

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Marco Antônio Machado de Arruda

**Análise de Códigos Usando Large Language Models para Identificação de
Code Smell em Aplicativos Criados com App Inventor**

Florianópolis
2025

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

**Análise de Códigos Usando Large Language Models para Identificação de
Code Smell em Aplicativos Criados com App Inventor**

Trabalho de Conclusão de Curso de Graduação em Ciências da Computação, do Departamento de Informática e Estatística, do Centro Tecnológico da Universidade Federal de Santa Catarina, requisito parcial à obtenção do título de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Jean Carlo Rossa Hauck

Florianópolis

2025

Marco Antônio Machado de Arruda

**Análise de Códigos Usando Large Language Models para Identificação de
Code Smell em Aplicativos Criados com App Inventor**

Trabalho de Conclusão de Curso de Graduação em Ciência da Computação, do Departamento de Informática e Estatística, do Centro Tecnológico da Universidade Federal de Santa Catarina, requisito parcial à obtenção do título de Bacharel em Ciência da Computação.

Orientador:

Prof. Dr. Jean Carlo Rossa Hauck
Universidade Federal de Santa Catarina

Banca Examinadora:

Prof. Dr. Cristian Koliver
Universidade Federal de Santa Catarina

Prof. Dr. Elder Rizzon Santos
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Primeiramente quero agradecer aos meus pais, Mari e Sergio, pois ao longo de toda a vida me incentivaram sempre a estudar e enfrentar os problemas e desafios com bons exemplos e amor. Também quero agradecer ao meu irmão, Cesar, por fazer parte de todo o trajeto até aqui, desde as idas à escola quando crianças até este momento em que ambos estamos fazendo graduação (boa sorte). Agradeço também à minha falecida vó (nona), dona Ida, que assim como toda minha família sempre esteve ao meu lado, por todas as tardes de conversas, e por inclusive dar um jeito de ajudar com o computador o qual escrevo estes agradecimentos.

Agradeço também aos amigos que fiz durante todo o curso, especialmente aqueles que conversávamos dia a dia e foram essenciais nesse trajeto, Bruno, Emily, Eric, Gabriel, Gabriela, Julien e Samantha, guardo com carinho todos os momentos que tivemos nos poucos tempos livres que conseguimos encontrar. Também a todos aqueles que conversávamos vez ou outra, ou aqueles que até mesmo apenas nos cumprimentávamos pela UFSC, muito obrigado por todos os momentos convívios.

Também aos meus amigos onde nossos principais encontros foram jogos (principalmente CS), Pepino, Enzo, Vi, Referas, Cuca, Sapo, Ian, Xitão, Vini, João, Fell, Gott, Guto, Rafa, Nz, Blunt, Drena, Jse, Bruno, Iuri, Lyaa, Bia, entre vários outros que vou esquecer, peço desculpas, mas não consigo listar todos, muito obrigado pelos momentos, e também por aceitarem ouvir “eu não posso entrar hoje” pois tinha que fazer o TCC, (uma desculpa que vou perder).

Dois momentos pré-curso que foram essenciais. Primeiro, quando João Eduardo, que estava cursando Ciências da Computação, veio comentando de bits e bytes enquanto estávamos na praça de Urupema, quase como um desafio para que eu entrasse no curso. Segundo, quando recebi a notícia de aprovação, com minha mãe entrando no quarto, e estava jogando com a Lyaa em chamada e agradeço pela felicidade até hoje.

Por fim, mas não menos importante, meu agradecimento a todos os professores e funcionários de educação que fizeram parte dessa jornada, desde o fundamental em Urupema, médio em Lages até minha graduação agora na UFSC.

RESUMO

Code smells são indícios de problemas estruturais em um código que, embora não comprometam imediatamente sua execução, podem impactar negativamente sua manutenção e evolução. A identificação desses padrões é essencial para garantir a qualidade do software, especialmente em ambientes educacionais, onde programadores iniciantes estão em processo de formação. Nesse contexto, a programação por blocos, como a oferecida pela plataforma *App Inventor*, destaca-se como uma abordagem acessível para o ensino de lógica de programação. Este Trabalho de Conclusão de Curso propõe uma abordagem automatizada para análise de códigos criados com *App Inventor*, utilizando técnicas de *RAG* com modelos de *LLM* para identificação de *code smells*. O objetivo é apoiar o ensino de boas práticas de desenvolvimento desde os primeiros contatos dos alunos com a programação, promovendo uma base mais sólida e sustentável para a aprendizagem em computação. Foram testados diferentes modelos locais de *LLM*, para a tentativa de identificação de *smells*, contudo, devido às limitações de *hardware* e modelos simples, não foram obtidos bons resultados para execução local.

Palavras-chave: *Code smells*, *Machine learning*, Linguagem de blocos, *LLM*, *RAG*.

LISTA DE FIGURAS

Figura 1. Área de desenvolvimento da interface gráfica App Inventor.....	15
Figura 2. Área de desenvolvimento da parte funcional App Inventor.....	17
Figura 3. Estrutura do arquivo “.aia“.....	19
Figura 4. Aplicativos da galeria com suas descrições App Inventor.....	21
Figura 5. Representação App Inventor do arquivo “.bky“ do Quadro 8.....	46
Figura 6. Conjunto de blocos sem code smell encontrado no dataset.....	47
Figura 7. Conjunto de blocos com Lazy Class inserido no dataset.....	48
Figura 8. Aplicação que um cachorro late com Lazy Class inserido.....	49
Figura 9. Aplicação de valores nutricionais com Lazy Class inserido.....	49
Figura 10. Visão de blocos do App Inventor para questão 2.....	51
Figura 11. Visão de blocos do App Inventor para questão 10.....	52
Figura 12. Visão de blocos do App Inventor para questão 9.....	53
Figura 13. Visão de blocos do App Inventor para questão 9.....	55
Figura 14. Exemplo de saída gerada pelo modelo Phi 4 mini.....	59
Figura 15. Exemplo de Feature Envy em Aplicativo.....	62
Figura 16. Exemplo de God Class em Aplicativo.....	63
Figura 17. Arquivos selecionados no App Inventor para Análise.....	65
Figura 18. Cinco “Blocos Raíz” em App Inventor.....	80
Figura 19. Quantidade de Tokens no Prompt para RAG.....	81

LISTA DE TABELAS

Tabela 1. Categorias e componentes do Designer do App Inventor.....	15
Tabela 2. Code smells encontrados em trabalho realizado por Galante e Hauck.....	24
Tabela 3. Resultados da busca.....	34
Tabela 4. Trabalhos encontrados.....	35
Tabela 5. Respostas de LLMs (versões web) com One-Shot Prompt.....	50
Tabela 6. Respostas de LLMs locais com One-Shot Prompt.....	53
Tabela 7. Comparação de detecção de smells por abordagem.....	73
Tabela 8. Métricas da Execução - DeepSeek R1.....	74
Tabela 9. Métricas da Execução - Gemma 3.....	75
Tabela 10. Métricas da Execução - Llama 3.1.....	76
Tabela 11. Interpretação de Cohen's Kappa por Landis e Koch - 1977.....	77
Tabela 12. Cohen's Kappa - Todos Modelos.....	77
Tabela 13. Métricas da Segunda Execução - Llama 3.1.....	83
Tabela 15. Cohen's Kappa - Segunda Execução - Llama 3.1.....	83

LISTA DE QUADROS

Quadro 1. Componentes da seção Blocks do App Inventor.....	18
Quadro 2. Síntese dos estudos selecionados com base nas questões de análise...	36
Quadro 3. Code smells identificados por estudo.....	38
Quadro 4. Métodos de aprendizado de máquina para detecção por estudo.....	39
Quadro 5. Tratamento de code smells por estudo.....	40
Quadro 6. Impacto da identificação e análise dos code smells por estudo.....	40
Quadro 7. Métricas adotadas nos estudos para análise de desempenho da identificação de code smells.....	41
Quadro 8. Trecho do script de extração.....	45
Quadro 9. Estrutura interna de um arquivo “.bky“	46
Quadro 10. Prompt usado para One-Shot Prompting.....	50
Quadro 11. Rotulação com metadados de um arquivo.....	66
Quadro 12. Código de Criação do Banco Vetorial.....	66
Quadro 13. Prompt final - parte 1.....	68
Quadro 14. Definições de smells para o Prompt.....	68
Quadro 15. Prompt final - parte 2.....	69
Quadro 16. Inicialização do mecanismo RAG.....	70
Quadro 17. Função de truncamento do código.....	81
Quadro 18. Template para Sumarização das Respostas por Blocos.....	82

LISTA DE ABREVIações

BERT	<i>Bidirectional Encoder Representations from Transformers</i>
FAISS	<i>Facebook AI Similarity Search</i>
GQS	Grupo de Qualidade de Software
IA	Inteligência Artificial
LGPD	Lei Geral de Proteção de Dados
LLM	<i>Large Language Model</i>
MIT	<i>Massachusetts Institute of Technology</i>
ML	<i>Machine Learning</i>
NLP	<i>Natural Language Processing</i>
RAG	<i>Retrieval-Augmented Generation</i>
RAM	<i>Random-Access Memory</i>
RL	<i>Reinforcement Learning</i>
RLHF	<i>Reinforcement Learning from Human Feedback</i>
UFSC	Universidade Federal de Santa Catarina
VPL	<i>Visual Programming Languages</i>
VRAM	<i>Video Random Access Memory</i>

Sumário

1. INTRODUÇÃO	10
1.1 CONTEXTUALIZAÇÃO	10
1.2 OBJETIVOS	11
1.2.1 Objetivo Geral	11
1.2.2 Objetivos Específicos	11
1.3 Metodologia	12
1.3.1 Etapa 1 - Síntese da fundamentação teórica	12
1.3.2 Etapa 2 - Levantamento do estado da arte	12
1.3.3 Etapa 3 - Análise e comparação das técnicas	13
2. FUNDAMENTAÇÃO TEÓRICA	14
2.1 APP INVENTOR	14
2.1.1 Editor de design	15
2.1.2 Editor de blocos	17
2.1.3 Compartilhamento/Exportacao de projetos	19
2.2 CODE SMELLS	22
2.2.1 Code Smells em VPL	23
2.2.2 Principais Code Smells	24
2.3 MACHINE LEARNING (ML)	28
2.3.1 Tipos de Aprendizado de Máquina	28
2.3.2 Large Language Model - LLM	30
2.3.3 Detecção de code smells com ML	31
3. TRABALHOS RELACIONADOS	33
3.1 PROTOCOLO DE PESQUISA	33
3.2 EXTRAÇÃO E ANÁLISE DOS DADOS	36
4. IDENTIFICAÇÃO DE CODE SMELLS COM LLMs	44
4.1 PREPARAÇÃO DO DATASET	44
4.2 TRATAMENTO DO DATASET	45
4.3 EXPERIMENTOS INICIAIS	47
4.3.1 Estruturação dos casos de teste	47
4.3.2 Aplicação dos exemplos em One-Shot Prompting	50
5. IMPLEMENTAÇÃO COM USO DE RAG	58
5.1 DECISÕES TECNOLÓGICAS	58
5.2 CODE SMELLS ESCOLHIDOS E DEFINIÇÕES	60
5.2.1 Feature Envy	61
5.2.3 Lazy Class	64
5.3 PREPARAÇÃO DO DATASET PARA RAG	64
5.4 APERFEIÇOAMENTO DO PROMPT	67
5.5 ANÁLISE DOS RESULTADOS	72
5.5.1 Precision, Recall, F1-Score, Accuracy	72
5.5.2 Cohen's Kappa	76

5.6 AMEAÇAS À VALIDADE DOS RESULTADOS	78
5.7 SEGUNDO EXPERIMENTO COM OTIMIZAÇÕES DE TOKENS	79
5.7.1 Alterações realizadas no segundo experimento	80
5.7.2 Análise dos Resultados do Segundo Experimento	82
5.8 DISCUSSÃO	84
6. CONCLUSÃO	86
REFERÊNCIAS	88
APÊNDICES	92

1. INTRODUÇÃO

Neste capítulo é feita uma contextualização sobre o trabalho, apresentando os temas e motivos principais para sua realização. Após isso são exibidos os objetivos do trabalho e por fim explicada a metodologia utilizada.

1.1 CONTEXTUALIZAÇÃO

Diversos dos estudos têm envolvido o uso de linguagens de programação visuais (*Visual Programming Languages* - VPLs), linguagens muito usadas para o primeiro contato com programação de diversos estudantes. Tais linguagens têm sido utilizadas em pesquisas com alunos de 12 a 14 anos, como demonstrado por Hermans e Aivaloglou (2016), evidenciando o acesso cada vez mais precoce à tecnologia por parte de crianças e adolescentes.

Seja por diversão ou para estudos, o contato das crianças desde a infância permite introduzir conceitos de computação como a criação de aplicativos móveis, que pode ser integrada à educação básica por meio de ferramentas como *App Inventor*, que é capaz de deixar a programação mais visual e intuitiva (Patton et al., 2019).

A ferramenta *App Inventor* não atua somente como uma forma de introduzir as crianças mais cedo à conceitos de programação, mas também como uma forma de democratização do desenvolvimento de aplicativos móveis, (Turbak, Mustafaraj, et al., 2017), pois é usado por mais de 23 milhões de usuários e mais de 115 milhões de aplicativos já foram produzidos¹.

No entanto, é comum que muitos alunos de programação, especialmente os iniciantes, produzam fragilidades estruturais e de estilo. Nesse sentido, este estudo busca trabalhar com *code smells*, conceitos originalmente popularizados por Martin Fowler em 1999, são fonte de debate e pesquisa até os dias de hoje. São padrões em códigos-fonte que, embora não representem erros propriamente ditos, funcionam como indícios de fragilidade estrutural, quando não identificados e tratados desde cedo podem acabar atuando como mensageiros de problemas.

A identificação de *code smells* não tem o objetivo apenas de prevenir desastres, mas sim de formar desenvolvedores que favorecem boas práticas de programação, melhorando a qualidade do *software* produzido (FOWLER; BECK,

¹ <https://appinventor.mit.edu>

2018). Nesse âmbito, o presente trabalho busca reconhecer *code smells* em ambientes de programação visual frequentemente utilizada por crianças tendo seus primeiros contatos com a computação.

Assim, este trabalho busca experimentar sistematicamente o desempenho de diferentes abordagens baseadas em aprendizado de máquina, com ênfase nas técnicas de *prompt-based learning* (BROWN et al., 2020) e *Retrieval-Augmented Generation* (RAG) com LLMs (LEWIS et al., 2020), aplicadas à identificação de *code smells* em projetos desenvolvidos na plataforma *App Inventor*. Busca-se avaliar a eficácia dessas abordagens em um contexto de programação visual, visando compreender seu potencial na promoção de boas práticas desde as fases iniciais da aprendizagem em computação.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo geral do presente projeto é avaliar e comparar, de forma sistemática, o desempenho da técnica de RAG utilizada em diferentes modelos de LLM na identificação de *code smells* em aplicativos desenvolvidos com *App Inventor*.

1.2.2 Objetivos Específicos

- Analisar o contexto da programação por blocos e os desafios inerentes a essa abordagem na literatura.
- Levantar o estado da arte em relação às técnicas de *machine learning* utilizadas em identificação de *code smells*.
- Elencar modelos de LLMs utilizando RAG's com bom desempenho para identificar *code smells* em aplicações *App Inventor*.
- Desenvolver experimentos para avaliar a viabilidade do uso de modelos locais de LLM para identificação de *code smells* em aplicações *App Inventor*.

1.3 Metodologia

Do ponto de vista da natureza da pesquisa, este trabalho se classifica como uma pesquisa aplicada (ou tecnológica), que tem por objetivo gerar um modelo inédito, com finalidades imediatas, com base em conhecimentos prévios. Para desenvolver os resultados esperados no presente projeto, é adotada uma combinação de metodologias de pesquisa de acordo com o respectivo objetivo/meta a ser alcançado.

Neste trabalho foram utilizadas ferramentas baseadas em modelos de linguagem natural (LLMs), incluindo o *ChatGPT* (OpenAI, 2025), *Gemini* (Google, 2025), *DeepSeek* (DeepSeek AI, 2025) e *NotebookLM* (Google, 2025b). Empregadas a fim de comparar qualidade de respostas, avaliação de caminhos, aprimoramento de textos, comparação de trabalhos correlatos. Ressalta-se que nenhuma delas foi utilizada para a geração integral de conteúdos, mas sim como apoio no processo de refinamento e validação das produções desenvolvidas pelo autor.

1.3.1 Etapa 1 - Síntese da fundamentação teórica

É realizada uma revisão rápida da literatura abordando os conceitos fundamentais relacionados a *code smells* (TRICCO et al., 2015), bem como um estudo sobre aplicativos desenvolvidos com o *App Inventor* e as possíveis relações entre essas duas áreas.

- Atividade 1.1: Sintetizar conceitos de desenvolvimento de aplicativos com *App Inventor*
- Atividade 1.2: Estudo da literatura sobre *Code Smells*
- Atividade 1.3 Sintetizar conceitos sobre LLM e RAG

1.3.2 Etapa 2 - Levantamento do estado da arte

Analisar o estado da arte sobre *code smells* em linguagens de programação visual, bem como em relação a abordagens existentes voltadas ao uso de *Machine Learning* na identificação de *code smells* em *apps*. Classificar algoritmos já existentes e definir métricas e algoritmos a serem utilizados. Para esta etapa é utilizada a técnica de mapeamento sistemático de literatura (Petersen et al., 2008).

- Atividade 2.1: Levantamento do Estado da Arte sobre *Code Smells* em

linguagens de programação visuais (VPLs).

- Atividade 2.2: Levantar o Estado da Arte de *Machine Learning* aplicado à identificação de *Code Smells*
- Atividade 2.3: Identificar e classificar abordagens existentes para detecção de *code smells*.
- Atividade 2.4: Definir métricas para extração de *code smells* dos aplicativos.
- Atividade 2.5: Definir os modelos de *LLM* a serem utilizados

1.3.3 Etapa 3 - Análise e comparação das técnicas

Seguindo a abordagem de estudo de caso (Wohlin *et al.*, 2012; Yin, 2017), é definido um estudo de caso para sistematicamente avaliar técnicas de identificação de *bad smells*² e compará-las. Isto envolve tanto a definição do estudo, quanto a definição de medidas adequadas de desempenho como *precision*, *recall* e *F1-Score*. Como conjunto de dados são utilizados *apps* disponíveis na galeria do *App Inventor* e dos MIT *appathons/App of the month*. Durante a execução da análise são aplicadas técnicas de identificação de *code smells*, medindo o desempenho de cada técnica para que ao final sejam comparados os resultados obtidos.

- Atividade 3.1: Preparação do *dataset* de códigos-fonte de projetos *App Inventor*
- Atividade 3.2: Preparar dados para RAG e realizar testes com modelos
- Atividade 3.3: Avaliação de desempenho das técnicas.

² Também chamado de *code smell* na literatura

2. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados conceitos fundamentais para o desenvolvimento deste trabalho. Primeiramente foram comentadas definições sobre os aplicativos criados com *App Inventor*, seus objetivos, técnicas e modos de criação de interface e também lógico. Após isso, foram demonstrados conceitos acerca de *code smells*, sua classificação e formas de avaliação. Por fim, é apresentada uma passagem sobre *Large Language Models (LLM)* introduzindo ideias, usos e aplicações neste projeto.

2.1 APP INVENTOR

*App Inventor*³ é uma plataforma online de desenvolvimento de aplicativos para dispositivos móveis. Seu principal objetivo é a democratização do conhecimento entre os jovens, permitindo, de uma maneira simples o contato com a criação de aplicativos por pessoas que mesmo com pouca ou nenhuma experiência são capazes de utilizá-lo. A plataforma web implementa técnicas de “O que você vê é o que você tem” por meio do uso da técnica de *drag and drop* para a codificação dos aplicativos, onde os usuários programam por meio de arrastar e soltar para criar a lógica necessária (Patton et al., 2019).

O *App Inventor* permite tanto programar a parte do design quanto a parte funcional em diferentes áreas da ferramenta. Dois editores são disponibilizados para que a programação aconteça da forma mais natural possível independentemente dos níveis de conhecimento dos usuários.

O *App Inventor* pertence à categoria das Linguagens de Programação Visual (*Visual Programming Languages / VPLs*), que são sistemas de programação baseados em representações gráficas ao invés da clássica sintaxe textual. O principal objetivo das VPLs é tornar o processo de desenvolvimento mais acessível, especialmente para iniciantes, por meio de interfaces intuitivas que abstraem conceitos complexos da programação tradicional. Por meio de blocos com diferentes tamanhos e cores as compatibilidades são expressadas, permitindo que os usuários possam focar na estrutura lógica (TSAI, 2019). Além de facilitar o ensino de lógica de programação, as VPLs promovem um aprendizado mais lúdico e

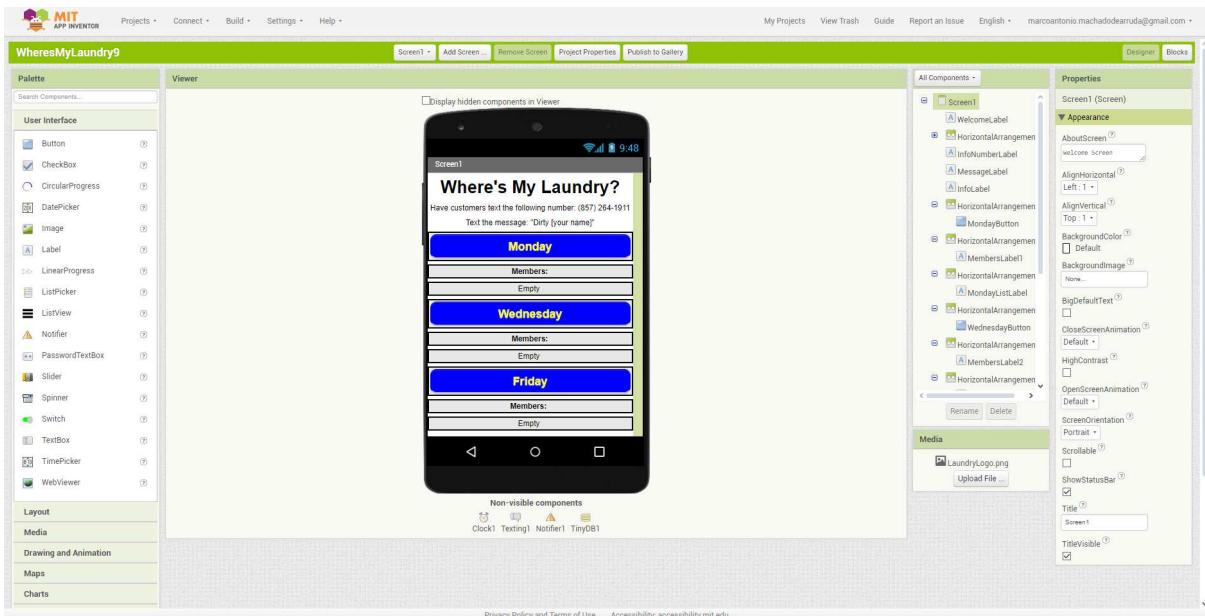
³ <https://appinventor.mit.edu>

interativo, o que está diretamente alinhado com a proposta educacional do App Inventor (Patton et al., 2019).

