanto o ATF identificou 16 instâncias em runtime: 12 itens de grade e 4 botões cujo `Modifier.size(28 dp)` quebrou a propagação do

Tabela 1: Validação com telas sintéticas (Lint e ATF).

Tela	Lint		ATF (ST)	
	Esp.	Real	Esp.	Real
ProfileScreen	5	5 ✓	–	5 [†]
GalleryGridScreen	1	1 ✓	12	16 [†]
SettingsFormScreen	6	6 ✓	6	6 ✓

[†] 4 extras: `Modifier.size` quebra a propagação de `contentDescription` nos `IconButton`s.
[‡] ATF não predefinido no ground truth; violações detectadas em runtime.

`contentDescription`, comportamento invisível à análise estática. *TouchTargetSizeCheck* retornou zero ocorrências pois o `IconButton` do Material 3 impõe 48 dp mínimos independentemente do tamanho visual. Na *SettingsFormScreen*, a correlação Lint–ATF foi perfeita (6–6), confirmando que as duas análises capturam o mesmo problema sob perspectivas complementares.

Tabela 2: Ocorrências detectadas: pipeline vs. avaliação manual (Claude Sonnet 4.6).

Interface	Pipeline	Scanner	TalkBack	Conf.
Music Player	7	0	6	✓
Login	8	0	3 [†]	✓
Social Feed	9	8 [‡]	7+	✓

[†] 2 ícones ignorados pelo leitor (IN) + rótulo duplicado no campo senha (TF).
[‡] Contraste e rótulos duplicados — categorias distintas das do pipeline.

Além disso, três outras interfaces foram inspecionadas manualmente com Accessibility Scanner (Google) e TalkBack. A Tabela 2 compara as ocorrências identificadas pelo pipeline com as identificadas por avaliação manual. O Scanner não identificou violações estruturais (ST, TF, TT e IN) nas interfaces Music Player e Login; na interface Social Feed, detectou violações de rótulos duplicados, categoria não coberta pelo pipeline. O TalkBack confirmou as deficiências detectadas pelo ATF: botões sem rótulo foram anunciados sem indicar a ação correspondente, e ícones com `contentDescription=null` foram completamente ignorados pelo leitor de tela.

4 AVALIAÇÃO EXPERIMENTAL

A Tabela 3 detalha as violações de acessibilidade por tela e modelo de LLM encontradas na avaliação experimental.

4.1 Procedimento

Foram selecionadas oito tipos de interface móvel: Login, Configurações, Dashboard, Chat, Galeria de Fotos, Music Player, E-Commerce e Social Feed [16]. A geração ocorreu via estratégia *zero-shot* (prompts idênticos, variando apenas a descrição da tela) com o *self-repair* via LLM desabilitado, sob temperatura 1,0, Kotlin 1.9.22, AGP 8.2.2 e Compose BOM 2024.02.00. O experimento totalizou 48 execuções (8 telas × 3 modelos × 2 modos visuais), com testes dinâmicos executados em dispositivo físico (Android 16,

Tabela 3: Ocorrências detectadas por tela e modelo (48 execuções: modo claro + escuro). ST = *SpeakableText*; TF = *TextField*; IN = *IconNull*; TT = *TouchTarget*; CT = *TextContrast*; Tot = Total.

Tela	Claude Sonnet 4.6						ChatGPT 5.5						Gemini 3.1 Pro					
	ST	TF	IN	TT	CT	Tot	ST	TF	IN	TT	CT	Tot	ST	TF	IN	TT	CT	Tot
Login	12	4	0	0	0	16	10	4	0	0	0	14	10	4	0	0	0	14
Config.	6	4	2	0	0	12	6	3	0	0	0	9	10	0	0	0	0	10
Dashb.	12	0	4	0	0	16	8	0	0	0	0	8	8	0	0	0	0	8
Chat	8	2	0	0	0	10	8	2	0	0	0	10	8	2	0	0	2	12
Galeria	38	2	0	4	12	56	38	0	0	4	0	42	38	0	0	4	0	42
Music	12	0	0	0	2	14	12	0	0	0	2	14	12	0	0	0	0	12
E-Com.	4	0	8	0	2	14	8	0	0	0	0	8	8	0	0	0	0	8
Social	14	4	0	0	0	18	14	0	0	0	4	18	23	0	0	0	1	24
Total	106	16	14	4	16	156	104	9	0	4	6	123	117	6	0	4	3	130