2.1.1 Editor de design

O desenvolvimento da interface de usuário acontece no editor de design. Sua interface é dividida em algumas categorias, sendo elas: paleta, visualizador, componentes, mídia e propriedades. O utilizador pode posicionar elementos na interface e receber o *feedback* visual instantaneamente, nomear os diferentes elementos bem como alterar as suas propriedades.

Figura 1. Área de desenvolvimento da interface gráfica *App Inventor*



Fonte: Autor (2025)

Dentro da divisão “paleta”, a plataforma disponibiliza vários componentes classificados em diversas categorias (Tabela 1). Isto inclui elementos de *UI design*, mas também acesso a mídia do celular e selecionar quais partes do dispositivo móvel vão ser acessadas, como câmera, microfone, gps, entre outros.

Tabela 1. Categorias e componentes do *Designer* do *App Inventor*

Categoria	Descrição	Componentes
Interface de Usuário	Componentes de interface	Botão, Legenda, Imagem, Deslizador, etc.

Organização	Permitem escolher as posições de outros componentes	Organização Horizontal, Vertical, em Tabela, etc.
Mídia	Componentes de acesso à mídia do dispositivo	Câmera, Som, Gravador, Reconhecedor de Voz, etc.
Desenho e Animação	Componentes de desenho ou com movimentos dinâmicos	Bola, Pintura e Sprite de Imagem.
Maps	Componentes voltados para uso com mapas	Mapa, Marcador Navegação, Polígonos, etc.
Charts	Uso de tabelas	Gráfico, Dados 2D para o Gráfico, Linha Tendência
Data Science	Permitem formas simples de análise de dados	Detecção de Anômalos e Regressão.
Sensores	Acesso aos sensores do dispositivo	Acelerômetro, Barômetro, Temporizador
Social	Acesso a alguns meios de comunicação do usuário	Escolher Contato, Ligação, Mensagem SMS, etc.
Armazenamento	Formas de armazenamento de dados	CloudDB, TinyDB, Arquivo, Planilha, etc.
Conectividade	Possibilitam ligações web ou bluetooth	Cliente/Servidor Bluetooth, Web, Serial, etc.
LEGO® MINDSTORMS®	Possibilitam integração com robôs LEGO	NxtAcionar, NXTSensorDeCor, etc.
Experimental	Componentes em fase de experimentação	ChatBot, ImageBot, FirebaseDB.
Extension	Extensões que podem ser adicionadas	-

Fonte: Autor (2025)

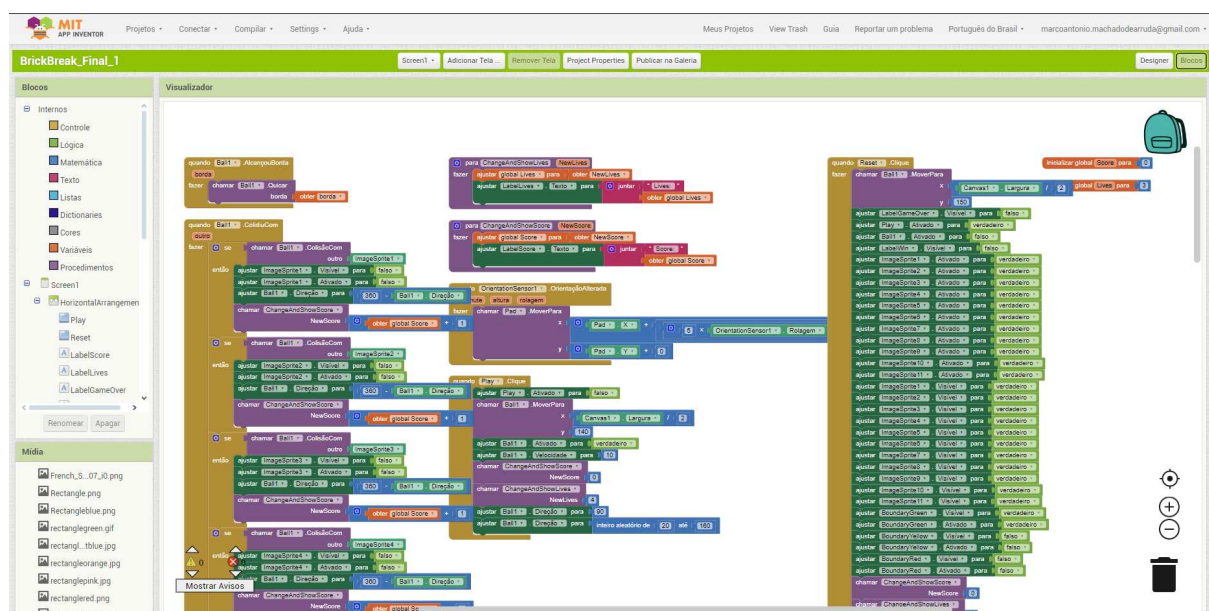
Observando o “Visualizador” tem o retorno visual das mudanças realizadas no editor de design sobre a tela selecionada, qualquer alteração realizada com os componentes visuais reflete em mudanças na imagem encontrada no visualizador. Já em “Mídia” pode-se fazer upload de mídias em geral, como imagens e vídeos que vão ser utilizadas no aplicativo.

Em “Componentes”, é possível encontrar as partes do aplicativo em desenvolvimento, seus componentes e também organizá-los de uma forma mais manual do que arrastar até o visualizador, como por exemplo: imagens compondo outras imagens. A parte de “Propriedades” contém detalhes de cada componente, sendo possível alterar nomes, aparências e comportamento associado, mas de forma individual, sem programar interação com outros componentes.

2.1.2 Editor de blocos

A parte lógica do sistema é realizada na parte da edição de blocos, onde por meio de uma forma visual e interativa os usuários conseguem usar blocos para replicar o funcionamento desejado de uma forma mais intuitiva à pessoas sem conhecimentos de programação avançados. É possível com a programação de blocos replicar comportamentos complexos, como a interação dos componentes e ações resultantes delas em outros componentes. A área do “visualizador”, no editor de blocos (Figura 2), passa a apresentar a programação lógica feita pelo usuário e não mais a interface do programa.

Figura 2. Área de desenvolvimento da parte funcional *App Inventor*



Fonte: Autor (2025)

A área “blocos” permite representar tanto conceitos clássicos de computação, como laços, funções, condicionais, etc. quanto condições, eventos e ações para um componente do aplicativo (Alves, 2023). É possível a atribuição de variáveis, alteração de componentes da interface e de seus comportamentos baseado em eventos de outros componentes, além de conseguir manter estruturas de dados como dicionários e listas e seu funcionamento totalmente a partir da ligação de blocos (Quadro 1).

Quadro 1. Componentes da seção *Blocks* do *App Inventor*

Categoria	Exemplo de Blocos	Equivalente Tradicional	Observações
Control	If-Then; If-Then-Else; For each; Break; Close screen; etc	Estruturas de controle (if, for,)	Lógica condicional e repetição
Logic	True; False; Not; And; Or; =	Operadores lógicos	Operações booleanas
Math	+; -; /; x; =; Min; Max; Mean; Square root; Absolute; Random; etc	Operações matemáticas	Funções matemáticas básicas
Text	Lenght; Is empty; Trim; Upcase; Contains; Split at; etc	Manipulação de strings	Concatenação, extração, verificações em strings
Lists	Create empty list; Make a list; Add items to list; is in list?; Length of list; Pick a random item; etc	Estrutura de lista/lista ordenada	Operações sobre listas incluindo map, sort, filter, entre outros
Dictionaries	Create empty dictionary; Get value for key; Remove entry for key; Dictionary to list of pairs	Estrutura de dicionário (chave-valor)	Manipulações mais limitadas do que aquelas sobre listas
Colors	Make color; Split color; (cores puras)	Componentes de cores RGB(A)	Make color exige uma lista com 3 ou 4 valores RGB(A)
Variables	Initialize global; Get; Set-to; Initialize local	Declaração e uso de variáveis	Variáveis locais e globais, utilizado
Procedures	To procedure do; To procedure Result	Função e procedimento	Abstração e reutilização de código, com ou sem retorno
“Componentes”	Botões; Organizadores; Espaçamentos; Imagens; Base de dados; etc	Elementos adicionados na parte de <i>design</i> , visíveis ou não.	Event Handlers para objetos, como cliques em botões, imagens. Também alteração de propriedades, como espaçamento, tamanho de imagens, etc

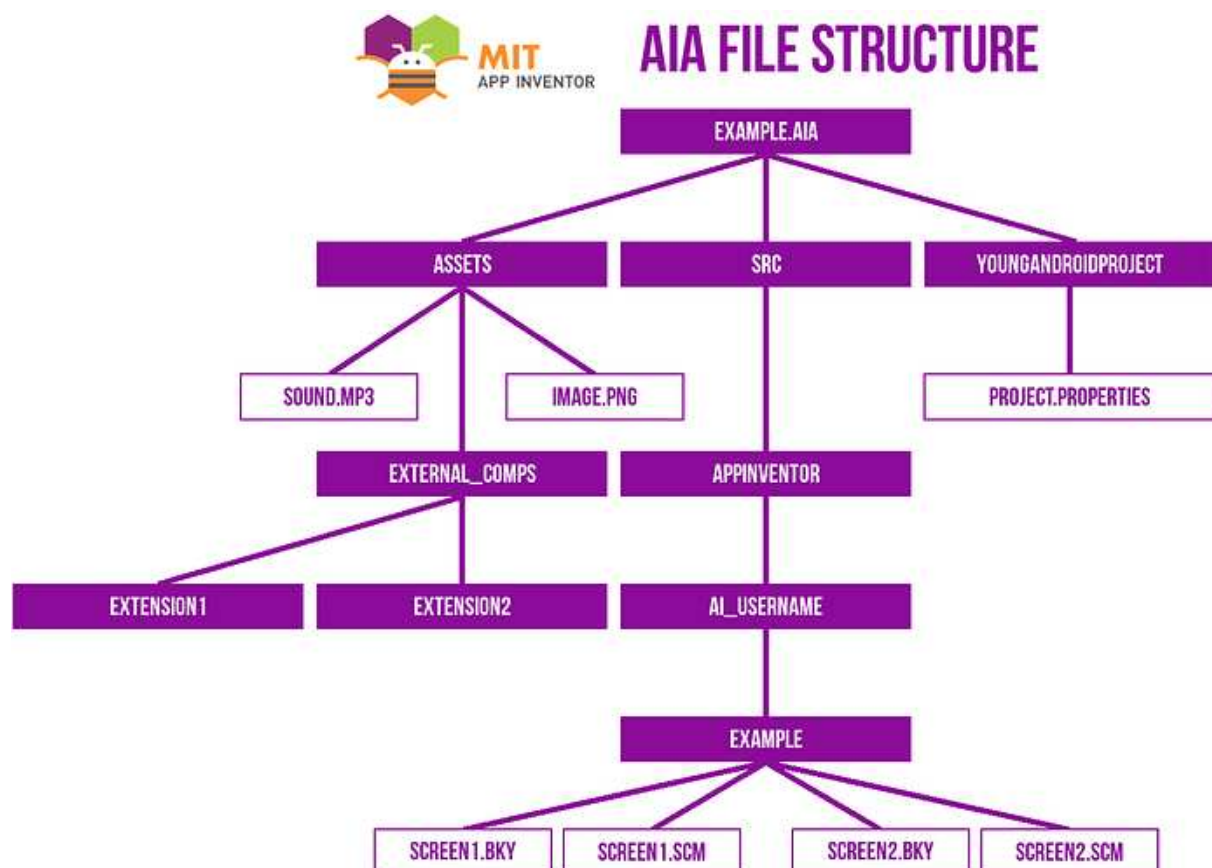
Fonte: Autor (2025)

2.1.3 Compartilhamento/Exportacao de projetos

Após criados para compartilhar com outras pessoas, os aplicativos podem ser exportados para o computador no formato “.aia”, um *zip* compactado (<https://docs.fileformat.com/pt/programming/aia/>).

Os arquivos “.aia” contém os arquivos de mídia, propriedades do projeto e para cada tela dois arquivos principais: um arquivo “.bky” e um no formato “.scm”. O arquivo “.bky” contém a estrutura XML com os blocos de programação usados, enquanto o “.scm” contém um JSON com os componentes visuais do aplicativo (Turbak, Mustafaraj, et al., 2017).

Figura 3. Estrutura do arquivo “.aia”

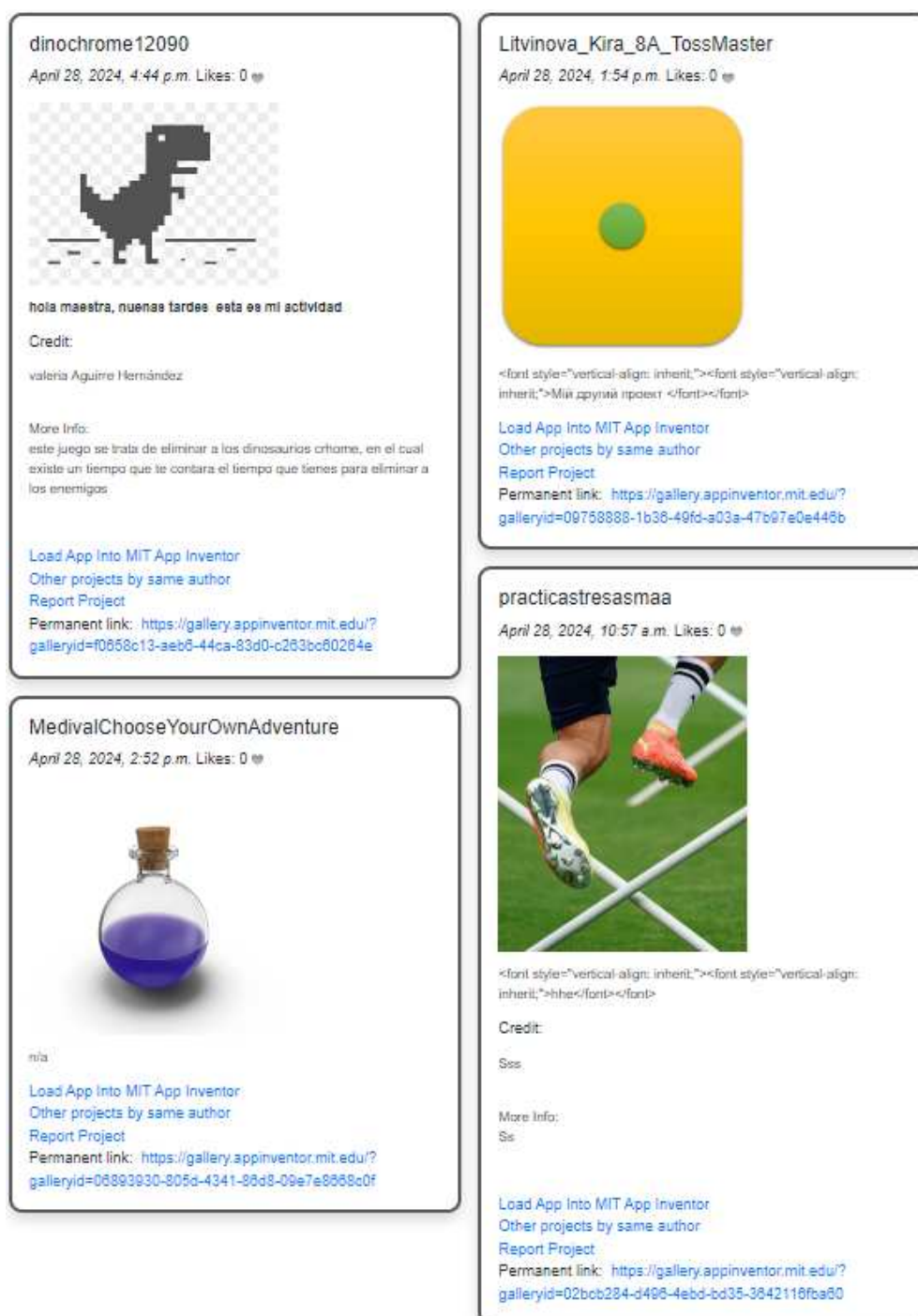


Fonte: (MIT APP INVENTOR COMMUNITY, 2019)

Os aplicativos criados também podem ser publicados da *Gallery* do *App Inventor*, onde ficam disponíveis para download por qualquer usuário. Para cada publicação há um conjunto de informações adicionais que podem ser dadas, como uma descrição, créditos ao(s) seu(s) autor(es), informações extras e uma imagem para o app.

Embora o *App Inventor* ofereça campos para descrição, créditos e informações adicionais, é comum notar aplicativos com descrições incompletas ou ausentes (Figura 3). Dado como foco o aprendizado de programação por crianças/jovens a documentação acaba sendo deixada de lado na maioria dos casos, não havendo filtros na aplicação para a postagem de descrições.

Figura 4. Aplicativos da galeria com suas descrições *App Inventor*



Fonte: Autor (2025)

São organizadas também competições como *App of the Month* e *Appathons* pela equipe do MIT *App Inventor*, que são maratonas para a construção de aplicativos na plataforma, para quais os participantes podem submeter seus aplicativos para a competição. Buscam explorar temas da comunidade, como mudança climática, educação, saúde mental, entre outros. Estas competições são

realizadas como um incentivo para não só o aprendizado de computação, mas também para o desenvolvimento de soluções para problemas do cotidiano.

2.2 CODE SMELLS

O termo *code smell* foi introduzido por Kent Beck e popularizado por Martin Fowler em sua obra *Refactoring: Improving the Design of Existing Code* (1999). Refere-se a características de código que podem indicar falhas mais profundas em sua estrutura. Embora um *code smell* não seja necessariamente um erro, ele sugere que o código pode se tornar difícil de entender, manter ou expandir no futuro, funcionando como uma indicação para a fatoração do código (FOWLER; BECK, 2018).

A detecção manual desses problemas exige conhecimento técnico, atenção, e interpretação pessoal, o que nem sempre é viável, especialmente em contextos educacionais ou projetos de grande escala. Nesse cenário, surgem ferramentas e abordagens automatizadas para identificação de *code smells*, utilizando desde regras heurísticas até técnicas mais avançadas como *Machine Learning* (DI NUCCI et al., 2018)

A relevância da identificação de *code smells* não está apenas na correção do problema, mas na promoção de boas práticas de desenvolvimento e na formação de profissionais mais conscientes da qualidade de seus códigos (FOWLER; BECK, 2018). Para programadores iniciantes, como os que utilizam plataformas educacionais como o App Inventor, compreender e evitar *code smells* desde cedo pode contribuir significativamente para seu aprendizado e evolução técnica (Patton, 2019).

Existem diversos tipos de *code smells*, que quando presentes em projetos de software, podem comprometer a manutenibilidade, legibilidade e eficiência do código. Baseado no estudo *Code smells and refactoring: A tertiary systematic review of challenges and observations* que sumariza diferentes estudos, aponta como os dez mais comuns *code smells* sendo: *Duplicated Code/Clones*, *Large Class/God Class*, *Feature Envy*, *Long Method*, *Long Parameter List*, *Divergent Change*, *Data Clumps*, *Refused Bequest*, *Shotgun Surgery*, *Lazy Class*. (LACERDA et al., 2020).

2.2.1 Code Smells em VPL

Recentemente, estudos como o de Yadav e Cooper (2017) destacam o ensino de computação como uma competência essencial para o futuro. No entanto, conforme apresentado por Yang et al. (2025), ainda existem desafios importantes, especialmente relacionados à idade ideal para introduzir crianças à programação de forma eficaz. Em um estudo com crianças de aproximadamente 8.5 anos, observou-se que, apesar de uma melhora significativa na habilidade de escrever código após 24 aulas utilizando uma linguagem de programação visual, não houve progresso relevante em termos de pensamento computacional. Essa limitação é particularmente relevante, pois o pensamento computacional é fundamental para a compreensão e identificação de *code smells*.

Code smells, quando não tratados, afetam negativamente diversos pontos do código como já discutidos anteriormente. No contexto de linguagens de programação visual (VPLs) voltadas ao aprendizado de computação, é essencial oferecer suportes como a indicação de quando tratar *code smells*, por exemplo, refatorar que faz com que o código fique mais fácil de entender (FOWLER; BECK, 2018). Ao facilitar a leitura e a estrutura do código, essas práticas contribuem para que crianças não se sintam confusas ou frustradas, evitando o abandono do processo de aprendizagem, promovendo uma experiência mais positiva e motivadora (HERMANS; AIVALOGLOU, 2016).

Analisando os dados apresentados na Tabela 2 por Galante e Hauck (2025), é possível observar que certos *code smells* aparecem com maior frequência nos trabalhos acadêmicos do que outros. Um exemplo recorrente é o *Duplicated Code*, que ocorre quando blocos semelhantes são repetidos de forma desnecessária, comprometendo a clareza e a manutenção do programa (HERMANS; AIVALOGLOU, 2016). Outro *smell* comum é o *Dead Code*, cuja alta incidência chama atenção, especialmente considerando a simplicidade de sua detecção. Li et al. (2017), por exemplo, destacam que o App Inventor deveria fornecer mais visibilidade para funções não utilizadas. Já o *Long Method*, em linguagens de programação visual, manifesta-se como cadeias extensas de blocos conectados, o que dificulta o entendimento da lógica do código e compromete sua legibilidade (HERMANS; AIVALOGLOU, 2016).

Tabela 2. *Code smells* encontrados em trabalho realizado por Galante e Hauck

Code Smell	Qtd	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S12BS12F
Duplicated Code	12	X		X	X	X	X	X		X	X	X	X	X
Dead Code	9	X			X	X	X	X	X		X	X		X
Long Method	8				X		X			X	X	X	X	X
Code Clone	4	X	X	X										X
Feature Envy	3				X		X							X
Large Class	1										X			
Spaghetti Code	1												X	
Long Param List	1				X									
Lazy Class	1				X									

Fonte: Galante e Hauck (2025)

2.2.2 Principais *Code Smells*

Nesta seção, são apresentados os principais *code smells* identificados na literatura com impacto sobre VPL's, utilizando por base os estudos agrupados no trabalho de Galante e Hauck (2025).

A seguir são demonstrados os conceitos dos nove *smells* mais comuns com base em Fowler (2018).

Duplicated Code / Code Clone

Código duplicado ocorre quando uma mesma estrutura ou trecho de lógica aparece repetidamente em diferentes partes do código, como em múltiplos métodos de uma mesma classe ou em classes distintas. Essa repetição pode comprometer a manutenibilidade do software, pois qualquer modificação futura exige que todas as cópias sejam atualizadas de forma consistente. Caso contrário, o sistema pode apresentar comportamentos inconsistentes ou falhas. Além disso, a presença de código duplicado aumenta a complexidade da revisão e dificulta a compreensão geral da aplicação. A refatoração para extrair esse trecho repetido em um método único reduz o esforço de manutenção e melhora a legibilidade do código.

Dead Code

Código morto refere-se a trechos de código que nunca são executados, seja

porque estão comentados, isolados, ou porque não são mais chamados em nenhuma parte do sistema. Não costumam impactar diretamente o desempenho, pois tendem a ser eliminados pelo compilador. Esses fragmentos ainda ocupam espaço no código-fonte e podem confundir outros desenvolvedores. O tempo gasto para ler, tentar entender ou verificar a utilidade desse código pode comprometer a produtividade e a manutenção do software. No passado, era comum manter trechos de código comentados com a justificativa de possível reutilização futura. No entanto, com o uso de sistemas de controle de versão, essa prática tornou-se obsoleta, já que qualquer versão anterior do código pode ser facilmente recuperada, eliminando a necessidade de manter código morto no projeto.