API 36) via ADB Wi-Fi em mai./2026.¹ A alternância de tema (claro/escuro) utilizou adb shell cmd uimode night yes/no. Cada configuração foi avaliada em execução única. O ATF percorreu o estado inicial e posições subsequentes obtidas por rolagem automática. Cada execução completou em 5,8 min em média (mín. 5,2; máx. 7,2), totalizando **4,6 h** de testes.

4.2 Resultados e Discussão

4.2.1 RQ1: Eficácia do pipeline em compilar código gerado e avaliar a acessibilidade. O pipeline alcançou **100%** de sucesso na compilação (48/48 execuções), sem correções manuais, resultado direto do módulo de auto-correção determinístico. O Android Lint nativo reportou **zero** problemas em todas as execuções, confirmando sua limitação para análise semântica em Compose [9]. As regras customizadas detectaram 31 ocorrências TF e 14 IN (total Lint: 45); o ATF identificou 364 ocorrências dinâmicas (327 ST, 12 TT, 25 CT), totalizando **409 ocorrências** nas 48 execuções. Lint e ATF capturam o mesmo defeito subjacente por camadas distintas e complementares.

4.2.2 RQ2: Diferenças entre Claude, ChatGPT e Gemini. Todos os três modelos produziram interfaces com violações de acessibilidade em todas as execuções. O Claude Sonnet 4.6 registrou o maior número (**156**), destacando-se pela presença exclusiva de *ComposeIconNullContentDescription* (14 ocorrências em Configurações, Dashboard e E-Commerce) e pelo maior número de *TextContrastCheck* (16, concentradas na Galeria de Fotos). O Gemini 3.1 Pro totalizou **130** ocorrências, com pico de ST no Social Feed (23 combinado) refletindo variabilidade entre execuções independentes. O ChatGPT 5.5 apresentou o menor total (**123**), com distribuição mais homogênea e ausência completa de ocorrências IN. *SpeakableTextPresentCheck* (ST) foi o erro dominante, presente em todas as 48 execuções (327 ocorrências), com pico na Galeria de Fotos (ST = 19 por execução nos três modelos). *ComposeTextFieldMissingLabel* (TF) afetou 10 das 24 combinações tela × modelo (31 ocorrências), com maior incidência em Configurações. *TouchTargetSizeCheck* (TT) concentrou-se exclusivamente na Galeria de Fotos (12 ocorrências nos três modelos). *ComposeIconNullContentDescription* (IN) foi exclusiva ao Claude: Configurações (2), Dashboard (4) e E-Commerce (8). *TextContrastCheck* (CT) detectou 25 ocorrências no total, sendo Claude (16), ChatGPT (6) e Gemini (3), evidenciando que cores fixas no código gerado produzem contraste insuficiente.

¹<https://github.com/LeonardoQuezado/Pipeline-Automatizado-para-Avalia-o-de-Acessibilidade-de-Interfaces-Android-Geradas-por-LLMs.git>

4.2.3 RQ3: Correspondência com avaliação manual. A Tabela 2 resume os resultados para três interfaces geradas pelo Claude Sonnet 4.6. O TalkBack confirmou 6 ocorrências ST no Music Player (botões sem rótulo), 3 no Login (ícones invisíveis ao leitor de tela e campo senha com rótulo duplicado) e 7 no Social Feed (nome, @usuário, timestamp, imagem e botões de interação sem texto acessível). O Scanner retornou zero nas duas primeiras interfaces; no Social Feed, detectou 8 ocorrências de contraste e rótulos duplicados, categorias distintas das reportadas pelo pipeline nessa interface. A inspeção manual fornece evidência qualitativa de convergência para categorias estruturais; Scanner e ATF revelam perspectivas complementares sobre acessibilidade.