Long Method/Function

Refere-se a funções excessivamente longas, que tendem a dificultar a compreensão, manutenção e reutilização do código. Embora no passado o custo de chamadas de funções fosse um fator técnico relevante, atualmente as linguagens modernas eliminaram esse impacto, deixando como principal obstáculo o esforço cognitivo exigido do programador, que precisa alternar mentalmente entre diferentes trechos de código para compreender sua lógica. Fowler (2018) argumenta que programas compostos por pequenas funções tendem a ser mais legíveis e sustentáveis ao longo do tempo, mesmo que, a princípio, causem estranhamento por parecerem apenas uma longa sequência de delegações.

Uma prática recomendada para identificar trechos que devem ser extraídos é observar quando há necessidade de escrever um comentário: nesse caso, é indicado criar uma nova função com uma boa nomenclatura. Essa abordagem contribui para a legibilidade, pois mantém cada trecho de código melhor atrelado a sua própria semântica. Além disso, nomes de funções bem escolhidos frequentemente eliminam a necessidade de o leitor examinar seu conteúdo interno. Em geral, é dito para extrair funções de forma bem agressiva, pois a clareza do código tende a melhorar com menores funções.

Feature Envy

Demonstra forte dependência de dados ou comportamentos de outra classe, acessando frequentemente seus atributos ou invocando muitos de seus métodos. Em sistemas bem modularizados, espera-se que os métodos interajam

prioritariamente com membros de sua própria classe, promovendo coesão interna e baixo acoplamento. Quando essa relação é invertida, o método revela uma "inveja" das funcionalidades de outra classe, indicando que possivelmente está localizado na classe errada.

A mais simples abordagem para corrigir esse problema é mover o método para a classe cujos dados ele utiliza com mais frequência. Em alguns casos, apenas uma parte do método apresenta essa dependência externa. Nesses cenários, recomenda-se primeiro extrair essa porção em um novo método, e então transferi-lo para a classe apropriada.

Large/God Class

God Class é um *code smell* caracterizado por uma classe que concentra responsabilidades demais, acumulando diversos métodos e atributos que frequentemente pertencem a contextos distintos. Essas classes tendem a conter métodos longos, listas extensas de parâmetros e um alto grau de acoplamento interno, tornando difícil compreender, testar e manter o código. Muitas vezes, os métodos utilizam apenas uma parte dos atributos disponíveis, evidenciando que a classe está assumindo múltiplas funcionalidades que deveriam estar distribuídas entre componentes especializados.

Essa centralização excessiva compromete os princípios de coesão e separação de responsabilidades, além de dificultar a evolução do sistema. Para resolver esse problema, uma abordagem comum é identificar agrupamentos de métodos que compartilham os mesmos dados e, a partir disso, extrair novas classes mais coesas. Alternativamente, pode-se criar uma superclasse contendo os atributos e comportamentos comuns, promovendo uma hierarquia mais adequada e modular.

Spaghetti Code

Embora não formalizado como um code smell no catálogo de Fowler (2018), o termo Spaghetti Code é amplamente utilizado na literatura para descrever trechos de código com estrutura desorganizada, ausência de funções, ou funções longas e com acesso a variáveis globais, sem modularização adequada e com fluxo de controle confuso, dificultando a leitura, compreensão e manutenção. Esse tipo de código geralmente resulta da combinação de diversos smells, como métodos

excessivamente longos, alto acoplamento, falta de coesão e aninhamentos profundos, reunidos em um único contexto mal estruturado. Robert C. Martin (2009), ao discutir as qualidades de um “código limpo”, critica fortemente essas práticas, mesmo sem utilizar diretamente o termo "*Spaghetti Code*". Tanto ele quanto Fowler enfatizam princípios que visam justamente evitar esse tipo de degradação estrutural no código.

Long Parameter List

Long Parameter List é um *code smell* que ocorre quando uma função ou método recebe um número excessivo de parâmetros. Esse problema costuma surgir na tentativa de evitar o uso de variáveis globais, mas acaba tornando o código mais complexo, difícil de entender e de modificar. Funções com muitas entradas dificultam a leitura, aumentam a chance de erros e comprometem a manutenção do software.

Para mitigar esse problema, algumas estratégias são recomendadas. Uma delas é analisar os parâmetros para identificar quais possuem uma fonte comum e, em vez de passar vários valores separadamente, encapsulá-los em um objeto ou estrutura que represente essa entidade. Outra abordagem consiste em passar apenas um identificador ou referência que permita buscar internamente o objeto necessário, reduzindo o número de parâmetros explícitos.

Quando os parâmetros incluem flags que alteram o comportamento da função, pode ser interessante dividir a função em múltiplos métodos específicos para cada caso, aumentando a clareza e facilitando testes e manutenção.

Lazy Class/Element

Em alguns casos, desenvolvedores antecipam que certas partes do sistema crescerão no futuro, criando desde cedo classes ou estruturas separadas em preparação para essas possíveis expansões. No entanto, quando essas evoluções não ocorrem, acabam permanecendo no código elementos com pouca ou nenhuma funcionalidade real, como classes contendo apenas um atributo ou sequer possuindo métodos. Esse cenário também pode ser resultado de refatorações intensas, nas quais uma classe outrora útil acaba reduzida até um ponto que sua existência deixa de fazer sentido. Nestes casos, recomenda-se incorporar a

funcionalidade dessa classe a outra mais relevante, deixando o código mais simples, coeso e compreensível.

2.3 MACHINE LEARNING (ML)

A ideia de simular a inteligência humana remonta à Antiguidade, sendo discutida desde os silogismos de Sócrates até os autômatos mecânicos. No entanto, foi apenas em 1956 que o termo inteligência artificial foi formalmente utilizado no campo da computação, por John McCarthy, durante a conferência de Dartmouth, considerada o marco inicial da área (RUSSELL; NORVIG, 2015). McCarthy propôs que uma máquina pode ser considerada inteligente se for capaz de resolver problemas que, tradicionalmente, exigiriam inteligência humana.

Com o tempo, surgiu o conceito de aprendizado de máquina (*machine learning*, ML), o que levou à distinção entre dois campos da inteligência artificial: a IA simbólica e a IA conexionista (Luger, 2009). O ML emergiu como resposta às limitações dos primeiros sistemas de IA, que eram baseados exclusivamente em regras fixas e conhecimento previamente codificado, características típicas da IA simbólica. Essa nova abordagem representou uma mudança de paradigma: em vez de programar explicitamente todos os comportamentos, os sistemas passaram a ser projetados para aprender a partir de dados, identificando padrões e tomando decisões com base nessas observações, princípio central da IA conexionista (Goodfellow et al., 2016).

2.3.1 Tipos de Aprendizado de Máquina

O aprendizado supervisionado ocorre quando os dados de entrada são fornecidos juntamente com seus respectivos rótulos, ou seja, o sistema aprende a associar entradas a saídas conhecidas. Durante o processo de treinamento, o modelo ajusta seus parâmetros com base nesses pares de entrada e saída, buscando generalizar para novos dados semelhantes. Para garantir a eficácia do modelo, o conjunto de dados costuma ser dividido em dois subconjuntos: um para o treinamento, onde o aprendizado acontece, e outro para a validação ou teste, utilizado para avaliar se o modelo é capaz de fazer previsões corretas em situações não vistas previamente (Goodfellow et al., 2016).

O aprendizado não supervisionado, diferentemente do supervisionado, seus dados de entrada não possuem rótulos. O objetivo do modelo é identificar estruturas ocultas, agrupamentos ou padrões nos dados sem qualquer informação prévia, não há um par de entrada e saída, sobre categorias ou classes. Técnicas como *clustering* (agrupamento) e redução de dimensionalidade são comumente utilizadas nesse tipo de aprendizado. Um exemplo clássico é o algoritmo *k-means*, que agrupa elementos semelhantes com base em suas características. Essa abordagem é especialmente útil quando não há conhecimento prévio suficiente para rotular grandes volumes de dados ou quando se deseja explorar padrões emergentes (Bishop, 2006).

Entre os dois paradigmas de aprendizado supervisionado e não supervisionado, há uma categoria intermediária conhecida como aprendizado auto-supervisionado (*self-supervised learning*), na qual o próprio modelo gera seus rótulos a partir de dados não rotulados. Nessa abordagem, partes textuais podem ser mascaradas, como no BERT (Devlin et al., 2019), ou removidas do final, como no GPT (Radford et al., 2018), então o modelo é treinado para prever os elementos ausentes com base no restante da informação disponível. Essa técnica permite o uso de grandes volumes de texto sem a necessidade de rotulação manual e tornou-se fundamental para o treinamento de modelos de linguagem de larga escala (*Large Language Model*), possibilitando que aprendam representações linguísticas complexas de forma autônoma.

O aprendizado por reforço (*Reinforcement Learning*, RL) é um paradigma de aprendizado de máquina baseado na interação entre um agente e um ambiente. Nesse contexto, o agente é uma entidade que toma decisões, por exemplo, um programa de computador, enquanto o ambiente representa o sistema com o qual ele interage. A cada passo, o agente observa o estado atual do ambiente, executa uma ação e, em troca, recebe uma recompensa (positiva ou negativa) e uma nova observação do ambiente.

O objetivo do agente é aprender uma política de ações que maximize a recompensa acumulada ao longo do tempo. Ao contrário do aprendizado supervisionado, o RL não depende de um conjunto fixo de exemplos rotulados, mas aprende por tentativa e erro, explorando diferentes possibilidades para melhorar seu desempenho. Essa abordagem é especialmente eficaz em problemas de tomada de decisão sequencial, sendo aplicada em áreas como jogos (como o xadrez), veículos

autônomos, controle robótico e sistemas de recomendação adaptativos (Sutton & Barto, 2018).

Nos modelos de linguagem de LLMs, o aprendizado por reforço é utilizado em etapas posteriores ao treinamento inicial, principalmente por meio de técnicas de *Reinforcement Learning from Human Feedback* (RLHF). Nesse processo, o modelo é ajustado com base em preferências humanas, avaliando, por exemplo, quais respostas são consideradas melhores. Essa técnica foi popularizada pela OpenAI no treinamento do InstructGPT e aplicada em versões do *ChatGPT*⁴ (Ouyang et al., 2022).

2.3.2 Large Language Model - LLM

Os modelos de linguagem de grande escala, conhecidos como *Large Language Models* (LLMs), representam uma evolução significativa no campo da inteligência artificial, especialmente no aprendizado de máquina aplicado à linguagem natural. Esses modelos são treinados em grandes volumes de texto (como livros, artigos, páginas da *web* e fóruns), utilizando técnicas baseadas principalmente em redes neurais profundas, como os *Transformers* (Vaswani et al., 2017). O objetivo principal de um LLM é aprender padrões linguísticos e realizar tarefas de processamento de linguagem natural (NLP), como tradução, sumarização, geração de texto, classificação e resposta a perguntas.

Os *Transformers* utilizam um mecanismo chamado *self-attention*, que permite ao modelo avaliar o peso de cada palavra em relação a todas as outras de uma sequência, independentemente da posição. Isso possibilita um entendimento mais profundo do contexto global da sentença, melhorando significativamente a capacidade do modelo de capturar relações semânticas e sintáticas entre termos e frases. (Vaswani et al., 2017)

Estudos como o de Brown et al. (2020) demonstraram que modelos de linguagem de grande escala, mesmo com bilhões de parâmetros, podem obter resultados mais assertivos em tarefas específicas, por meio de estratégias conhecidas como *prompt-based learning* na qual um pequeno número de exemplos é fornecido diretamente no *prompt* para orientar a realização da tarefa desejada

⁴ <https://openai.com/pt-BR/index/chatgpt/>

emerge o *few-shot learning*, ou seja, a capacidade de generalizar uma tarefa a partir de poucos exemplos.

Tradicionalmente, *few-shot learning* referia-se a modelos menores capazes de aprender novas tarefas com volumes limitados de dados. A inovação de Brown et al. (2020) consistiu em demonstrar que *few-shot learning* pode emergir naturalmente em modelos de grande escala já treinados. Nesse caso, o modelo não precisa de reajuste, são utilizados os chamados "*shots*", que referem-se à quantidade de exemplos explícitos incluídos na entrada textual: no *zero-shot*, nenhum exemplo é dado; no *one-shot*, apenas um; e no *few-shot*, alguns exemplos são usados como referência. Essa abordagem permite que o modelo generalize padrões de forma análoga àquela observada nos modelos menores, porém sem necessidade de um novo treinamento.

De forma complementar, Lewis et al. (2020) propuseram o *Retrieval-Augmented Generation* (RAG), uma arquitetura que combina a capacidade gerativa dos modelos de linguagem com a recuperação dinâmica de informações a partir de grandes bases de dados externas, como a *Wikipédia*. Diferentemente do *few-shot learning*, que depende exclusivamente do contexto fornecido no prompt, a arquitetura RAG incorpora documentos ou trechos relevantes durante o processo de geração, aumentando a precisão e buscando diminuir os casos de "alucinação" das LLM's.

2.3.3 Detecção de *code smells* com ML

A detecção de code smells por meio de técnicas de aprendizado de máquina tem sido explorada de diversas formas na literatura. Azeem et al. (2019) realizaram uma revisão sistemática que identificou 15 estudos que aplicaram aprendizado de máquina para detectar code smells, sendo *God Class*, *Long Method*, *Functional Decomposition* e *Spaghetti Code* foram os casos mais pesquisados. Dentre os algoritmos empregados, os mais comuns foram Árvores de Decisão e as Máquinas de Vetores de Suporte, enquanto que técnicas como *Random Forest* e *JRip* apresentaram os melhores desempenhos em termos de F-measure, com valores que chegaram a aproximadamente 83%. Contudo, no estudo é apontado que quando modelos são utilizados em diferentes projetos dos treinados, o desempenho

chega a ter 19% menos eficiência, sinalizando um nível significativo de *over-fitting*⁵ nos modelos.

Em complemento, Palomba et al. (2017) conduziram um estudo empírico com foco semelhante e observaram *code smells* recorrentes aos identificados por Azeem et al. (2019). Os autores também constataram bons resultados com os classificadores *Random Forest* e *B-JRip* (uma variação do algoritmo *JRip*), porém o algoritmo que apresentou desempenho superior de forma geral foi o *B-J48 Pruned*, alcançando aproximadamente 96% de acurácia na detecção de determinados *smells*.

⁵ **Overfitting:** Fenômeno em que um modelo de aprendizado de máquina se ajusta excessivamente aos dados de treinamento, capturando ruídos ou padrões específicos desses dados, o que compromete sua capacidade de generalização para novos dados.

3. TRABALHOS RELACIONADOS

Neste capítulo é realizado um mapeamento sistemático da literatura buscando encontrar trabalhos sobre identificação de *code smells* em programação por blocos com *machine learning*, se possível utilizando LLMs. Primeiro são definidos os protocolos de pesquisa, por fim, é realizada a extração e análise dos trabalhos correlatos

3.1 PROTOCOLO DE PESQUISA

Como identificar e analisar *code smells* em códigos VPL utilizando técnicas de Machine Learning?

QA1) Quais *code smells* foram identificados nos estudos?

QA2) Quais métodos de aprendizado de máquina foram utilizados para detecção?

QA3) Como os *code smells* foram tratados ou corrigidos nos estudos?

QA4) Qual o impacto da identificação e análise dos *code smells* sobre a qualidade do software?

QA5) Quais métricas foram adotadas na avaliação do estudo?

Critérios de inclusão/exclusão. Para garantir a relevância e a qualidade dos estudos analisados, foram estabelecidos alguns critérios de inclusão e exclusão. Foram incluídos apenas artigos publicados em inglês, com revisão por pares (*peer-reviewed*), que abordassem explicitamente a detecção e/ou análise de *code smells* por meio de técnicas de aprendizado de máquina. Também foram considerados estudos que envolvessem linguagens de programação visual (VPL) ou que apresentassem abordagens aplicáveis a esse contexto.

Foram excluídos artigos que não tratavam diretamente de *code smells*, que abordavam apenas refatoração sem relação com aprendizado de máquina. Trabalhos duplicados, documentos incompletos, apresentações de conferência sem artigo completo disponível, e publicações em idiomas distintos do inglês e português também foram desconsiderados.

Fontes de dados. As buscas foram realizadas nas principais bases científicas da área de Engenharia de Software e Computação a que alunos da Universidade Federal de Santa Catarina têm acesso, incluindo:

- IEEE Xplore (<https://ieeexplore.ieee.org>)
- ScienceDirect (<https://www.sciencedirect.com>)
- Scopus (<https://www.scopus.com>)
- Google Scholar (<https://scholar.google.com>)

A seleção dessas fontes se deu pela ampla cobertura de publicações da área de Ciência da Computação. O Google Scholar foi utilizado como apoio para identificar estudos adicionais fora dos indexadores formais.

String de buscas: As strings de busca foram elaboradas com base em palavras-chave relacionadas à pergunta de pesquisa e às quatro questões de análise (QA1–QA4). Os termos utilizados incluíram sinônimos e variações terminológicas que possibilitassem a recuperação de um conjunto abrangente de estudos. A *string* aplicada é: ("code smell" OR "bad smell") AND ("machine learning" OR "LLM") AND ("VPL" OR "visual programming language" OR "App Inventor").

Em complemento à uma primeira iteração de busca utilizando a *string* completa, não foram encontrados um número relevante de trabalhos, então parte da busca foi realizada removendo-se a parte de VPL. Focando assim a busca em possibilidades de aplicação de ML em identificações de *code smells* como apresentado na Tabela 3.

Tabela 3. Resultados da busca

Base de Dados	Resultados Iniciais	Segunda Filtragem
IEEE XPLORE	6	2
ScienceDirect	5	1
Scopus	5	2
Google Scholar	25	8

Fonte: Autor (2025)

Em cada uma das bases de dados selecionadas, foram avaliadas as 10 primeiras páginas de resultados retornadas pela aplicação da string de busca. A seleção inicial dos trabalhos foi realizada por meio da leitura dos títulos, leitura dos resumos e seções de resultados, de forma a identificar estudos com níveis de relevância em relação à pergunta de pesquisa.

Após essa triagem preliminar, foi executada uma análise mais refinada, considerando repetição de objetivos em diferentes trabalhos, bem como a clareza dos objetivos metodológicos de cada pesquisa. Trabalhos que abordam explicitamente a detecção de *code smells* com o uso de aprendizado de máquina, preferencialmente, mas não limitante, em contextos relacionados a linguagens visuais de programação (VPL), foram priorizados.

Tabela 4. Trabalhos encontrados

ID	Referência
SM1	CARAM et al., 2019
SM2	AZEEM et al., 2019
SM3	ARCELLI FONTANA et al., 2015
SM4	SADIK; GOVIND, 2025
SM5	WU et al., 2024
SM6	GUGGULOTHU; MOIZ, 2020
SM7	CRUZ; SANTANA; FIGUEIREDO, 2026
SM8	YAHYA; TAHA, 2023
SM9	KIYAK; BIRANT; BIRANT, 2019
M1	ELDAR SULTANOW et al., 2018
SV1	HERMANS; STOLEE; HOEPELMAN, 2016
SV2	HERMANS; EFTHIMIA AIVALOGLOU, 2016
SV3	TECHAPALOKUL; TILEVICH, 2019

Fonte: Autor (2025)

3.2 EXTRAÇÃO E ANÁLISE DOS DADOS

A partir das questões de análise definidas, dados relevantes dos estudos encontrados (Tabela 4) foram organizados e sintetizados no Quadro 2. Em relação a cada questão de análise QA1, QA2, QA3, QA4 e QA5 os dados foram distribuídos nos Quadros 3, 4, 5, 6 e 7 respectivamente. Também foi realizada uma comparação entre os resultados obtidos por trabalhos com temas semelhantes.