5 CONSIDERAÇÕES FINAIS

Este artigo apresentou um pipeline automatizado para avaliar interfaces Jetpack Compose geradas por LLMs. Nas 48 execuções, todos os *outputs* compilaram sem correções manuais, e o pipeline reportou **409 violações de acessibilidade**. Os três modelos produziram *outputs* com violações de acessibilidade. As limitações do estudo incluem uma única execução por configuração, validação manual restrita aos *outputs* de um único modelo, contagens brutas e preservação semântica das regras não verificada. Apesar das limitações, o estudo mostrou que o pipeline pode ser uma ferramenta eficiente de *benchmarking* de LLMs em relação à temática de acessibilidade, permitindo tanto a designers e pesquisadores escolherem os modelos mais adequados. Como trabalhos futuros, a ideia é estudar a estabilidade dos modelos com mais repetições de execução e a capacidade de correção de problemas de acessibilidade, assim como estudos de usabilidade da ferramenta.

AGRADECIMENTOS

Esta pesquisa foi parcialmente apoiada pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), sob o número de processo 307831/2025-6.

CUIDADOS ÉTICOS E USO DE IA

Esta pesquisa não envolveu participantes humanos. O modelo Claude Sonnet 4.6 foi utilizado exclusivamente como suporte à redação do texto e à codificação, sendo a responsabilidade pelo conteúdo e decisões intelectuais inteiramente dos autores.