Quadro 2. Síntese dos estudos selecionados com base nas questões de análise

ID	Tema
SM1	Realizado um mapeamento sistemático da literatura voltado à detecção de <i>code smells</i> com o uso de técnicas de aprendizado de máquina. Como <i>smells</i> mais investigados destaca-se a categoria <i>Bloater</i> ⁶ , presente em 35% dos estudos analisados. A técnica mais frequentemente utilizada foi a aplicação de algoritmos genéticos, mencionada em 22,22% dos trabalhos. Em relação ao desempenho, os classificadores baseados em árvores de decisão, seguidos pelo <i>Random Forest</i> , destacaram-se como os mais eficazes.
SM2	Feito um mapeamento sistemático da literatura sobre detecção de <i>code smells</i> com aprendizado de máquina. <i>God Class</i> , <i>Long Method</i> , <i>Functional Decomposition</i> , e <i>Spaghetti Code</i> foram os <i>smells</i> mais estudados. As técnicas mais utilizadas encontradas foram <i>Decision Trees</i> e <i>Support Vector Machines</i> . Em termos de performance, <i>JRip</i> e <i>Random Forest</i> foram avaliados como mais eficientes.
SM3	Comparado o desempenho de 32 algoritmos de aprendizado de máquina na detecção de quatro tipos de <i>code smells</i> : <i>Data Class</i> , <i>Large Class</i> , <i>Feature Envy</i> e <i>Long Method</i> . Para isso, foram utilizadas 1986 instâncias rotuladas manualmente com o apoio de ferramentas gratuitas de análise estática, como o <i>iPlasma</i> (MARINESCU et al., 2005 apud ARCELLI FONTANA et al., 2015). Embora todos os modelos tenham apresentado bom desempenho, os algoritmos <i>J48</i> e <i>Random Forest</i> obtiveram os melhores resultados, enquanto <i>Support Vector Machines</i> apresentou o pior desempenho entre os avaliados. É destacado também o fato de que a baixa frequência de <i>code smells</i> no conjunto de dados pode ter influenciado a performance dos modelos.
SM4	Análise comparativa entre duas LLMs, <i>OpenAI GPT-4.0</i> e <i>DeepSeek-V3</i> , o dataset escolhido tinha <i>code smells</i> pré rotulados, assim como programas em 4 linguagens de programação, <i>Java</i> , <i>Python</i> , <i>JavaScript</i> e <i>C++</i> , buscando assim uma comparação não dependente de linguagens. O modelo <i>GPT-4.0</i> conseguiu uma maior precisão, 0.79 contra 0.42 do <i>DeepSeek</i> , contudo, o <i>recall</i> do <i>GPT-4.0</i> se manteve em 0.41, indicando que vários <i>smells</i> não foram detectados. Como conclusão, acredita-se que as LLMs ainda não estejam preparadas para substituir ferramentas determinísticas.
SM5	Propõe a arquitetura <i>iSMELL</i> , composta por uma etapa inicial de detecção de <i>code smells</i> com o modelo <i>Mixture of Experts</i> , que utiliza ferramentas especializadas para diferentes tipos de <i>smells</i> . Seguida pela refatoração dos códigos identificados, que é realizada com o apoio de LLMs, utilizadas apenas

⁶ Bloaters indicam que algo cresceu muito e sem planejamento, dificultando a manutenção (CARAM et al., 2019), como exemplos os já explicados *Large Parameter List*, *God Class* e *Long Method*.

	após a detecção automatizada, devido às limitações desses modelos na identificação de alguns casos de <i>smells</i>, como classes excessivamente grandes ou chamadas encadeadas que acessam muitas outras classes, causadas, por exemplo, pela limitação de 4000 <i>tokens</i> do GPT-3.5. Os resultados mostram que o <i>iSMELL</i> superou o uso direto de LLMs para identificação, com melhorias de 35,05% no F1-Score e 28,47% na acurácia.
SM6	Aborda a detecção de <i>code smells</i> como um problema de classificação <i>multi-label</i>, propondo um modelo que considera a correlação entre diferentes smells, segundo o estudo ainda pouco explorado na literatura até 2020. Utilizando os <i>code smells Long Method</i> (LM) e <i>Feature Envy</i> (FE), foi observada uma correlação de 61,7% entre LM e FE, e 71,4% no sentido oposto. Foram testados três métodos de classificação: Relevância Binária (BR), Cadeia Classificadora (CC) e Combinação de Rótulos (LC). O modelo BR, por não considerar correlações, alcançou 92,7% de acurácia, enquanto LC e CC obtiveram 96% e 95,5% respectivamente.
SM7	Investiga o impacto do <i>feedback</i> de especialistas na detecção de <i>code smells</i>, utilizando o classificador <i>Random Forest</i>. Foram avaliados quatro smells: <i>God Class</i>, <i>Long Method</i>, <i>Feature Envy</i> e <i>Refused Bequest</i>. O modelo realiza uma predição inicial, envia os casos positivos a especialistas, e utiliza as respostas para rotular novos dados por <i>oversampling</i>, então retreina o modelo. Dois cenários foram analisados: um com <i>feedback</i> em uma única rodada, variando a porcentagem de respostas e o grau de <i>oversampling</i>; e outro com <i>feedback</i> contínuo, em 50 iterações, com 10% dos casos avaliados por rodada. No cenário de uma rodada, os melhores resultados ocorreram com 70% de <i>feedback</i> e apenas 1% de <i>oversampling</i>, elevando o F1 em 0.1 para <i>God Class</i> e 0.26 para os restantes <i>code smells</i>. No modelo contínuo, o incremento de F1 variou de 0,13 (<i>God Class</i>) até 0.72 (<i>Feature Envy</i>).
SM8	Avalia modelos de <i>Deep Learning</i> supervisionado para detecção de três <i>code smells</i>: <i>God Class</i>, <i>Feature Envy</i> e <i>Long Method</i>. Foram testadas arquiteturas de Redes Neurais Recorrentes (RNN), <i>Long Short Term Memory</i> (LSTM) e <i>Gated Recurrent Unit</i> (GRU). Os resultados mostram desempenho elevado em todas as métricas, com destaque para a LSTM e RNN, que alcançaram 100% de precisão em <i>God Class</i> e <i>F1-Scores</i> de 98% (<i>God Class</i>), 98% (<i>Feature Envy</i>) e 97% (<i>Long Method</i>). As redes RNN e GRU também apresentaram bons resultados, com F1 variando entre 94% a 97% para os três smells.
SM9	Avaliou dois métodos de rotulagem: Cadeia Classificadora (CC) e Combinação de Rótulos (LC) aplicados a cinco algoritmos classificadores para detecção de <i>code smells</i>. Dentre os modelos testados, <i>Random Forest</i> apresentou o melhor desempenho, mantendo alta acurácia e <i>F1-Score</i> de forma consistente, independentemente de os smells estarem presentes em nível de classe ou de método, e também em ambos os métodos de rotulagem utilizados.
M1	Propõe o uso de <i>Machine Learning</i> para realizar análise estática de código sem depender de regras explícitas, com o objetivo de identificar problemas não reconhecidos por métodos tradicionais. Dois identificadores foram treinados de forma não supervisionada, buscando capturar padrões implícitos no código. O modelo é capaz de analisar repositórios em tempo real, avaliando novos <i>commits</i> com base no pressuposto de que a versão anterior do código estava "limpa". Em um teste prático, o sistema identificou corretamente um método <i>Java</i> que deveria ser anotado como não-nulo, sugerindo uma possível correção.
SV1	Busca demonstrar a existência de <i>code smells</i> em linguagens de programação visual, analisando 44 programas desenvolvidos em <i>LEGO MINDSTORMS EV3</i> e <i>Kodu</i> por dois grupos distintos: crianças e membros da comunidade em geral. Os resultados indicam que smells estão presentes em 88% dos programas EV3

	e em 93% dos programas Kodu.
SV2	Experimento com 61 crianças entre 12 e 14 anos, iniciantes em <i>Scratch</i> , divididas em três grupos: um com programas sem <i>smells</i> , outro com <i>duplicated code</i> , e o terceiro com <i>long method</i> . Os participantes realizaram tarefas de compreensão e modificação de programas, avaliadas por tempo e corretude. Os resultados indicam que a presença de <i>smells</i> reduziu significativamente a corretude das respostas, contudo não tenha impactado fortemente o tempo de execução. O <i>Long Method</i> afetou principalmente a compreensão geral do sistema, enquanto <i>duplicated code</i> a modificação dos programas.
SV3	Desenvolveu quatro estratégias de refatoração para o <i>Scratch</i> : Extrair Bloco Customizado, Extrair <i>Sprite</i> Pai, Extrair Constante e Reduzir Escopo da Variável, aplicadas sobre um conjunto de 448 projetos para tratar <i>code smells</i> específicos. Uma pesquisa com 24 participantes avaliou o impacto da ferramenta no apoio à programação. Em termos de eficácia, a extração de bloco corrigiu 79% dos casos de <i>Duplicated Code</i> , a extração de <i>sprite</i> pai tratou 11% de <i>Duplicated Sprite</i> , a extração de constantes removeu totalmente os <i>Duplicated Constants</i> e a redução de escopo solucionou 81% dos casos de <i>Broad Scope Variable</i> .

Fonte: Autor (2025)

Quadro 3. *Code smells* identificados por estudo

ID	Code smells identificados
SM1	<i>BLOB, Data Class, Data Clumps, Divergent Change, Duplicated Code, Feature Envy, Functional Decomposition, God Class, Large Class, Lazy Class, Long Method, Long, Message Chains, Middle Man, Parallel Inheritance Hierarchies, Parameter List, Shotgun Surgery, Spaghetti Code, Speculative Generality, Switch Statements</i>
SM2	<i>Antisingleton, Class Data should be Private, Complex Class, Data Class, Delegator, Duplicated Code, Feature Envy, Functional Decomposition, God Class, Large Class, Lazy Class, Long Method, Long Parameter List, Message Chain, Middle Man, Refused Parent Bequest, Spaghetti Code, Speculative Generality, Swiss Army Knife, Switch Statement</i>
SM3	<i>Data Class, Feature Envy, Large Class, Long Method</i>
SM4	<i>Alternative Classes with Different Interfaces, Comments, Data Class, Data Clumps, Dead Code, Divergent Change, Duplicated Code, Feature Envy, Inappropriate Intimacy, Large Class, Lazy Class, Long Method, Long Parameter List, Message Chains, Middle Man, Parallel Inheritance Hierarchies, Primitive Obsession, Refused Bequest, Shotgun Surgery, Speculative Generality, Switch Statements, Temporary Field</i>
SM5	<i>Feature Envy, God Class, Refused Bequest</i>
SM6	<i>Feature Envy, Long Method</i>
SM7	<i>Feature Envy, God Class, Long Method, Refused Bequest</i>
SM8	<i>Feature Envy, God Class, Long Method</i>
SM9	<i>Data Class, Feature Envy, God Class, Long Method</i>

M1	-
SV1	<i>Dead Code, Deprecated Interface, Duplicated Code, Feature Envy, Inappropriate Intimacy, Lazy Class, Long Method, Many Parameters, Message Chain, No-op, Unused Field</i>
SV2	<i>Duplicated Code, Long Method</i>
SV3	<i>Broad Scope Variable, Duplicated Code, Duplicated Constant, Duplicated Sprite</i>

Fonte: Autor (2025)

Quadro 4. Métodos de aprendizado de máquina para detecção por estudo

ID	Métodos de ML apresentados
SM1	<i>Genetic Algorithms, Naive Bayes Classifier, Association Rules, Nearest Neighbor, Bayesian Networks, Clustering, Linear Statistics, Semi-supervised, Random Forest, Support Vector Machine, Directed Acyclic Graph, Linear discriminant analysis, Decision Tree, Text-Based</i>
SM2	Decision Tree, Support Vector Machine, Random Forest, Naive Bayes, JRip, Bayesian Belief Networks, Sequential Minimal Optimization, Instance-Based, Voting Feature Interval, Logistic Regression, Immune-based Detection Strategy, Fuzzy Rule Learner, Polynomial Regression, Random Forest Regression, Recursive Neural Network, Natural language processing, Linear Regression
SM3	<i>J48 Pruned, J48 Unpruned, J48 Reduced Error Pruning, JRip, Random Forest, Naive Bayes, SMO RBF Kernel, SMO Poly Kernel, LibSVM C-SVC Linear Kernel, LibSVM C-SVC Polynomial Kernel, LibSVM C-SVC Radial Kernel, LibSVM C-SVC Sigmoid Kernel, LibSVMv-SVC Linear Kernel, LibSVMv-SVC Polynomial Kernel, LibSVM v-SVC Radial Kernel, LibSVM v-SVC Sigmoid Kernel.</i> Para cada um dos algoritmos utilizados, também foi considerada uma versão melhorada, dobrando o número de identificações.
SM4	<i>Large Language Models - OpenAI GPT-4.0 e DeepSeek-V3</i>
SM5	<i>Large Language Models - Apenas para refatoração de código</i>
SM6	<i>J48 Pruned, Random Forest, B-J48 pruned, B-J48 UnPruned, B-Random Forest</i>
SM7	<i>Random Forest</i>
SM8	<i>Gated Recurrent Units, Long Short Term Memory, Recurrent Neural Network</i>
SM9	<i>Decision Tree, Naive Bayes, Neural Network, Random Forest, Support Vector Machine</i>
M1	Apenas cita que parte de análise utiliza modelo não supervisionado
SV1	Não se aplica
SV2	Não se aplica
SV3	Não se aplica

Fonte: Autor (2025)

Quadro 5. Tratamento de *code smells* por estudo

ID	Tratamento de <i>code smells</i>
SM1	Não se aplica
SM2	Não se aplica
SM3	Não tratado
SM4	Não tratado
SM5	Utilizou ferramentas especializadas clássicas para identificação, a refatoração foi realizada por LLM, observando um prompt particular para cada caso, contudo, <i>code smells</i> em classes impactam o sistema todo, o que pode fazer com que muitas iterações de refatoração pela LLM seja necessário, tornando o processo custoso, então apenas smells em nível de métodos foram refatorados
SM6	Não tratado
SM7	Não tratado
SM8	Não tratado
SM9	Não tratado
M1	Não se aplica
SV1	Não tratado
SV2	Destaque para o trabalho de Moreno et al. (apud HERMANS; AIVALOGLOU, 2016), que forneceu <i>feedback</i> automático a crianças de 10 a 14 anos, resultando em melhoria na qualidade do código.
SV3	Foram desenvolvidos módulos para o Scratch capazes de extrair blocos, sprites, constantes ou reduzir o escopo de uma variável. Disponibilizando assim mais ferramentas para que usuários utilizem eliminando <i>code smells</i> identificados

Fonte: Autor (2025)

Quadro 6. Impacto da identificação e análise dos *code smells* por estudo

ID	Impacto da identificação e análise dos <i>code smells</i>
SM1	Não se aplica
SM2	Não se aplica
SM3	Descoberta de algoritmos que alcançaram acurácia >95% com apenas 100 exemplos de treino. Variância insignificante de performance em uso prático dos melhores algoritmos do estudo
SM4	O modelo GPT apresentou melhores resultados em termos de análise, porém, considerando o custo, observa-se um <i>trade-off</i> : o <i>DeepSeek</i> pode ser mais vantajoso financeiramente em projetos de baixa complexidade, mesmo com desempenho inferior

SM5	A identificação se mostrou muito mais eficiente que todos os métodos comparados. A refatoração não foi comparada, apenas métricas quantitativas conhecidas na literatura como WMC (<i>Weighted Methods per Class</i>), DIT (<i>Depth of Inheritance Tree</i>), NOC (<i>Number of Children</i>), entre outras, foram avaliadas, mostrando uma boa refatoração na maioria dos casos
SM6	Dos métodos utilizados para classificação: Relevância Binária (BR), Cadeia Classificadora (CC) e Combinação de Rótulos (LC). O modelo BR, obteve um resultado fraco, sendo desconsiderado como uma boa opção para <i>multi-label</i>
SM7	A identificação de <i>code smells</i> possibilitou provar que a evolução do modelo de <i>Random Forest</i> realmente aconteceu ao passar das iterações. Também observar que quanto mais <i>feedback</i> é dado pelo especialista melhor foram as identificações
SM8	Os três modelos de <i>Deep Learning</i> mostraram taxas de identificação e métricas como F1-Score muito altas para os 3 <i>code smells</i> estudados
SM9	<i>Random Forest</i> foi apontado como o algoritmo mais eficaz dentre os cinco para <i>multi-label</i> . Também foi notado que, de maneira leviana, o método de Cadeia Classificadora (CC) obteve melhor resultado de maneira geral que a Combinação de Rótulos (LC)
M1	Não se aplica
SV1	É identificado que <i>smells</i> clássicos inspirados em orientação à objetos ocorrem também em linguagens de bloco. Identificada necessidade de adaptar conceitos clássicos de <i>code smells</i> para VPL
SV2	Recolhimento de dados mostrando a relação de <i>smells</i> com menor entendimento do código em VPLs quando comparados com grupos que receberam códigos limpos
SV3	A identificação é essencial para a refatoração dos blocos, contudo ela é pouco abordada no estudo

Fonte: Autor (2025)

Quadro 7. Métricas adotadas nos estudos para análise de desempenho da identificação de code smells

ID	Impacto da identificação e análise dos <i>code smells</i>
SM1	<i>Precision</i> (13x), <i>Recall</i> (12x), <i>F-measure</i> (6x)
SM2	<i>Precision</i> (11x), <i>Recall</i> (10x), <i>F-measure</i> (3x), <i>Accuracy</i> (3x), <i>ROC area</i> (3x), <i>Positive predictive value</i> (1x), <i>Negative predictive value</i> (1x), <i>Tempo</i> (1x), <i>Especificidade</i> (1x)
SM3	<i>Accuracy</i> , <i>F-Measure</i> , <i>ROC area</i>
SM4	<i>Precision</i> , <i>Recall</i> , <i>F1-Score</i>
SM5	<i>Accuracy</i> , <i>F1-Score</i> , <i>Precision</i> , <i>Recall</i>
SM6	<i>Accuracy</i> , <i>F-Measure</i> , <i>ROC area</i>

SM7	<i>Precision, Recall, F1-Score</i>
SM8	<i>Accuracy, Precision, Recall, F1-Score, MAE, RMSE, RAE, RRSE</i>
SM9	<i>Accuracy, F1-Score, ROC area, Hamming Score, Exact Match Ratio</i>
M1	Não se aplica
SV1	Não se aplica
SV2	Não se aplica
SV3	Não se aplica

Algumas discussões a seguir são levantadas referentes aos trabalhos extraídos, houve o uso de IA (NotebookLM) para levantar possíveis tópicos e comparar partes de trabalhos. Os parágrafos não foram copiados da geração, todos os dados gerados foram conferidos em suas respectivas pesquisas.

Estudos citam a subjetividade intrínseca à identificação de *code smells* (SM1, SM2, SM3, SM5, SM6, SM7, SM9, SV3), frequentemente apontado como uma das principais motivações para o uso de técnicas de ML na identificação, pois há a delegação da identificação para o modelo. Da mesma forma, existem poucos estudos sobre o tema, e os *datasets* utilizados são bem variados (SM1, SM2) dificultando a comparação entre estudos. Ainda assim, há quase consenso sobre a eficácia de certos algoritmos como *Random Forest* (SM1, SM3, SM7, SM9) entre os mais diferentes estudos.

Trabalhos pioneiros como SM6 e SM9 destacam um campo a ser estudado em identificação de *smells*, o *multi-label*. Conforme indicado em SM2, *datasets* rotulados manualmente estão sujeitos a erros, deixando de englobar casos reais. A identificação de múltiplos *smells* simultaneamente também permite a correlação entre dados, como apresentado nos estudos já comentados, tendem a aumentar a eficácia dos métodos.

Há também a falta de padronização de *code smells* em estudos voltados para linguagens de programação visuais (SV1, SV2, SV3) pois apresentam interpretações próprias em relação a *smells*, normalmente sendo adaptado para cada ferramenta VPL utilizada, não foi encontrado uma literatura consolidada que estabeleça definições nesse contexto específico.

Poucos estudos analisados consideram as LLMs uma alternativa viável para a identificação de *code smells*, apenas a revisão da sistemática de SM1 menciona processamento de linguagem natural durante sua fundamentação teórica. Além disso, trabalhos como SM5 desencorajam o uso de LLMs para detecção devido às limitações na entrada e tokenização de estruturas de código complexas.

Em relação aos métodos de avaliação de desempenho, há pouca diversidade entre os estudos analisados, com mais diversidade em revisões sistemáticas da literatura (SM2). A maioria adota métricas padronizadas, como *Accuracy*, *Recall*, *Precision* e variações da *F-Measure*, especialmente o *F1-Score*. Entretanto, apesar de sua ampla utilização, alguns trabalhos (SM3, SM7) destacam problemas importantes, como o possível viés no uso de *Accuracy* em conjuntos de dados desbalanceados. Nesses casos, determinados *smells* podem ocorrer com baixa frequência, de modo que um modelo tendencioso a negar sua existência terá valores elevados de acurácia, mascarando a real efetividade do método de detecção.

Revisões sistemáticas (SM2) também apontaram a dificuldade em comparar diferentes trabalhos, devido à falta de padronização de dados e das análises realizadas, como apontado em SM6 por D Nucci et al., (2018), alguns conjuntos de dados não representavam fielmente o mundo real, indicando que os modelos tinham mais facilidade em indicar *code smells*.

4. IDENTIFICAÇÃO DE *CODE SMELLS* COM *LLMs*

Neste capítulo, é apresentado o dataset utilizado no projeto, detalhando parte de seu pré-processamento para execução do projeto. Também são fornecidos exemplos de *Lazy Class* na interface do *App Inventor*, assim como os resultados de um experimento inicial realizado com o conjunto de dados.

Todo código, apresentado, citado ou utilizado no desenvolvimento deste trabalho está disponível em: <https://codigos.ufsc.br/gqs/llm-code-smells>

4.1 PREPARAÇÃO DO *DATASET*

Para o desenvolvimento da proposta deste trabalho, foi utilizada uma base de dados já existente, pertencente ao laboratório GQS da UFSC, composta por 88863 projetos desenvolvidos na plataforma *App Inventor*, reunida a partir da galeria pública de aplicativos disponibilizada pelo próprio MIT. Essa coleção foi obtida por meio de um processo automatizado de coleta, com o apoio de um script que utilizou *links* diretos fornecidos por Jeffrey Schiller, resultando em um *dataset* bruto de arquivos no formato ".aia", extensão padrão dos projetos exportados do *App Inventor*.

Arquivos ".aia" podem ser abertos ao serem descompactados. Como ilustrado na Figura 3, o interesse recai sobre os arquivos ".bky" (*blockly*), que representam o código fonte visual da ferramenta e servem como base para a análise e identificação de *code smells*. Para isso, foi necessário realizar um processo de rotulagem em parte dos arquivos, com o objetivo de criar um conjunto de dados anotados e indexado para uso no mecanismo de recuperação do modelo RAG, auxiliando na geração de respostas relacionadas à presença ou ausência de *smells*.

O material disponibilizado para *download* pelo orientador deste trabalho, Prof. Dr. Jean Carlo Rossa Hauck sob licença *Creative Commons 4.0* é um subconjunto do total de aplicativos. Dessa forma, não houve trabalho sobre a obtenção do *dataset*, apenas sobre a etapa de anotação e preparação dos dados para indexação no sistema RAG.

Considerando o tamanho total da base (88863 aplicativos), foi realizado o cálculo amostral, para determinar a quantidade de exemplos a serem classificados pelos modelos de LLM. Foi escolhido um grau de confiança de 95% e uma margem

de erro de 5%, resultando em uma amostra de 383 aplicativos selecionados aleatoriamente.

4.2 TRATAMENTO DO DATASET

Os arquivos “.aia”, formato nativo de exportação de projetos da plataforma *App Inventor*, são arquivos comprimidos. Para acessar os dados de interesse, isto é, os dados com extensão “.bky”, cada arquivo precisa ser descompactado individualmente e então percorrer por estruturas de pastas como⁷:

```
/src/appinventor/ai_<usuário>/<nome_do_projeto>/
```

Nesta pasta, localizam-se os arquivos com a extensão “.bky”, cada arquivo representa uma tela criada pelo usuário, dentro dele há a lógica de programação dos aplicativos em formato baseado em blocos (*blockly*). Um *script* em *Python* (Quadro 8) foi desenvolvido para automatizar esse processo de extração, percorrendo os projetos e coletando os arquivos “.bky” para uma análise futura.

Quadro 8. Trecho do script de extração

```
for root, dirs, files in os.walk(caminho_raiz): # busca recursiva
    for arquivo in files:
        if arquivo.endswith('.bky'):
            origem = os.path.join(root, arquivo)
            subpath = os.path.relpath(root,
caminho_raiz).replace(os.sep, '_')
            # substitui espaços por underline
            subpath = subpath.replace(' ', '_')
            arquivo_limpo = arquivo.replace(' ', '_')
            nome_base_limpo = nome_base.replace(' ', '_')
            destino_nome =
f"{nome_base_limpo}_{subpath}_{arquivo_limpo}"
            destino = os.path.join(CAMINHO_SAIDA_BKY, destino_nome)
            shutil.copy2(origem, destino)
            encontrados += 1
            print(f"[✓] Extraído: {destino}")
```

Fonte: Autor (2025)

Os arquivos *Blockly* extraídos podem ser abertos diretamente, e seu conteúdo é organizado de forma semi-estruturados por meio de *tags* XML. Isso permite que o conteúdo seja analisado de forma legível tanto por humanos quanto por sistemas automatizados, como demonstrado no Quadro 9. Além disso, ao subir

⁷ Algumas versões diferentes de App Inventor geram outro caminho, mas todos foram incluídos no código

o arquivo “.aia” no ambiente do *App Inventor*, é possível observar a representação visual dos blocos, o que facilita a identificação dos elementos lógicos do programa, conforme ilustrado na Figura 5.

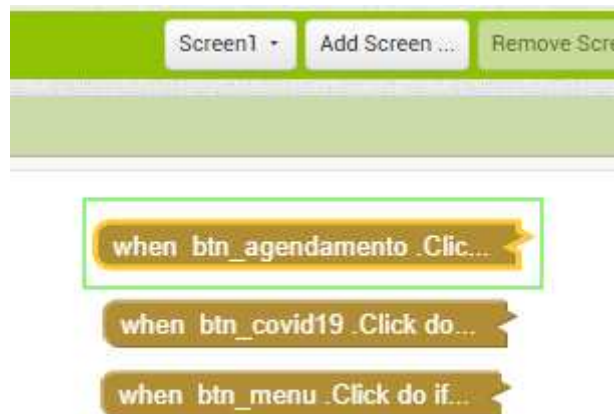
Quadro 9. Estrutura interna de um arquivo “.bky”

```
<xml xmlns="http://www.w3.org/1999/xhtml">
  <block type="component_event" id="nZCF{vcM;3oUMoToJssi" collapsed="true"
x="-871" y="-1179">
    <mutation component_type="Button" is_generic="false"
instance_name="btn_agendamento" event_name="Click"></mutation>
    <field name="COMPONENT_SELECTOR">btn_agendamento</field>
    <statement name="DO">
      <block type="controls_openAnotherScreen" id="HO+!z1Hu#)8^qer@N3ge">
        <value name="SCREEN">
          <block type="text" id="|IZwbRqdkBPUX$`P3B??">
            <field name="TEXT">Screen2</field>
          </block>
        </value>
      </block>
    </statement>
  </block>
  ...

```

Fonte: Autor (2025)

Figura 5. Representação App Inventor do arquivo “.bky” do Quadro 8



Fonte: Autor (2025)

Após a extração, foi realizada uma limpeza sobre os arquivos, com finalidade de remover seções que não agregam informação útil aos modelos, apagando o atributo “id” quando presente e também a *tag* padrão adicionada ao final de toda aplicação de *App Inventor* chamada “yacodeblocks”.

4.3 EXPERIMENTOS INICIAIS

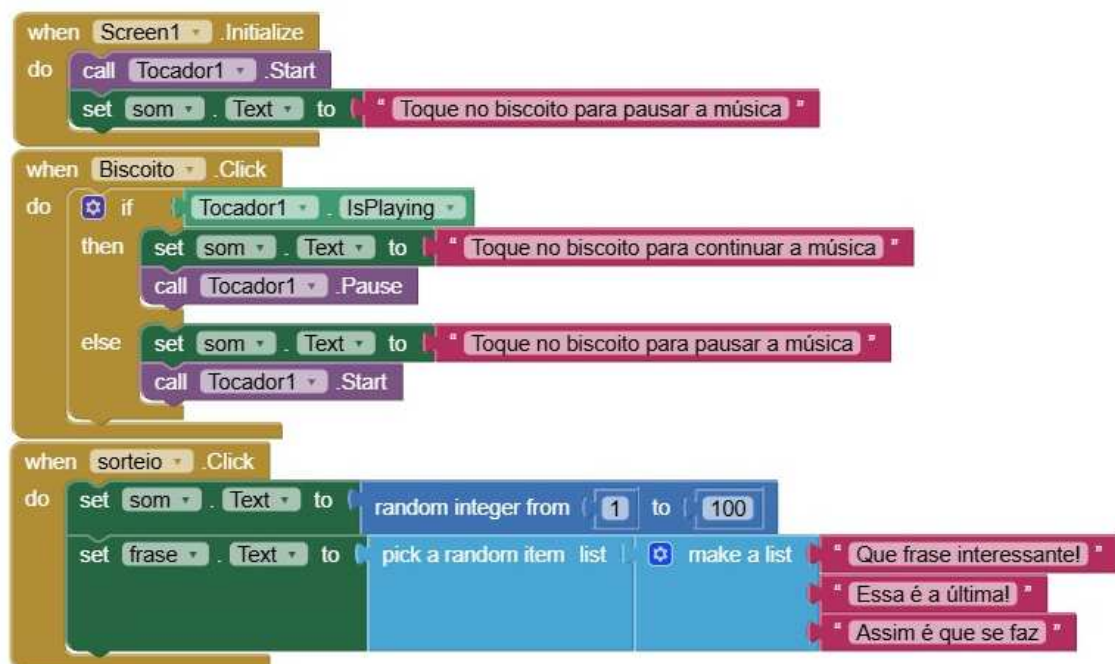
4.3.1 Estruturação dos casos de teste

De forma mais simples do que o RAG, o experimento inicial realizado utilizou o modelo de One Shot com 3 modelos de linguagem natural (LLMs), incluindo o *ChatGPT* (OpenAI, 2025), o *Gemini* (Google, 2025) e o *DeepSeek Chat* (DeepSeek AI, 2025).

Para preparar os *shots*, um arquivo “.bky“, que equivale a uma tela em um aplicativo, foi escolhido aleatoriamente do *dataset*. O trabalho foi então importado para o *App Inventor* e modificado propositalmente, de modo a introduzir um *smell* intencional. A Figura 6 mostra um conjunto de blocos original e na Figura 7 a inserção de *Lazy Class*.

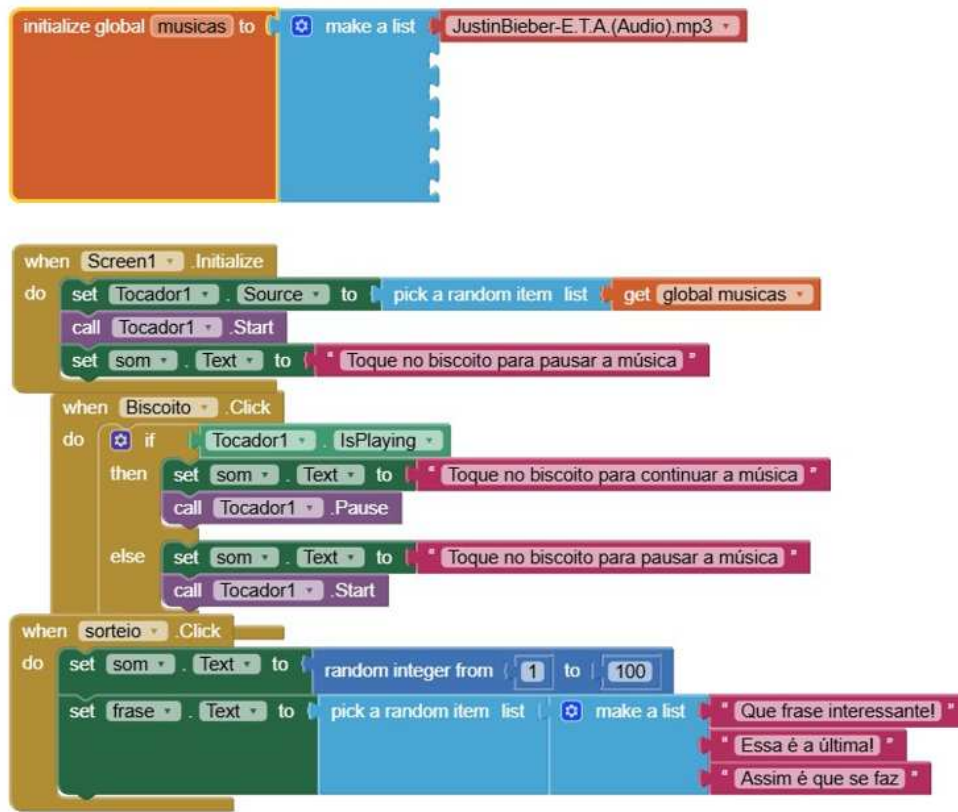
Lazy Class foi selecionado para os experimentos por se tratar de um tipo pouco explorado na literatura voltada a linguagens de programação visuais (VPLs). Entre os trabalhos relacionados, apenas Hermans, Stolee e Hoepelman (2016) abordaram esse *smell* em VPLs. Dessa forma, optou-se por aprofundar o conhecimento desse *smell*, geralmente negligenciado em estudos anteriores.

Figura 6. Conjunto de blocos sem *code smell* encontrado no *dataset*



Fonte: Autor (2025)

Figura 7. Conjunto de blocos com *Lazy Class* inserido no *dataset*



Fonte: Autor (2025)

Como o método adotado será do tipo *one-shot*, optou-se por compor uma base de exemplos a partir de dez arquivos “.bky” sem indícios aparentes de *code smells*. Para garantir que os programas selecionados apresentem algum grau mínimo de lógica e funcionalidade, foram considerados ao menos 5 com um número mínimo de linhas sendo 100.

A Figura 8 tem um exemplo de *Lazy Class*, caracterizado pela presença de um procedimento criado no projeto que nunca é invocado em nenhum ponto da execução. Esse elemento também pode ser interpretado como um caso de *Dead Code*, uma vez que permanece inativo. Contudo o nome do procedimento sugere que havia uma intenção de uso posterior, indicando um comportamento comum em situações onde estruturas são criadas antecipando funcionalidades que não foram implementadas.

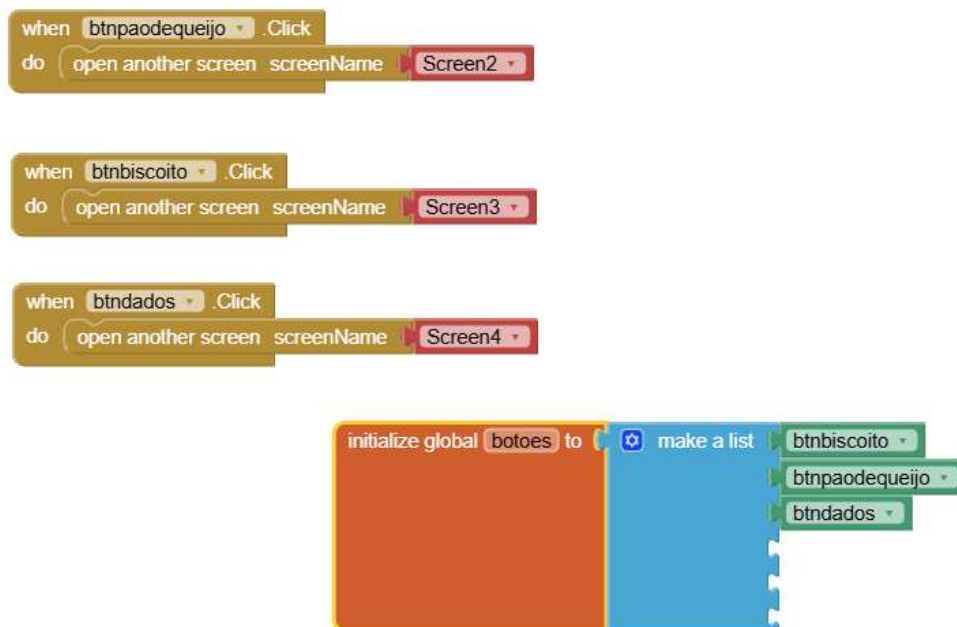
Figura 8. Aplicação que um cachorro late com *Lazy Class* inserido



Fonte: Autor (2025)

Já ilustrado na Figura 9 observa-se uma condição mais próxima da adicionada para o *shot* (Figura 7) com criação de uma instância de uma variável global, mas que neste caso não é utilizada, e embora tenha sentido com o programa foi deixado de lado.

Figura 9. Aplicação de valores nutricionais com *Lazy Class* inserido



Fonte: Autor (2025)

4.3.2 Aplicação dos exemplos em One-Shot Prompting

Para a realização do *one-shot*, foi elaborado um *prompt* inicial básico, sem uso de estratégias, que passou por três ciclos de refinamento com o apoio do modelo *ChatGPT* (OPENAI, 2025), até se alcançar a versão final apresentada no Quadro 10.

Quadro 10. Prompt usado para One-Shot Prompting

```
Exemplo de smell:
{
    Código .bky (XML) com parte do code smell selecionada
}

Smell identificado: Lazy Class
Justificativa: A variável global 'musicas' é inicializada como uma lista com capacidade para seis itens, porém apenas um item é realmente atribuído. Isso indica uma estrutura subutilizada, sugerindo um planejamento não concretizado ou abandono de funcionalidade. Tal característica configura o smell conhecido como Lazy Class.

Avalie este trecho:
```

Fonte: Autor (2025)

Juntamente com o *prompt* contendo o exemplo previamente rotulado, é enviado um código “.bky” inteiro e o modelo é questionado se há presença de *Lazy Class* no código, então se espera que as LLMs, com o conhecimento prévio somado ao exemplo passado seja capaz de resolver o problema com maior precisão e assertividade.

A Tabela 5 mostra as correlações com os modelos de LLM e suas respectivas respostas na ordem das perguntas. Ressaltando que as perguntas foram feitas a partir de modelos sem nenhuma conta ligada, ou contas que não tinham histórico nas plataformas.

Tabela 5. Respostas de LLMs (versões *web*) com One-Shot Prompt

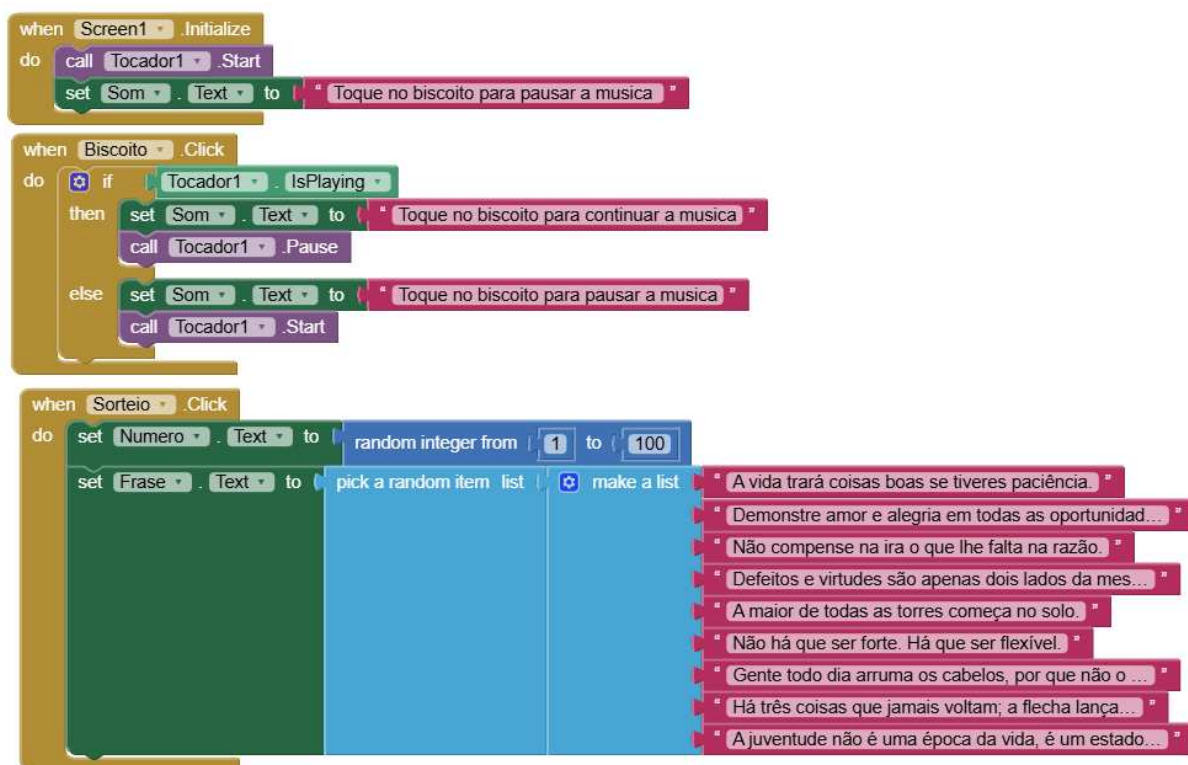
Número	Presença de <i>Lazy Class</i>	<i>ChatGPT 4.0</i>	<i>Gemini 2.5 flash</i>	<i>DeepSeek-R1</i>
1	Não	Não	Não	Não
2	Não	Sim	Não	Não
3	Sim	Sim	Sim	Sim

4	Não	Não	Não	Não
5	Não	Não	Não	Não
6	Sim	Sim	Sim	Sim
7	Não	Não	Não	Não
8	Não	Não	Não	Não
9	Não	Não	Sim	Não
10	Não	Sim	Não	Não

Fonte: Autor (2025)

Avaliando os valores obtidos, é notável que sempre que houve a presença de um *code smell* todas as LLMs conseguiram classificá-lo, pontualmente houve discrepâncias, *ChatGPT* acabou classificando 2 programas sem *code smells* como *smells*. Abrindo o projeto no *App Inventor* do projeto 2 (Figura 10) observa-se que não só ele não tem um *Lazy Class* como também é uma versão mais avançada do programa original que recebeu a adição do *code smell* representado na Figura 7.

Figura 10. Visão de blocos do *App Inventor* para questão 2



Fonte: Autor (2025)

Por outro lado, o *ChatGPT* também respondeu para a questão número 10 a existência de *code smells*, mas avaliando a Figura 11 que representa a estrutura de blocos respectiva, não são destacados motivos para que tenha sido marcado como *Lazy Class*.

Figura 11. Visão de blocos do *App Inventor* para questão 10

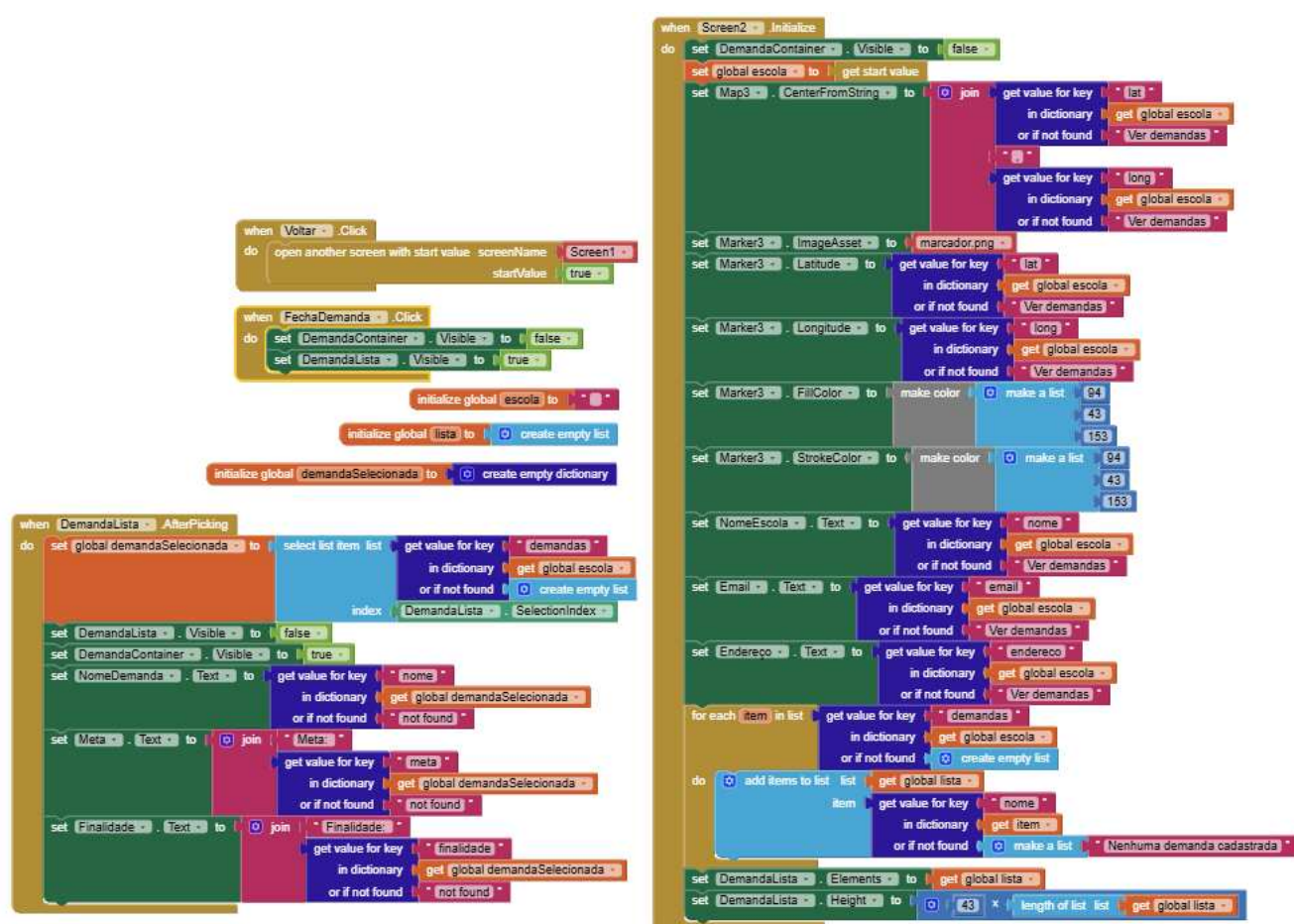


Fonte: Autor (2025)

Observando a diferença apontada pelo *Gemini*, na questão número 9 acabou apontando que havia *code smell* no código. Ao avaliar o projeto 9 no *App Inventor* notam-se vários blocos com muita lógica, fatores como 3 variáveis globais inicializadas podem ter influenciado o algoritmo a afirmar a existência de *Lazy Class*.

Por outro lado, no pequeno teste a LLM *DeepSeek-R1* não errou nenhuma predição sobre *code smells*, obviamente o experimento foi contido em um universo pequeno e apenas demonstrativo, mas o resultado conjunto dos algoritmos teve um resultado agradável, mostrando que, mesmo com parte da literatura contraindicado o uso de LLMs para identificação de *code smells*, ainda existe campo para o desenvolvimento.

Figura 12. Visão de blocos do *App Inventor* para questão 9



Fonte: Autor (2025)

Foram também realizados testes com versões locais de alguns modelos com *one-shot* seguindo a mesma metodologia de *prompt* utilizada no primeiro caso, os dados assim como o número de parâmetros de cada modelo são apresentados na Tabela 6.

Tabela 6. Respostas de LLMs locais com *One-Shot Prompt*

Número	Presença de Lazy Class	<i>Phi 4-mini 3.8B</i>	<i>Llama 3.1 8B</i>	<i>Gemma 3 4B</i>	<i>Deepseek-R1 7B</i>
1	Não	Não	Sim	Não	Não
2	Não	Não	Sim	Sim	Não
3	Sim	Sim	Sim	Sim	Não
4	Não	Sim	Sim	Não	Sim

5	Não	Não	Sim	Sim	Não
6	Sim	Não	Sim	Sim	Não
7	Não	Não	Sim	Sim	Não
8	Não	Não	Não	Não	Não
9	Não	Não ⁸	Sim	Não	Não ⁹
10	Não	Sim	Sim	Não	Não

Fonte: Autor (2025)

Neste trabalho são utilizados modelos locais, conforme detalhado na Seção 5.1. Dessa forma, é necessária uma avaliação mais pontual dos modelos testados em ambiente local, considerando que, devido às limitações de hardware disponíveis, o desempenho sem otimizações tende a apresentar maiores variações ou taxas de erro em comparação com grandes modelos de LLM avaliados na Tabela 5.

Phi 4 - Mini

Durante o *prompt* foi indicado que a resposta deve respeitar o formato “sim ou não”, mas o modelo desrespeitou esse pedido, adicionando explicações junto da resposta, fator que permitiu avaliar melhor a linha lógica durante os erros que apresentou nos testes.

Observando a questão 4, o modelo misturou conceitos do exemplo com a avaliação. O trecho inicial de sua resposta foi: “Sim, há um instância indubiada de Lazy Class neste trecho. A variável global 'musicas' é inicializada como uma lista com capacidade para seis itens; no entanto, nenhum item está realmente atribuído na declaração da variável.”. Referindo-se diretamente ao exemplo inserido na entrada e não sobre a questão que deveria ser analisada. O código avaliado tinha conteúdo de tamanho considerável, tal alucinação pode apontar para problemas como tamanho de janela de contexto nesses modelos menores.