REFERÊNCIAS

- [1] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility issues in Android apps: state of affairs, sentiments, and ways forward. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (ICSE '20). Association for Computing Machinery, New York, NY, USA, 1323–1334. <https://doi.org/10.1145/3377811.3380392>
- [2] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. How Is ChatGPT's Behavior Changing over Time? arXiv:2307.09009 [cs.AI]. <https://arxiv.org/abs/2307.09009>.
- [3] Sen Chen, Chunyang Chen, Lingling Fan, Mingming Fan, Xian Zhan, and Yang Liu. 2022. Accessible or Not? An Empirical Investigation of Android App Accessibility. arXiv:2203.06422 [cs.SE]. <https://arxiv.org/abs/2203.06422>
- [4] Daniel de Sousa Moraes, Polyana Bezerra da Costa, Antonio JG Busson, José Matheus Carvalho Boaro, Carlos de Salles Soares Neto, and Sergio Colcher. 2023. On the Challenges of Using Large Language Models for NCL Code Generation. In *Simpósio Brasileiro de Sistemas Multimídia e Web (WebMedia)*. SBC, 151–156.
- [5] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 81, 13 pages. <https://doi.org/10.1145/3597503.3639219>
- [6] Nayse Fagundes and Leopoldo Teixeira. 2025. Evaluating LLMs for Multimodal GUI Test Generation in Android Applications. In *Simpósio Brasileiro de Testes de Software Sistemático e Automatizado (SAST)*. SBC, 28–36.
- [7] Nadeen Fathallah, Daniel Hernández, and Steffen Staab. 2025. AccessGuru: Leveraging LLMs to Detect and Correct Web Accessibility Violations in HTML Code. In *Proceedings of the 27th International ACM SIGACCESS Conference on Computers and Accessibility*. Association for Computing Machinery, 1–22. <https://doi.org/10.1145/3663547.3746360>
- [8] Reinaldo Ferraz, Ana Duarte, João Bárbara, Adriano C.M. Pereira, and Wagner Meira. 2023. A platform to check website compliance with web accessibility standards. In *Proceedings of the 20th International Web for All Conference* (Austin, TX, USA) (W4A '23). Association for Computing Machinery, New York, NY, USA, 75–78. <https://doi.org/10.1145/3587281.3587289>
- [9] Google. 2024. Jetpack Compose Accessibility Guide. <https://developer.android.com/jetpack/compose/accessibility>. Acesso em: ago. 2025.
- [10] Google. 2024. Jetpack Compose UI App Development Toolkit. <https://developer.android.com/compose>. Acesso em: ago. 2025.
- [11] Google. 2026. Accessibility Test Framework for Android. <https://github.com/google/Accessibility-Test-Framework-for-Android>. Acesso em: mai. 2026.
- [12] Guilherme Lima, Rodrigo Costa, and Marcio Moreno. 2019. Semantics and multimedia: an introduction to symbolic artificial intelligence applied to multimedia. In *Proceedings of the 25th Brazilian Symposium on Multimedia and the Web* (Rio de Janeiro, Brazil) (WebMedia '19). Association for Computing Machinery, New York, NY, USA, 7–9. <https://doi.org/10.1145/3323503.3345031>
- [13] Matheus Lima, Guido Lemos, and Raoni Kulesza. 2021. A Proposal for an Accessible Videoconferencing Interface for People with Hearing Impairments. In *Proceedings of the Brazilian Symposium on Multimedia and the Web*. 29–36.
- [14] Alberto Dumont Alves Oliveira, Paulo Sérgio Henrique Dos Santos, Wilson Estêcio Marcilio Júnior, Wajdi M Aljedaani, Danilo Medeiros Eler, and Marcelo Medeiros Eler. 2023. Analyzing Accessibility Reviews Associated with Visual Disabilities or Eye Conditions. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) (CHI '23). Association for Computing Machinery, New York, NY, USA, Article 37, 14 pages. <https://doi.org/10.1145/3544548.3581315>
- [15] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirel. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv:2302.06590 [cs.SE]. <https://arxiv.org/abs/2302.06590>.
- [16] Diego M. F. Rabelo, Matheus Andrade, Ribamar de Souza Martins, and Windson Viana. 2025. Breaking Barriers in Mobile Accessibility: A Study of LLM-Generated Native Android Interfaces. In *Proceedings of the 12th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft '25)*. <https://doi.org/10.1109/MOBILESoft66462.2025.00010>
- [17] Stack Overflow. 2025. Stack Overflow Developer Survey 2025: AI Agent Out-of-the-Box Tools. <https://survey.stackoverflow.co/2025/ai/3-ai-agent-out-of-the-box-tools>. Acesso em: jun. 2026.
- [18] Hyunjae Suh, Mahan Tafreshipour, Sam Malek, and Iftekhar Ahmed. 2025. Human or LLM? A Comparative Study on Accessible Code Generation Capability. arXiv:2503.15885 [cs.SE]. <https://arxiv.org/abs/2503.15885>.
- [19] Michael Cheng-Tek Tai. 2020. The impact of artificial intelligence on human society and bioethics. *Tzu Chi Medical Journal* 32, 4 (Oct–Dec 2020), 339–343. https://doi.org/10.4103/tcmj.tcmj_71_20
- [20] Christopher Vendome, Diana Solano, Santiago Liñán, and Mario Linares-Vásquez. 2019. Can Everyone use my app? An Empirical Study on Accessibility in Android Apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 41–52. <https://doi.org/10.1109/ICSME.2019.00014>
- [21] Carlos Ventura, Daniel Rabelo, Ticianne Darin, and Windson Viana. 2025. Accessibility Evaluation of LLM-Generated Android Code in React Native. In *Anais do XXIV Simpósio Brasileiro sobre Fatores Humanos em Sistemas Computacionais* (Belo Horizonte/MG). SBC, Porto Alegre, RS, Brasil, 1301–1322. <https://doi.org/10.5753/ihc.2025.10833>
- [22] Lucas Xavier, Daniel Rabelo, Ribamar Martins, and Windson Viana. 2025. Image-to-UI Accessibility: Assessing ChatGPT's Effectiveness in Generating Accessible Android Screens from Figma Templates. In *Proceedings of the 31st Brazilian Symposium on Multimedia and the Web* (Rio de Janeiro/RJ). SBC, Porto Alegre, RS, Brasil, 302–311. <https://doi.org/10.5753/webmedia.2025.15298>
- [23] Shunguo Yan and P. G. Ramachandran. 2019. The Current Status of Accessibility in Mobile Apps. *ACM Trans. Access. Comput.* 12, 1, Article 3 (Feb. 2019), 31 pages. <https://doi.org/10.1145/3300176>