Também analisando a questão 10, dos modelos remotos também errada pelo *ChatGPT*, o modelo apontou falta de implementação das telas “Borboleta3” e

⁸ A resposta apresentada foi: “Não, as classes do código fornecido não possuem o atributo “lazy”, que é tipicamente usado em Python para gerenciar recursos de forma otimista.” Acertando por conta de confusão com a pergunta.

⁹ A resposta apresentada foi: “Não, não há uma classe chamada Lazy Class no código fornecido.”, deixando clara a dificuldade dos modelos locais em trabalhar com entradas grandes.

“Borboleta5”, indicando uma previsão de conteúdo futuro ainda não criado. Contudo, essas ligações como apresentadas na Figura 11, refletem conteúdos de outras telas, sendo assim, o código de ambas telas de “Borboleta” não deve estar junto do *link* para elas, como se observa na estrutura do arquivo “.aia” (Figura 3), os códigos de bloco são divididos por telas. Esse erro, também causado pelo *ChatGPT* indica que arquivos de exemplo sem *smells*, podem ser úteis em uma RAG.

Llama 3.1

O modelo local com maior quantidade de parâmetros do teste, tendo 8 bilhões, foi também o modelo que apresentou pior desempenho, respondendo de forma recorrente a identificação de *Lazy Class* nos códigos apresentados. Por se tratar de um modelo mais robusto que os outros, esperava-se melhor resultado, contudo, ainda há espaço para melhorias utilizando uma melhor engenharia de *prompt*, que será apresentada no futuro.

Diante da quantidade de erros observados, é destacado apenas um caso. A Figura 13 apresenta a visão em blocos de um dos menores códigos avaliados, que foi incorretamente classificado como contendo o *code smell* analisado. É uma resposta curiosa, possivelmente devido à má interpretação do modelo, semelhante ao comportamento observado com o *Phi-4 Mini* na questão 10.

Figura 13. Visão de blocos do *App Inventor* para questão 9



Fonte: Autor (2025)

Gemma 3

Com apenas três diferenças dos dez exemplos usados, tem um desempenho semelhante ao do modelo *Phi*. O primeiro erro apresentado pelo modelo *Gemma 3*, assim como o *ChatGPT* nos testes com modelos *online*, foi na questão 2 (Figura 10), que se trata do código original do exemplo, antes das modificações realizadas na Figura 7, onde o *smell* foi inserido manualmente.

Para os outros 2 erros, ambas questões, apresentavam código referente a players de música, assim como no código de exemplo usado no *prompt*, fator que pode ter atrapalhado o modelo a inferir a parte correta do *smell*, portanto exemplos mais variados são necessários na RAG.

Deepseek - R1

Obtive um resultado não tão esperado, tendo em vista que seu modelo remoto respondeu corretamente todas as perguntas. O modelo local apresentou alguns problemas, não identificou nenhum dos *smells* inseridos, contudo como o modelo passa por um estágio de “pensamento” e transcreve frases humanas para explicar seu resultado pode-se analisar suas escolhas:

Para a questão 3, marcada como falsa observa-se:

- Observo que há uma seção com `global_declaration` chamada 'botoes' e dentro dela são criados componentes como `btndados`, `btnpaodequeijo` e `btmbiscoito`.
- Cada componente tem um `component_component_block` associado, que são os botões.
- Verifico que há três botões criar: `btmbiscoito`, `btnpaodequeijo` e `btndados`

Os comentários referem-se à Figura 9, onde foi criada uma lista, mas sem inserções de itens em todas as posições, o modelo segue a observação do exemplo, onde há apenas um elemento na lista, descartando o *smell* que deveria ter sido identificado por conta da lista subutilizada.

Analisando os resultados sobre a questão 4: “*There are two component_event blocks. Each block has a mutation component triggered on click (e.g., "Botão1" and "Botão2"). These mutations don't seem to have any attached logic; they're just event listeners without action*”.

É possível notar que assim como no *LLama*, o modelo infere não haver ligação do botão que troca de cena com alguma lógica, então vê-se deixar claro para os modelos em *prompts* futuros que as ligações estão sim acontecendo.

Por fim, a pergunta 6 é referente à Figura 8, onde há um procedimento que nunca é invocado, indicando uma preparação para atualizações futuras mas ainda sem implementação prática. Os “pensamentos” do modelo simplesmente não

identificaram problemas nesse caso, como ajuste, diferentes exemplos podem ser usados no RAG para aumentar a amplitude de identificação dos modelos.

5. IMPLEMENTAÇÃO COM USO DE RAG

Após a realização dos testes iniciais apresentados no Capítulo 4 foram desenvolvidos os experimentos deste capítulo o qual apresenta: o processo de criação e uso de RAG com modelos de LLM. São definidas restrições tecnológicas do projeto, *code smells* identificados pela aplicação, criação dos exemplos para RAG, técnica de *prompting* utilizada, implementação e avaliação dos resultados. Ao final apresenta-se outro experimento, objetivando o estudo de opções de trabalhos futuros. Na Seção 5.8 são levantados debates sobre ameaças à validade do projeto e indicações de trabalhos futuros.

5.1 DECISÕES TECNOLÓGICAS

No intuito de melhorar os resultados obtidos nos experimentos iniciais, no desenvolvimento do projeto optou-se pela utilização de modelos de LLM em conjunto ao uso de *Retrieval Augmented Generation (RAG)*. Abaixo são descritos os motivos das tecnologias adotadas.

- **Modelos**

. Dentre os critérios de escolha para os modelos, destacam-se:

- a) Limitações de hardware, modelos suportados em um computador com uma placa de vídeo GTX 1660 6GB, processador I5 9400F e 16GB de memória RAM. Isto é, priorização de modelos leves, com até 8 bilhões de parâmetros.
- b) Modelos com custo gratuito para viabilidade acadêmica e financeira.
- c) Modelos populares, com comunidade ativa em fóruns, assim como com boa documentação. Garantindo que eventuais problemas sejam resolvidos mais rapidamente.
- d) Optou-se pela execução de modelos locais em detrimento de hardware mais potente disponível por serviços nas nuvens, visto que serão usados dados de ambiente educacional, garantindo conformidade com a Lei Geral de Proteção de Dados (LGPD).
- e) Modelos compatíveis com RAG.

- f) Modelos compatíveis com *Ollama* disponíveis diretamente no site oficial¹⁰.

Respeitando as restrições definidas foram elencados 4 possíveis modelos para execução com RAG: *DeepSeek-R1*, *LLama 3.1*, *Gemma 3*, *Phi 4 mini*.

Os modelos foram testados em ambiente local com a execução do RAG preparado. Três modelos apresentaram desempenho aceitável e foram selecionados, o modelo *Phi 4 mini*, contudo, durante a execução gerava saídas totalmente não interpretáveis, como apresentado na Figura 14 levando a sua exclusão do estudo.

Figura 14. Exemplo de saída gerada pelo modelo *Phi 4 mini*

arquivo	resposta
n51_4833801691463680_ai_felipefontesmulina	1%H%#=-75.,.'C6/BCG<><DAH79486='..?=5)BF
n51_4833801691463680_ai_felipefontesmulina	G-.F6#A3."#253EI=B41G2,"?#,\$%",E-F"?7G)85B
n51_4833801691463680_ai_felipefontesmulina	=8A6E-'F,")-9A?)/>DA25.369H\$24:48-,%'6'/6=C/
n52_cee4d49d-c100-4bcf-8d5c-509dc1873ae2	!4D/%C-2G\$F'=41.C/1338.!&9A-/7/,\$3""<1#>&!)!
n52_cee4d49d-c100-4bcf-8d5c-509dc1873ae2	,9>:6."EB#&4+8=938>B%H,>9<%3.<AAF12!43):
n52_cee4d49d-c100-4bcf-8d5c-509dc1873ae2	HA&* & .AB;23-DFC4>G\$/C.C'GA*7E.&E4D="E
n53_5798095138586624_ai_mgalaraujo_jogu	,=#D9*%<EB78!=AB:+A>,42&GDE8E+5175B">A;
n53_5798095138586624_ai_mgalaraujo_jogu	5.7%E=6%?I76?2%--%81+>5\$/,,%,6&"\$!1B\$G7&
n53_5798095138586624_ai_mgalaraujo_jogu	3:B,12/F=.14+,AC>="*9:'7845!+D1DCE*G83?+,,H
n54_6686403691544576_ai_21702225_teste	D.-B18GE=#C6?C.%9C\$A\$5-.9B\$=1=F26+#!#=#
n54_6686403691544576_ai_21702225_teste	&F65*E7)++<-=/E?H8=H4"2?F*.8%!2!D3/.D'&6:<
n55_6397143584342016_ai_themarcos1996	'E2.5;.D"6'&5,<&,'G?&??E7.ID):>32-G3,3\$F.31:8

Fonte: Autor (2025)

● FAISS

Na etapa de recuperação é necessário um mecanismo capaz de recuperar arquivos baseado em similaridade. A biblioteca FAISS (*Facebook AI Similarity Search*), implementa as principais funcionalidades de um banco vetorial, isto é, uma estrutura que armazena representações numéricas (vetores) dos arquivos e realiza busca por similaridade entre eles, diferentemente de bancos clássicos que dependem da busca por valores exatos. Escolhida devido a popularidade e simplicidade de integração com outras bibliotecas.

¹⁰ <https://ollama.com/search>

- **All-MiniLM-L6-V2**

Na etapa de geração de *embeddings* (representações vetoriais semânticas), foi selecionado o modelo *all-MiniLM-L6-v2*, por se tratar de uma arquitetura leve e eficiente, capaz de processar textos de até 256 tokens sem impacto significativo no tempo de execução. O modelo é compatível com o FAISS, o que possibilita sua integração direta à etapa de indexação e recuperação vetorial adotada neste trabalho.

- **Lang Chain**

O framework *LangChain* foi adotado principalmente como interface de comunicação com os modelos de linguagem, por meio da classe `ChatOllama`. Trata-se de uma ferramenta amplamente utilizada na literatura devido a sua flexibilidade e pela capacidade de integração com diferentes tecnologias de inteligência artificial. Embora sua utilização tenha sido simplificada na execução do trabalho, possui diversas integrações com as principais tecnologias de IA.

- **Ollama**

O *Ollama* foi escolhido por permitir a execução local de modelos de linguagem de grande escala (LLMs) de forma simples e eficiente, sem dependência de serviços externos. Essa abordagem garante maior controle sobre a segurança dos dados e possibilita a reprodução do experimento em outros ambientes locais.

5.2 CODE SMELLS ESCOLHIDOS E DEFINIÇÕES

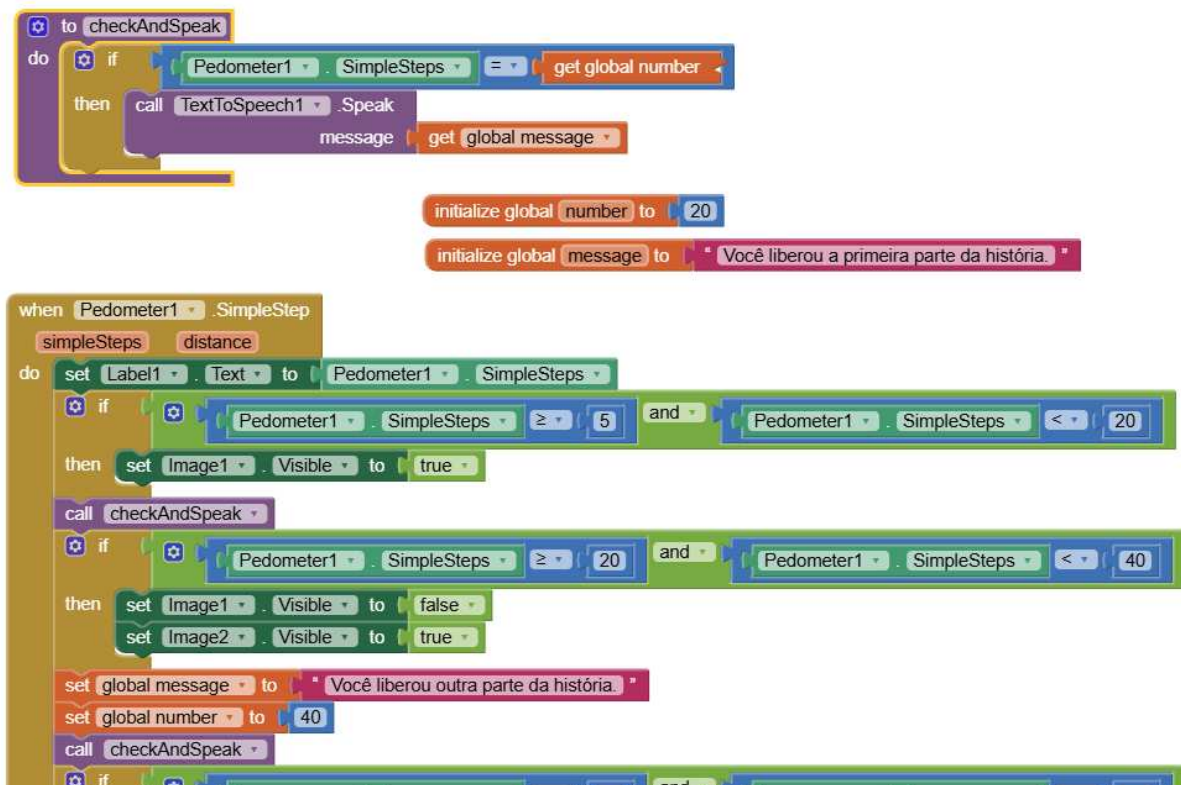
Para a realização deste trabalho, foram considerados apenas *code smells* descritos na literatura clássica (FOWLER; BECK, 2018) e entre os mais frequentemente identificados em ambientes de programação por blocos (Tabela 2). Também houve influência das revisões sistemáticas da literatura de CARAM et al. (2019) e AZEEM et al. (2019). Além disso, adotou-se a ideia de escolha por *smells* não facilmente identificados com análise estática de código.

Foram, então, selecionados três *code smells* para aplicação com o uso de RAG. As definições específicas para o *App Inventor* basearam-se tanto nos conceitos clássicos de Fowler (2018) quanto nas adaptações propostas por Hermans, Stolee e Hoepelman (2016), que definiu smells para as plataformas LEGO MINDSTORMS EV3 e Kodu. A seguir, são apresentados os *smells* escolhidos e suas respectivas definições.

5.2.1 Feature Envy

Observando a definição clássica: ocorre quando um método depende excessivamente de dados ou comportamentos de outra classe, demonstrando “inveja” de suas responsabilidades. No contexto do *App Inventor*, pode ser identificado quando blocos e eventos delegam quase completamente suas funcionalidades a outros componentes, sem incorporar lógica própria, como verificações, validações ou mensagens de retorno ao usuário. Assim, eventos como o clique de um botão acabam apenas repassando dados para outros blocos ou componentes. Como indicado no Quadro 1, o *App Inventor* permite a criação de procedimentos e funções, recurso que tende a favorecer a organização da lógica. No entanto, também pode-se identificar *Feature Envy* como o uso excessivo de variáveis globais em funções e procedimentos, o que deve ser evitado, pois aumenta o acoplamento, sendo muitas vezes recomendado a alteração para passagem de parâmetros, como pode ser visto na Figura 15.

Figura 15. Exemplo de *Feature Envy* em Aplicativo



Fonte: Autor (2025)

Como o *App Inventor*, isola diferentes telas, não é possível alterar dados utilizados exclusivamente em outros locais, portanto fica-se dependente do uso de variáveis para identificar grande parte dos casos de *Feature Envy*. No exemplo da Figura 15 há o uso de duas variáveis globais, *number* e *message*, atualizadas periodicamente pelo loop do pedômetro. No entanto, é necessário atualizá-las manualmente antes de cada chamada ao procedimento *checkAndSpeak*, sem grande funcionalidade para outros procedimentos, deveriam ser implementadas como variáveis locais pelo pedômetro e passadas como argumentos.

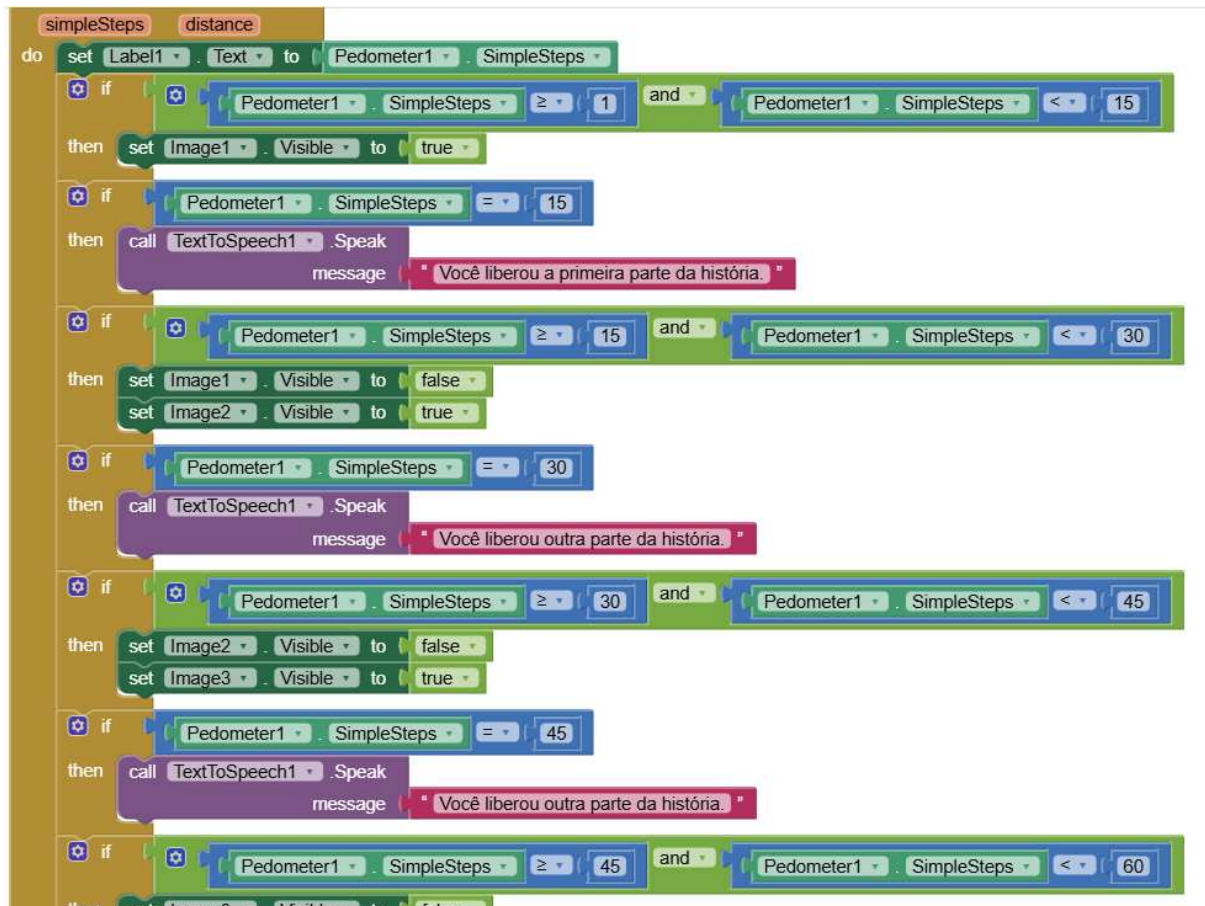
5.2.2 Large/God Class¹¹

God Class foi identificado como o *code smell* mais recorrente em AZEEM et al. (2019) e empatado na terceira posição em CARAM et al., 2019, mostrando-se relevante e necessário investigá-lo também em trabalhos voltados a linguagens de programação em blocos. Esse *smell* ocorre quando uma única classe concentra diversas responsabilidades, atuando em múltiplos contextos e rompendo com o

¹¹ Adotou-se o uso de *God Class* no projeto

princípio de separação de responsabilidades. No contexto do *App Inventor*, esse problema pode ser analisado de forma análoga à definição clássica: por exemplo, quando um único componente ou procedimento centraliza grande parte da lógica do aplicativo, tornando a manutenção mais complexa e aumentando o risco de inconsistências.

Figura 16. Exemplo de *God Class* em Aplicativo



Fonte: Autor (2025)

Avaliando a versão do aplicativo da Figura 16, observa-se que a função principal concentra diversas responsabilidades, controle da lógica de progressão, atualização da interface (imagens e *labels*) e execução de mensagens de áudio, tudo em um único bloco. Essa centralização excessiva caracteriza um caso de *God Class*, pois o código apresenta baixa coesão e forte acoplamento com diversos componentes, como *Pedometer1*, *TextToSpeech1*, *Label1* e múltiplas imagens. Além disso, há repetição de estruturas condicionais quase idênticas, o que aumenta a complexidade e o esforço necessário para compreender ou modificar o

comportamento do aplicativo. A lógica deveria estar distribuída em funções menores e mais especializadas, separando a verificação de progresso, atualização visual e a geração de mensagens.

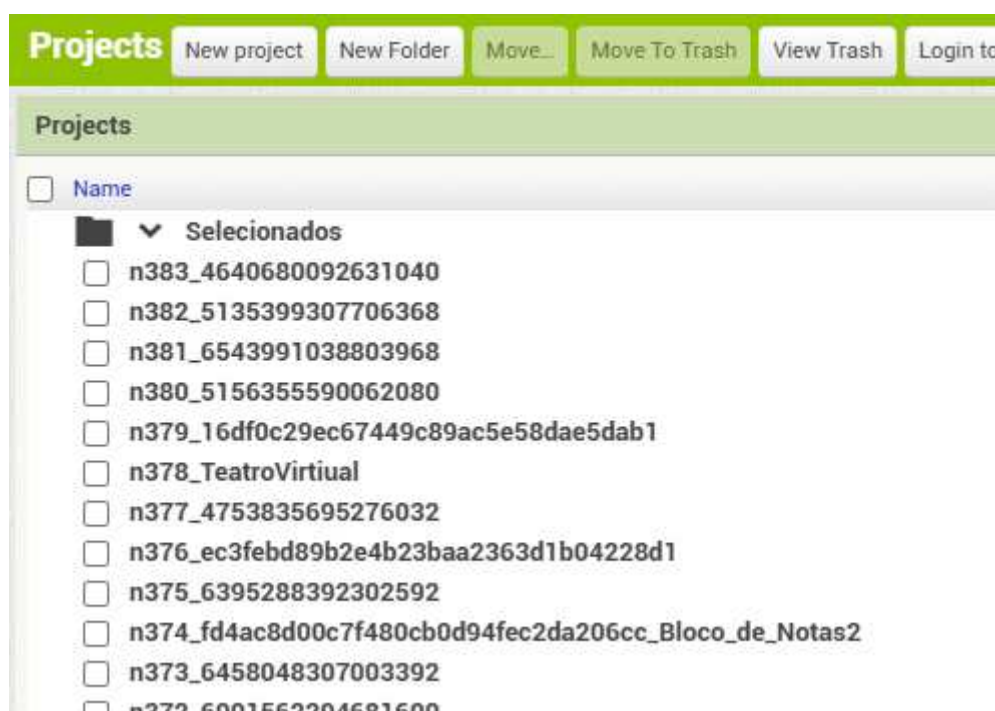
5.2.3 Lazy Class

Como ressaltado na Seção 5.2, *Lazy Class* refere-se a classes que apresentam pouca ou nenhuma responsabilidade, não exercendo função prática no sistema. No contexto do App Inventor, esse smell pode ser visto de diversas formas: procedimentos criados mas nunca invocados, variáveis globais declaradas e não utilizadas, ou componentes da interface inseridos sem qualquer vínculo com a lógica do aplicativo (caso não possível de avaliar observando apenas arquivos “.bky”). A identificação desse *smell* em linguagens de blocos exige uma interpretação fortemente subjetiva, uma vez que programação em blocos tem uma estrutura diferente das linguagens tradicionais, forçando em definição grande similaridade com *Dead Code*. Nesse caso, torna-se necessário avaliar se um procedimento, variável ou componente contribui para a funcionalidade da aplicação. Caso não apresente uso prático, pode ser caracterizado como uma *Lazy Class*, como observado nas Figuras 7, 8 e 9.

5.3 PREPARAÇÃO DO DATASET PARA RAG

Conforme já apresentado na Seção 4.2, os códigos-fonte de 383 aplicativos desenvolvidos com *App Inventor* foram selecionados, extraídos e tratados. Assim, a última etapa necessária foi realizar a classificação desses arquivos, de forma a identificar a presença ou ausência daqueles *smells* escolhidos. Esse processo permitiu a construção de uma base de dados rotulada, essencial para os experimentos de detecção automática e análise. Para isso, os arquivos foram subidos um a um no *App Inventor* (Figura 17), para serem então analisados manualmente, julgando haver ou não presença dos três *smells* escolhidos na abordagem do projeto. Esse processo resultou em 956 arquivos “.bky” rotulados (pois cada aplicativo desenvolvido pode ter mais de uma tela e, portanto, mais de um arquivo “.bky”).

Figura 17. Arquivos selecionados no *App Inventor* para Análise



Fonte: Autor (2025)

Após os dados para análise estarem preparados, foi criada uma pequena base de dados para o modelo de *embeddings* (*all-MiniLM-L6-v2*), responsável por fazer as comparações entre entrada e dados armazenados, recuperando os mais similares para alimentar a entrada do modelo. Para isso, foram selecionados do *dataset* original 22 arquivos.

Foram escolhidos 6 arquivos para cada *smell* abordado (*Lazy Class*, *God Class*, *Feature Envy*), que previamente continham *smells* ou que foram inseridos manualmente. Também foram inseridos 6 exemplos que não apresentavam nenhum dos três *smells*. Esses arquivos passaram pela mesma extração e limpeza que o conjunto de 383 já rotulados, permitindo assim que os modelos façam comparações com arquivos de mesma natureza.

Para que o banco vetorial possa recuperar arquivos semelhantes, é necessário gerar *embeddings*¹² a partir de cada arquivo. Esses vetores de representação são utilizados para comparar similaridades com o *embedding* do arquivo alvo da avaliação. Além disso, como forma de enriquecer o *prompt* para o RAG, foram criados também metadados associados a cada um dos arquivos armazenados no banco vetorial, esses metadados contêm, além do nome do

¹² Representações vetoriais dos códigos dos arquivos

arquivo, seu conteúdo original, a saída esperada caso fosse utilizado como entrada, e uma breve explicação justificando tal resultado, como pode ser observado no Quadro 11.

Quadro 11. Rotulação com metadados de um arquivo

```
[ ...  
  
  {  
    'arquivo': 'grande_14_com_feature_envy_tela_endscreen.bky',  
    'explicacao': 'This block exhibits a Feature Envy smell because a single  
                  button is responsible for both computational logic and  
                  interface updates. It reads user inputs, calculates the  
                  BMI, and updates the label on the interface, thus  
                  concentrating multiple responsibilities within a single,  
                  focused area of the program.',  
    'saida_esperada': 'Yes: [Feature Envy]',  
    'texto': "<?xml version='1.0' encoding='utf-8'?>\n"  
             '<xml xmlns="https://developers.google.com/blockly/xml" '  
             'xmlns:html="http://www.w3.org/1999/xhtml"><block '  
             'type="component_event" x="-1849" y="-374">  
             ...  
             </block>  
             </xml>' }  
    ...  
  ]
```

Fonte: Autor (2025)

Tanto o banco vetorial quanto os metadados são criados a partir do código apresentado no Quadro 12, que mostra a parte principal do processo de criação do banco vetorial FAISS assim como dos metadados associados.

Quadro 12. Código de Criação do Banco Vetorial

```
# Modelo de embeddings  
modelo = SentenceTransformer("all-MiniLM-L6-v2")  
  
vetores = []  
metadados = []  
  
for arquivo in os.listdir(CAMINHO_BKY):  
    if arquivo.endswith(".bky"):  
        caminho = os.path.join(CAMINHO_BKY, arquivo)  
        with open(caminho, "r", encoding="utf-8", errors="ignore") as f:  
            texto = f.read()  
  
        # Gera um único embedding para o texto completo do arquivo  
        emb = modelo.encode([texto])[0]  
  
        # Obtém a explicação correspondente  
        arquivo_sem_ext = os.path.splitext(arquivo)[0]
```

```
explicacao = EXPLICACOES_METADADOS.get(arquivo_sem_ext)
saida_esperada = SAIDA_ESPERADA_METADADOS.get(arquivo_sem_ext)

vetores.append(emb)
metadados.append({
    "arquivo": arquivo,
    "texto": texto,
    "explicacao": explicacao,
    "saida_esperada": saida_esperada
})

# Converter para formato FAISS
vetores_np = np.array(vetores).astype("float32")
index = faiss.IndexFlatL2(vetores_np.shape[1])
index.add(vetores_np)

# Salvar base
faiss.write_index(index, CAMINHO_FAISS)
with open(CAMINHO_META, "wb") as f:
    pickle.dump(metadados, f)
```

Fonte: Autor (2025)

Como observado no Quadro 12, houve a escolha de gerar apenas um *embedding* por arquivo. A decisão de não dividir o arquivo “.bky” em *chunks*, mesmo sendo uma prática muito comum e adotada em diversos usos de RAG, é justificada por conta da peculiaridade deste caso. A fragmentação poderia fazer com que a comparação entre os *chunks* do arquivo alvo recuperasse trechos não representativos dos arquivos de exemplo, condicionando o modelo a tomar decisões enviesadas com base em parâmetros que não apresentam o que é descrito.

5.4 APERFEIÇOAMENTO DO *PROMPT*

O primeiro *prompt* usado no exemplo de *one-shot* apresentado na Seção 4.3 foi uma versão inicial. O objetivo do modelo com RAG é identificar a ocorrência de zero, um ou mais *smells* em uma mesma execução do modelo, para isso o *prompt* foi melhor elaborado. Para aprimorar o desempenho, no *prompt* final foram utilizadas técnicas de *roleplaying* propostas por Xu et al. (2025), seguindo seu *template* e tentando manter boas práticas de *prompting* tanto quanto possível, adicionando contexto e exemplos, utilizando frases curtas tentando ser claro e direto.

Observando o Quadro 13, as *tags* “SYS” são uma forma ‘geral’ de indicar ao modelo que aquela entrada é parte do contexto geral dentro do qual ele deve

responder a pergunta, dentro do bloco foi inserida a “especialidade” do modelo, seus conhecimentos e definições dos *smells*, apresentadas no Quadro 14, procurados adaptados para *App Inventor*.

Quadro 13. *Prompt* final - parte 1

```
PROMPT = f"""

[INST] <<SYS>>

You are an expert in App Inventor and software quality.

You are familiar with Kent Beck's concept of code smells and Martin
Fowler's definitions from his book "Refactoring: Improving the Design of
Existing Code."

Code smells are signs of structural issues in a program that, while not
preventing it from running, can harm its maintainability and long-term
evolution.

You know that App Inventor uses a visual, block-based programming
environment called Blockly.

You understand that each program is represented internally in XML, where
blocks define events, procedures, and logic connections.

You also know that blocks such as simple button navigation between screens,
without internal logic or computation, are not considered code smells by
definition.

You classify code smells as follows:

Feature Envy:

{FEATURE_ENVY_DEFINITION}

God Class:

{GOD_CLASS_DEFINITION}

Lazy Class:

{LAZY_CLASS_DEFINITION}

<</SYS>>
```

Fonte: Autor (2025)

Quadro 14. Definições de *smells* para o *Prompt*

```
FEATURE_ENVY_DEFINITION = (
    "Feature Envy occurs when a block or procedure depends excessively on
    data or behavior "
```

```

    "from other blocks or global variables instead of encapsulating logic
    within itself. "
    "Typical signs include event blocks that only transfer data or invoke
    other components "
    "without performing meaningful local processing."
)

GOD_CLASS_DEFINITION = (
    "God Class occurs when one block or procedure accumulates excessive
    logic and control, "
    "performing multiple unrelated tasks by accessing blocks of different
    natures, "
    "instead of focusing on a single responsibility."
)

LAZY_CLASS_DEFINITION = (
    "A Lazy Class refers to blocks or components that contribute little or
    nothing to the system's functionality. "
    "This may include unused procedures, unreferenced global variables, or
    interface elements not linked to any event or logic."
)

```

Fonte: Autor (2025)

O Quadro 15 apresenta a real questão para o modelo, ela contém os exemplos recuperados com base no arquivo, suas explicações e saídas esperadas, também regras para saída¹³ esperada após a análise do LLM.

Quadro 15. *Prompt* final - parte 2

```

Context with similar examples:

{{context}}

Code to analyze:

{{code_to_analyze}}

Answer rules:

If you clearly identify one or more code smells, answer:

"Yes: [list the detected smells]"

If no smell is detected, answer:

```

¹³ Muitas vezes desrespeitadas pelo modelo, mesmo durante a execução final, exigindo muitas vezes que a avaliação da resposta seja manual ou que o modelo seja perguntado novamente até gerar uma resposta no padrão.

```
"No"
```

```
If the context is uncertain, answer:
```

```
"Doubt"
```

```
Example of expected answer format:
```

```
Yes: [God Class, Feature Envy]
```

```
No
```

```
Doubt
```

```
IMPORTANT: You MUST only answer exactly as in the examples above. Do NOT  
explain, do NOT add anything else.
```

```
Answer:
```

```
[/INST]
```

```
""
```

Fonte: Autor (2025)

5.5 IMPLEMENTAÇÃO DO RAG

O algoritmo é simples e utiliza as mesmas tecnologias utilizadas para criar a base FAISS, inicialmente é preciso instanciar o modelo *all-MiniLM-L6-v2* (Quadro 16), responsável por gerar *embeddings* das entradas que serão utilizados internamente para a comparação com os dados do banco, recuperando em conjunto os metadados associados àquele arquivo, como observado no Quadro 17. Também é realizada a instanciação do modelo antes de cada pergunta, garantindo o isolamento entre as questões.

Quadro 16. Inicialização do mecanismo RAG

```
class ChatBKY:
    def __init__(self):
        self.prompt = PromptTemplate.from_template(PROMPT)

        # FAISS + Metadados
        print("> Loading FAISS index and metadata ...")
        self.index = faiss.read_index(CAMINHO_FAISS)
        with open(CAMINHO_META, "rb") as f:
            self.metadados = pickle.load(f)
```

```
print("> FAISS loaded with", len(self.metadados), "files")

# Modelo de embeddings
self.embed_model = SentenceTransformer("all-MiniLM-L6-v2")
```

Fonte: Autor (2025)

Quadro 17. Busca de arquivos e geração da resposta no RAG

```
def ask_file(self, bky_path, top_k=3):
    """Recebe um arquivo .bky, busca contextos relevantes no FAISS e gera a
    resposta."""
    if not os.path.isfile(bky_path):
        return f"Error: file {bky_path} not found."
    # Ler arquivo completo
    with open(bky_path, "r", encoding="utf-8", errors="ignore") as f:
        texto = f.read()
    # Recuperar arquivos similares da base
    retrieved_texts = self.retrieve_similar_files(texto, top_k=top_k)
    # Montar contexto e prompt
    # Iterando de trás para frente para que os mais relevantes estejam na
    parte de baixo
    # Dando maior probabilidade de estarem no contexto do LLM caso haja
    corte
    context_text = "\n---\n".join(reversed(retrieved_texts))
    prompt_filled = self.prompt.format_prompt(
        context=context_text,
        code_to_analyze=texto
    ).to_string()
    # Gerar resposta com LLM
    parser = StrOutputParser() # Não é um modelo, apenas um utilitário para
    extrair string
    # LLM setada isoladamente para cada pergunta
    model = ChatOllama(
        model="deepseek-r1:7b",
        num_predict=100,
        n_gpu_layers=-1,
        num_gpu=1,
        reasoning=False,
        temperature=0.1,
    )
    raw_output = model.invoke(prompt_filled)
    return parser.invoke(raw_output)
```

Fonte: Autor (2025)

O Quadro 17 apresenta as configurações adotadas para a execução do modelo local. O parâmetro `num_predict` limita a geração a até 100 *tokens* de resposta, um valor escolhido para reduzir o risco de alucinações e respostas excessivamente longas. Além disso, a *flag* `reasoning` foi desativada, pois o processo de raciocínio explícito de modelos como o *DeepSeek-R1* consome uma quantidade significativa de tokens da janela de contexto, o que comprometeria o espaço disponível para o conteúdo relevante. Também foi alterado `temperature`,

esse parâmetro indica quão criativo o modelo pode ser, respeitando o intervalo de 0 até 1, por padrão seu valor é 0.8 o que dá maior liberdade criativa ao modelo, como o objetivo é classificar um código, o valor foi diminuído para 0.1.

A janela de contexto, isto é, o número máximo de tokens que o modelo pode processar entre entrada e saída, é limitada a 2048 tokens por padrão. Devido às restrições de hardware, neste caso, uma placa de vídeo GTX 1660 com 6 GB de VRAM, aumentar esse limite mostrou-se inviável em termos de desempenho e tempo de inferência. Assim, embora modelos como o *DeepSeek-R1* e o *Gemma 3* suportem janelas de até 131072 tokens, não foi possível explorar de forma satisfatória essa capacidade. É muito provável que, com uma janela maior, as respostas geradas apresentassem melhor resultado.

5.5 ANÁLISE DOS RESULTADOS

Nesta seção são apresentadas análises sobre as estatísticas geradas pela execução dos modelos. Primeiramente são apresentadas métricas clássicas para a avaliação do modelo. Por final é feito o cálculo de índice de concordância de Cohen's Kappa.

5.5.1 *Precision, Recall, F1-Score, Accuracy*

A partir dos 383 aplicativos, da base de dados vetoriais geradas e do RAG, os modelos: *Llama 3.1*, *Gemma 3* e *DeepSeek-R1* foram executados para classificar os *smells*. Por se tratar de modelos de LLM, algumas respostas obtiveram resultados não interpretáveis. Para amenizar o número de arquivos não classificados, aconteceu uma segunda execução, com apenas as entradas sem resposta para cada modelo. Ao final da execução, aqueles novos resultados que se apresentaram dentro dos padrões foram inseridos junto aos dados da primeira execução.

Após a execução do modelo, foram criadas tabelas para cada *smell* de forma manual¹⁴, onde colunas simbolizam: nome da aplicação, *Lazy Class*, *Feature Envy*, *God Class* e *None* respectivamente. Conforme a interpretação da saída gerada pelos modelos, cada linha foi preenchida com um 'x'.

¹⁴ Para o modelo Gemma foi possível fazer a categorização com auxílio de um *script*

Alguns arquivos não foram classificados em nenhuma categoria pois a resposta das LLMs não indicou nenhuma classe a qual o código pertenceria. Os números de arquivos mal classificados foram de: 166 para o *DeepSeek*, 14 para o *Gemma* e 40 para o *LLama*. A Tabela 7 mostra a contagem de identificações para cada modelo.

Tabela 7. Comparação de detecção de *smells* por abordagem¹⁵

Abordagem	<i>Lazy Class</i>	<i>Feature Envy</i>	<i>God Class</i>	<i>Nenhum</i>	<i>Não Classificados</i>
Manual	111	106	54	739	0
<i>DeepSeek-R1</i>	63	113	48	572	166
<i>Gemma 3</i>	60	621	535	75	14
<i>Llama 3.1</i>	436	112	412	56	40

Fonte: Autor (2025)

Observa-se na Tabela 7, classificações díspares, principalmente quanto aos modelos *Gemma 3* e *Llama 3.1*. Por outro lado, o modelo *DeepSeek* classificou um número de arquivos mais próximo do número de *smells* encontrados na base rotulada. Uma análise com as medidas selecionadas será apresentada a seguir.

Baseado na revisão sistemática da literatura foram selecionadas para a avaliação dos resultados as seguintes métricas: *Accuracy*, *Recall*, *Precision* e *F1-Score*. Essas métricas permitem quantificar de forma objetiva o desempenho do sistema na detecção de *code smells*, medindo tanto sua capacidade de identificar corretamente os casos positivos quanto de evitar falsos positivos e negativos.

- **Accuracy** $\frac{TP + TN}{TP + TN + FP + FN}$

Definida como a razão entre o número de previsões corretas (tanto verdadeiros positivos quanto verdadeiros negativos) e o total de previsões realizadas. Indica a acurácia total do modelo.

- **Recall:** $\frac{TP}{TP + FN}$

¹⁵ Números não somam para um mesmo total pois arquivos podem ser classificados com mais de um *code smell*

Definido como a razão entre o número de previsões verdadeiras positivas e a soma das previsões verdadeiras positivas com as previsões falsas negativas. Indica se o modelo comete poucos falsos negativos

- **Precision:** $\frac{TP}{TP + FP}$

Definido como a razão entre o número de previsões verdadeiras positivas e a soma das previsões verdadeiras positivas com as previsões falsas positivas. Indica se o modelo comete poucos falsos positivos.

- **F1-Score:** $2 \times \frac{(Precision \times Recall)}{(Precision + Recall)}$

Definido como a média harmônica entre *precision* e *recall*. Buscando equilíbrio entre as duas médias. Média harmônica em *F1-Score* é dada por duas vezes a divisão entre: a multiplicação de *precision* por *recall* e sua soma.

As Tabelas 8, 9 e 10 mostram o desempenho dos modelos para cada métrica utilizada, a seguir são apresentadas análises sobre cada uma das tabelas.

Tabela 8. Métricas da Execução - *DeepSeek R1*

<i>Code Smell</i>	<i>Precision</i>	<i>Recall</i>	<i>F1-Score</i>	<i>Accuracy</i>
<i>Lazy Class</i>	0,111	0,063	0,080	0,833
<i>Feature Envy</i>	0,150	0,160	0,155	0,806
<i>God Class</i>	0	0	0	0,893
Nenhum	0,872	0,675	0,761	0,673

Fonte: Autor (2025)

Observando a Tabela 8, o modelo *DeepSeek R1*, com 7 bilhões de parâmetros, apresentou desempenho insatisfatório na execução local. Observou-se especialmente baixo desempenho na detecção do *smell God Class*, cujo *recall* e *precision* foram ambos nulos, simbolizando que o modelo não reconheceu corretamente nenhum exemplo dessa categoria.

Para *Lazy Class* e *Feature Envy*, todos os valores são baixos, evidenciando que o modelo não conseguiu generalizar padrões desses *smells*. Apenas os valores associados a “Nenhum *code smell*” apresentam métricas razoáveis (F1-Score de 0,761), indicando que o modelo prevê ausência de *smells* com alguma consistência. Esse resultado, contudo, pode ser explicado parcialmente pela natureza do *App Inventor*, que se trata de um aplicativo introdutório de programação. No conjunto de dados rotulados, há a presença de diversos projetos simples, sem blocos ou com lógica mínima, como apenas alterar uma tela no click de um botão, o que tende a inflar dados categorizados como “Nenhum”

Tabela 9. Métricas da Execução - *Gemma 3*

<i>Code Smell</i>	<i>Precision</i>	<i>Recall</i>	<i>F1-Score</i>	<i>Accuracy</i>
<i>Lazy Class</i>	0,083	0,045	0,058	0,832
<i>Feature Envy</i>	0,132	0,774	0,226	0,411
<i>God Class</i>	0,077	0,759	0,139	0,470
Nenhum	0,800	0,081	0,147	0,274

Fonte: Autor (2025)

Analisando a Tabela 9, observa-se que o modelo *Gemma 3*, com 4 bilhões de parâmetros, apresentou um desempenho também insatisfatório. Em contraponto ao resultado do modelo *DeepSeek*, o *Gemma* obteve alguns valores consideráveis, como o *Recall* para *Feature Envy* (0,774) e para *God Class* (0,759). Contudo, observando os valores baixos de *F1-Score* para ambos *smells* (0,226 e 0,139 respectivamente), nota-se um desequilíbrio entre o valor de *Recall* e *Precision*.

O desequilíbrio indicado pelo valor baixo de *F1-Score* indica que o modelo superestimou a presença de *smells*, gerando um número alto de falsos positivos, fato corroborado pela Tabela 7, que mostra 621 identificações de *Feature Envy* e 535 de *God Class*, valores muito elevados quando comparados aos valores reais observados na base rotulada. Assim, o modelo estava propenso a classificar a identificação desses dois *smells* em detrimento a detecção de *Lazy Class* ou da categoria “Nenhum”.

A alta *precision* de “Nenhum” (0,800) ocorreu por conta de sua baixa predição. Contudo, o baixo *recall* dessa categoria (0,081) evidencia que o *Gemma*

falhou em reconhecer a não existência de *smells*, pois sendo a categoria mais abundante no conjunto com 739 ocorrências, foram identificadas somente 75 casos (Tabela 7), mantendo um desempenho ruim mesmo sob hipótese de que todas as predições estivessem corretas.

Tabela 10. Métricas da Execução - *Llama 3.1*

<i>Code Smell</i>	<i>Precision</i>	<i>Recall</i>	<i>F1-Score</i>	<i>Accuracy</i>
<i>Lazy Class</i>	0,061	0,189	0,092	0,566
<i>Feature Envy</i>	0,125	0,132	0,128	0,801
<i>God Class</i>	0,117	0,889	0,206	0,613
Nenhum	0,714	0,054	0,101	0,252

Fonte: Autor (2025)

Observando a Tabela 10, o modelo *Llama 3.1*, com 8 bilhões de parâmetros, apresenta um padrão semelhante ao identificado nos modelos anteriores. O *code smell God Class* apresenta um valor alto de *recall* (0,889), evidenciando poucos falso negativos, assim como o *Gemma 3*.

Uma diferença notável em relação às demais execuções está na categoria *Lazy Class*. Observa-se que o modelo classificou 436 arquivos (Tabela 7) como pertencentes a essa categoria. Contudo, apresenta um valor de *recall* baixo, demonstrando a presença significativa de falsos negativos, ou seja, *smells* classificados como “*Lazy Class*” na base rotulada que não foram identificados pelo modelo. Esse resultado sugere que, embora o *Llama* tenha inferido frequentemente a presença de *Lazy Class*, grande parte das predições estavam incorretas, refletindo uma baixa capacidade de generalização para esse *smell*.

5.5.2 Cohen’s Kappa

Além das métricas clássicas, devido a subjetividade da classificação de *code smells* até mesmo por especialistas com critérios semelhantes, foi empregado o cálculo do índice de concordância de Cohen’s Kappa, uma métrica estatística amplamente utilizada em estudos que envolvem julgamento humano. Assim, o uso combinado dessas medidas fornece uma avaliação mais equilibrada e confiável da

consistência e da precisão das respostas geradas pelo modelo. O índice de Cohen's Kappa pode ser interpretado pela definição de Landis e Koch (1977), pode ser observado na Tabela 11 e aplicado na Tabela 12.

Tabela 11. Interpretação de Cohen's Kappa por Landis e Koch - 1977

Kappa	Interpretação
< 0,00	Concordância pior que o acaso
0,00 - 0,20	Fraca
0,21 - 0,40	Razoável
0,41 - 0,60	Moderada
0,61 - 0,80	Substancial
0,81 - 1,00	Quase perfeita

Fonte: Autor (2025)

Tabela 12. Cohen's Kappa - Todos Modelos

Modelo	Smell	Kappa	Interpretação
<i>DeepSeek-R1</i>	<i>Lazy Class</i>	-0,004	Concordância pior que o acaso
	<i>Feature Envy</i>	0,046	Fraca
	<i>God Class</i>	-0,056	Concordância pior que o acaso
	Nenhum	0,266	Razoável
<i>Gemma 3</i>	<i>Lazy Class</i>	-0,025	Concordância pior que o acaso
	<i>Feature Envy</i>	0,045	Fraca
	<i>God Class</i>	0,041	Fraca
	Nenhum	0,006	Fraca
<i>Llama 3.1</i>	<i>Lazy Class</i>	-0,102	Concordância pior que o acaso
	<i>Feature Envy</i>	0,016	Fraca
	<i>God Class</i>	0,118	Fraca

Nenhum

-0,009 Concordância pior que o acaso

Fonte: Autor (2025)

Analisando a Tabela 12, observa-se o esperado a partir das análises apresentadas na Subseção 5.5.1: a concordância entre os modelos e os dados rotulados foi considerada muito ruim. Embora a detecção de *code smells* envolva certo grau de subjetividade, o índice de Cohen's Kappa indica uma relação tão fraca entre as respostas que essa subjetividade, por si só, não é suficiente para justificar os resultados obtidos.

5.6 AMEAÇAS À VALIDADE DOS RESULTADOS

Na etapa de geração das respostas, foram observadas diversas imprecisões, variando desde erros em análises simples até respostas totalmente desconexas das regras de saída estabelecidas no Quadro 14. Possivelmente sugerindo uma limitação na janela de contexto utilizada, indicando que o modelo pode ter recebido muita informação para realizar inferências no processo de RAG. Sendo assim, há respostas que podem não condizer com as capacidades reais do modelo.

Uma das estratégias adotadas para mitigar a pequena janela de contexto foi restringir o número de documentos recuperados do banco FAISS a três, limitando o impacto do RAG. Entretanto, devido às alucinações observadas nas saídas, é provável que o limite de 2048 tokens tenha sido ultrapassado em diversas consultas..

A utilização de hardware com maior capacidade de processamento e VRAM poderia impactar tanto a qualidade quanto a velocidade das respostas, permitindo que modelos mais complexos, como o *DeepSeek-R1*, mantenham seu módulo de raciocínio ativado. Consiste em uma ameaça visto que o ponto diferencial de modelos com “reasoning” foi desativado, potencialmente deteriorando o desempenho do modelo.

Outro possível problema por conta de janela de contexto ocorre sobre o modelo de *embedding* (neste trabalho o all-MiniLM-L6-v2), que se trata de um modelo pequeno (256 tokens), o que pode resultar em truncamento da entrada, ou

seja, dos exemplos para o RAG e do dado de entrada, fazendo com que a recuperação não abranja todo o conteúdo. Houve a tentativa de mitigar parte desse problema ao recuperar o código completo dos metadados, inserindo ele novamente no *prompt* da LLM.

No pré-processamento das tags dos arquivos “.bky”, utilizou-se a biblioteca `xml` do *Python* para realizar a limpeza de atributos “id” e da tag `yacodeblocks`. Contudo, essa biblioteca modifica a estrutura interna do XML, por exemplo, removendo *tags* `mutation` e adicionando-as à *tag* externa, alterando assim a estrutura natural do documento. Essas mudanças podem impactar a representação do código e, conseqüentemente, afetar a capacidade do modelo de gerar inferências corretas.

Na etapa de preparação dos arquivos para o banco vetorial do RAG, não foram incluídos arquivos com sintaxe quebrada, que, na etapa de rotulação, representavam uma quantidade considerável. Na maioria dos casos, esses arquivos foram classificados como *Lazy Class*. Blocos com sintaxe incorreta ou esquecidos no meio do código podem não ser interpretados pelo modelo com o mesmo peso da categoria *Lazy* durante a inferência, comprometendo a fidelidade da avaliação e constituindo uma ameaça à validade interna dos resultados.

O conjunto de dados utilizado para a seleção dos aplicativos deste projeto corresponde, na realidade, a um subconjunto de 88.863 aplicações, contendo 2.822 aplicativos, majoritariamente em português, com algumas ocorrências em inglês e espanhol. Trata-se de uma amostragem realizada previamente, que serviu como base para este trabalho. Essa restrição pode limitar a generalização dos resultados.

5.7 SEGUNDO EXPERIMENTO COM OTIMIZAÇÕES DE TOKENS

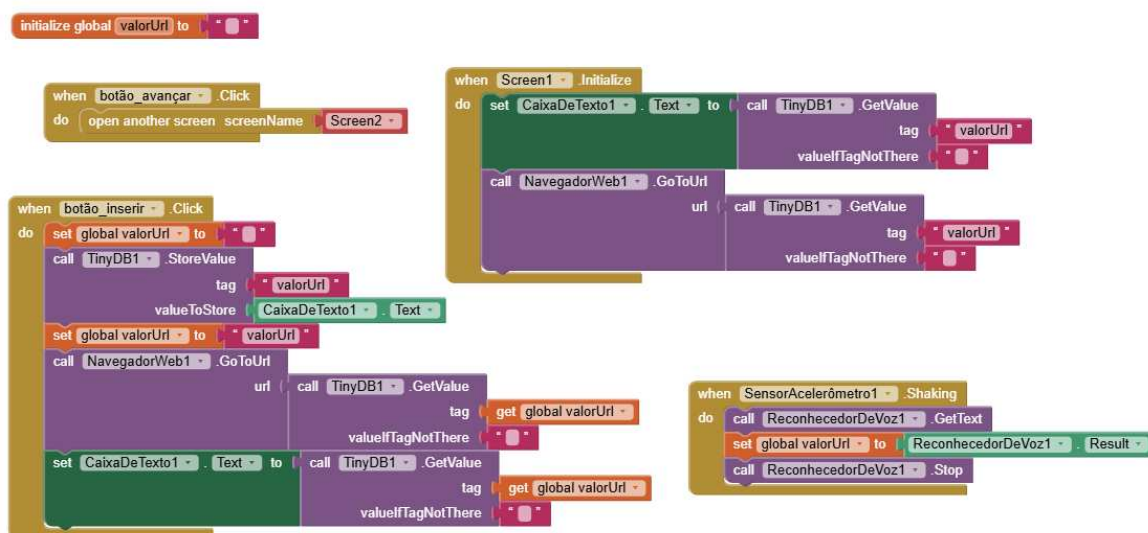
Nesta seção são apresentados os resultados obtidos no terceiro experimento realizado neste trabalho, o segundo com RAG em modelos locais. Primeiro são destacadas as alterações em relação ao primeiro experimento com RAG (apresentado nas Seções 5.3 e 5.4), seguido de uma rápida análise dos resultados.

5.7.1 Alterações realizadas no segundo experimento

Os resultados insatisfatórios da Seção 5.5 estimularam o desenvolvimento de um segundo ciclo de experimentação. Buscou-se otimizar o RAG no quesito de número de *tokens* da janela de contexto, nos modelos locais limitado em 2048 *tokens*.

Utilizou-se a técnica de *chunking* com uma adaptação para códigos de *App Inventor*. Diferentemente da técnica convencional, onde a separação é realizada em intervalos iguais, neste trabalho o *chunking* foi adaptado para ser realizado em cada “bloco raiz”, ou seja, os blocos mais externos aos outros blocos no *App Inventor*. A Figura 18 mostra 5 blocos base na interface gráfica do *App Inventor*.

Figura 18. Cinco “Blocos Raiz” em *App Inventor*



Fonte: Autor (2025)

Conforme apresentado na Figura 18, tanto o banco vetorial quanto a entrada foram separados em *chunks* que dependem da estrutura do código analisado. Como comentado na Seção 5.3, com a realização de *chunking* há perda de contexto para a análise do modelo.

Buscando diminuir o impacto negativo da separação dos códigos, antes de dividir o arquivo em blocos é realizada uma análise que encontra blocos de variáveis globais, funções e procedimentos¹⁶, então outra varredura é feita contando quantas vezes esses blocos são mencionados no código. Caso não aconteça menção à

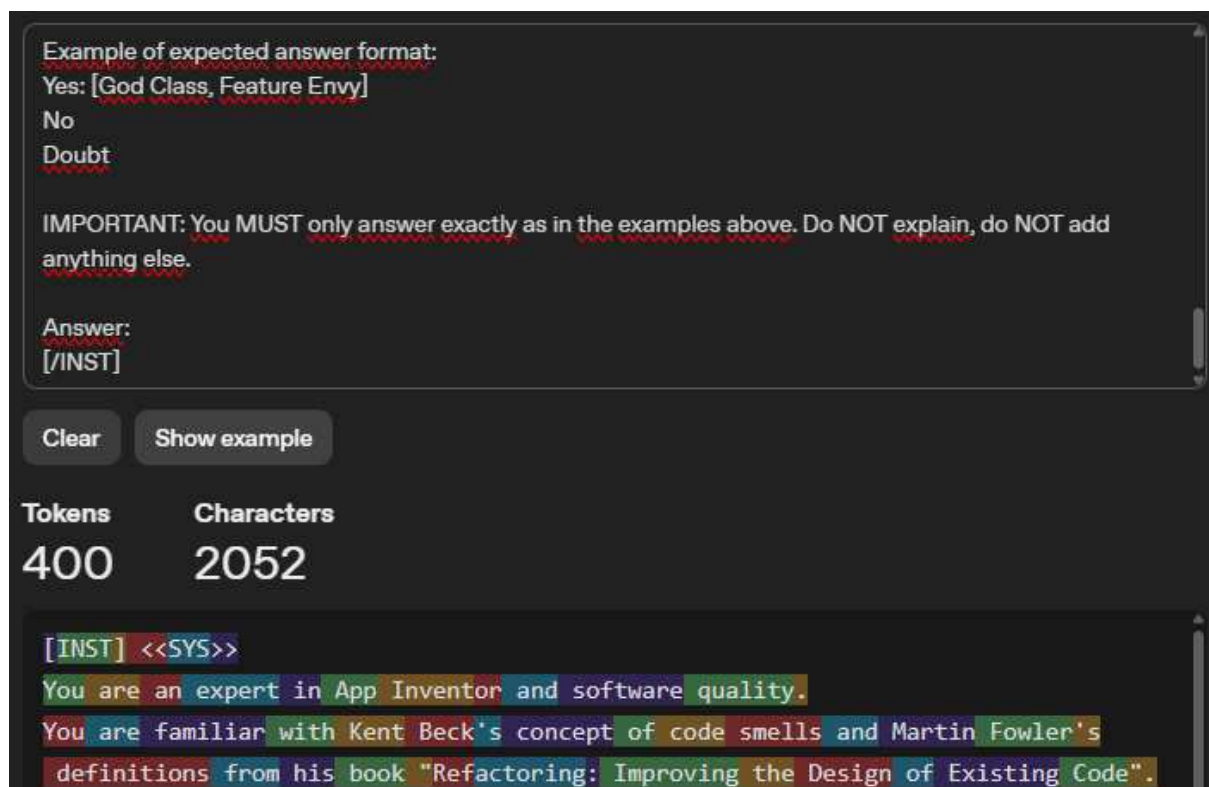
¹⁶ Possível pois o *App Inventor* usa nomenclaturas padrões para esses tipos de blocos: “*global_declaration*”, “*procedures_defreturn*”, “*procedures_defnoreturn*”.

determinado bloco é inserido um aviso de potencial *Lazy Class* no *prompt*. Dessa forma, parte do contexto global perdido é recuperado mesmo utilizando *chunking*.

Objetivando diminuir o número final de *tokens* por requisição para o modelo, foram realizadas análises de gastos de tokens por partes da pergunta.

Primeiramente a quantidade de tokens no *prompt* textual foi analisada de maneira genérica (<https://platform.openai.com/tokenizer>), apresentando por volta de 400 *tokens*, como ilustrado na Figura 19.

Figura 19. Quantidade de Tokens no *Prompt* para RAG



Fonte: Autor (2025)

A segunda restrição utilizada foi diminuir o número de recuperações feitas pelo RAG, de três para dois exemplos. Entretanto, também foi implementado um truncamento, apresentado no Quadro 17. Após ser recuperado do banco vetorial, o arquivo é truncado pelos primeiros 500 *tokens*, ignorando assim o final do código, mas respeitando um limite de 1000 *tokens* para os dois exemplos.

Quadro 17. Função de truncamento do código

```
def truncate_tokens(text, max_tokens, model=MODELO_LLM):  
    enc = get_tokenizer(model)
```

```
ids = enc.encode(text)
if len(ids) <= max_tokens:
    return text
return enc.decode(ids[:max_tokens])
```

Fonte: Autor (2025)

Por fim, o bloco do código analisado, caso necessário, também passa pelo truncamento de 500 *tokens*. Desta forma, espera-se que a entrada se mantenha perto dos 2048 *tokens* e que na maioria dos casos não ultrapasse esse limite.

Por conta da separação por blocos, foram realizadas invocações do modelo para cada bloco da entrada, as respostas foram armazenadas e então realiza-se uma chamada final para sumarizar as respostas do modelo para cada bloco, seguindo o *template* apresentado no Quadro 18.

Quadro 18. *Template* para Sumarização das Respostas por Blocos

```
LAST_MESSAGE_TO_SUMMARIZE = """
[INST]
Based on the provided answers. Provide an answer that summarizes the
presence of Lazy Class, Feature Envy and/or God Class in the entire
application.
Follow the answer format using exactly one of the following options:
Yes: [list the detected smells]
No
Doubt

Answers provided:
{previous_answers}

Summarize answer:
[/INST]
"""
```

Fonte: Autor (2025)

5.7.2 Análise dos Resultados do Segundo Experimento

Apenas um modelo foi utilizado para executar esse segundo experimento, optou-se arbitrariamente pelo *Llama* 3.1 por conter o maior número de parâmetros (8 Bilhões). Em relação à resposta apresentada no primeiro experimento (Tabela 10), apresentou-se uma melhoria considerável na qualidade das respostas que podem ser observadas na Tabela 13.

Tabela 13. Métricas da Segunda Execução - *Llama 3.1*

<i>Code Smell</i>	<i>Precision</i>	<i>Recall</i>	<i>F1-Score</i>	<i>Accuracy</i>
<i>Lazy Class</i>	0,170	0,892	0,286	0,482
<i>Feature Envy</i>	0,176	0,981	0,300	0,489
<i>God Class</i>	0,152	0,685	0,248	0,766
Nenhum	0,994	0,215	0,354	0,392

Fonte: Autor (2025)

Da Tabela 13 destaca-se a *precision* para a classificação de quando foi identificado não conter um *smell* (0,994). Contudo, o baixo *recall* de 0,215 aponta que muitos falsos negativos aconteceram, isto é, muitos códigos classificados como não contendo *smell* na base de dados, foram marcados como contendo um dos três apresentados.

De maneira geral o *F1-Score* também melhorou, inclusive triplicando o resultado para “*Lazy Class*” e “Nenhum”. De mesmo modo, a Tabela 14 também indica que a interpretação de Cohen’s Kappa melhorou de modo geral. Contudo, ainda assim os valores obtidos não são satisfatórios. Mas mostram que diferentes experimentações são necessárias e podem apresentar melhorias significativas mesmo com modelos limitados.

Tabela 15. Cohen’s Kappa - Segunda Execução - *Llama 3.1*

Modelo	<i>Smell</i>	Kappa	Interpretação
<i>Llama 3.1</i>	<i>Lazy Class</i>	0,113	Fraca
	<i>Feature Envy</i>	0,136	Fraca
	<i>God Class</i>	0,172	Fraca
	Nenhum	0,108	Fraca

Fonte: Autor (2025)

5.8 DISCUSSÃO

As decisões tecnológicas adotadas foram fortemente influenciadas pelas restrições impostas ao projeto, os resultados teriam sido mais satisfatórios caso modelos maiores, com maior janela de contexto tivessem sido usados. O tempo de execução também teria sido muito menor visto que a execução de cada modelo no primeiro experimento levou cerca de seis horas, enquanto que no segundo cerca de 24 horas para os 956 arquivos.

Os *code smells* selecionados foram definidos com o objetivo de abordar aspectos que a análise estática clássica nem sempre captura de forma eficaz. Contudo, seria possível adicionar, ou trocar, um dos *smells* por algum com identificação mais direta, como *Dead Code*, pois o resultado obtido de um *smell* simples pode ser utilizado para tomar decisões sobre o sistema, auxiliando na identificação de possíveis falhas de configuração ou problemas na execução.

Após a execução do projeto, é levantada a necessidade de diminuir o número de *tokens* da entrada, isto é, aplicar um pré-processamento mais eficiente. Como apresentado durante todo o processo, os códigos “.bky” são armazenados em XML, cuja estrutura gera, em média, mais *tokens* que seu equivalente em JSON. Uma possível abordagem para otimizar a representação seria converter o XML para JSON, em seguida, transformar o JSON em um formato orientado a tokens, como TOON¹⁷, visando melhorar a eficiência do processamento pelos modelos e diminuindo possíveis truncamentos de contexto.

O *prompt* utilizado foi desenvolvido em um formato geral, compatível com todas as LLMs avaliadas. Contudo, ele pode ser melhorado com a utilização de *tags* específicas, pois alguns modelos disponibilizam *tags* que orientam melhor o modelo sobre a entrada. Embora, neste trabalho, tenha sido adotado um formato uniforme, adaptar o prompt para cada modelo individualmente pode potencialmente melhorar a qualidade e a consistência das respostas geradas.

Sobre a análise dos resultados, observa-se discrepância entre valores de acurácia e as demais métricas, isso acontece devido aos modelos acertarem muitas classificações simplesmente por não identificar a presença do *smell*. Trabalhos futuros podem explorar medidas como o Matthews Correlation Coefficient (MCC), utilizado por Cruz, Santana e Figueiredo (2026), que levam em conta verdadeiros

¹⁷ <https://github.com/toon-format/toon>

positivos, falsos positivos, verdadeiros negativos e falsos negativos, fornecendo uma avaliação mais equilibrada do desempenho dos modelos.

Os resultados mostram que modelos locais não estão totalmente preparados para executarem com RAG sem diversas otimizações adicionais, como a redução do número de tokens de entrada. Observa-se que há espaço para melhorias significativas, pois em experimentos com prompts simples na Seção 4.3, os modelos locais apresentaram desempenho superior ao obtido no sistema com RAG, sugerindo que ajustes no pré-processamento e na formulação dos prompts podem impactar positivamente os resultados.

6. CONCLUSÃO

Este trabalho trata do desenvolvimento de um sistema de RAG local com LLM, foram experimentados diferentes modelos com a finalidade de encontrar boas indicações locais que auxiliem na identificação de *code smells* para aplicações *App Inventor*. Entretanto nenhum dos modelos evidenciou bom desempenho, indicando assim que melhorias no sistema devem ser estudadas.

No Capítulo 3, na revisão sistemática da literatura nota-se a escassez de trabalhos relacionados ao tema deste estudo, não sendo encontrado nenhum que abrangesse simultaneamente os três conceitos base: *code smells*, *App Inventor* e LLM. Os poucos trabalhos que abordam *code smells* em programação por blocos, geralmente tratam de estudos de caso com grupos escolares sobre o impacto dos *smells*. O trabalho de Hermans, Stolee e Hoepelman (2016), que trata da análise da presença de *smells* em programação por blocos, foi incluído na revisão da literatura e uma influência direta na definição de *smells* na Seção 5.2.

Também durante a revisão da literatura, no Capítulo 3, observa-se que o estado da arte na identificação de *code smells* não envolve o uso de LLMs, sendo baseado principalmente em algoritmos clássicos e suas adaptações para detecção de *smells*. Trabalhos como Wu et al. (2024) indicam o uso de LLMs focado em refatoração, e não na identificação de *smells*. Já Sadik e Govind (2025) apresentam um *benchmark* comparando *DeepSeek-V3* e *GPT-4.0*, que, apesar de se tratarem de modelos maiores, não obtiveram bons resultados na identificação de *smells*, apresentando um recall baixo de forma geral.

No Capítulo 4, foram conduzidos experimentos iniciais com modelos web e locais, com o objetivo de orientar a escolha de modelos locais para este estudo. No estudo, foram utilizadas versões mais leves de LLMs, amplamente conhecidas e disponibilizadas gratuitamente. No Capítulo 5, realizou-se a preparação, execução e análise dos experimentos com os modelos locais utilizando RAG. A avaliação dos resultados, realizada por meio de métricas de desempenho, não evidenciou modelos com desempenho aceitável para aplicação prática local, indicando a necessidade de aprimoramentos no sistema, no pré-processamento e nas estratégias de *prompt*.

Durante a execução do projeto foram realizados experimentos, no Capítulo 4 e 5, para comparar diferentes modelos, todos modelos foram testados sobre o uso

de RAG. O modelo *Phi 4 Mini* não conseguiu gerar mais do que dez respostas seguidas sem alucinar, mesmo com uma nova instância da classe `ChatOllama` a cada pergunta enviada, tornando seu uso inviável. O modelo *Deepseek-R1* não apresentou alucinação, contudo, suas respostas em grande maioria geraram texto, não respeitando os padrões definidos, mas ainda assim permitindo sua análise. Assim dentro dos limites impostos, os modelos foram comparados em sua execução do sistema

Como trabalho futuro, propõe-se diferentes técnicas de *chunking* (segmentação), dos arquivos, por exemplo, em limites de 500 tokens e com *overlapping* (com uso de um modelo maior de *embedding*), pois tende a apresentar um melhor resultado, contudo, como citado na Seção 5.3 o modelo é exposto ao risco de inserir comparações sem sentido para tomada de decisão. Também sobre pontos levantados na Seção 5.8, destaca-se a tradução de arquivos do modelo XML, para um modelo que gere menos *tokens*, assim como uma limpeza mais profunda do XML e também o uso de *prompts* que explorem especificidades de cada modelo.

Outro trabalho futuro proposto é a realização de *fine-tuning* em modelos como *Gemma* e *DeepSeek*, para a identificação de *code smells* em projetos do *App Inventor*. Inicialmente, o desenvolvimento de *fine-tuning* fazia parte do escopo deste trabalho, contudo, após análises durante a implementação e uma revisão da literatura sobre o tema, optou-se por redirecionar o foco para a construção de um sistema baseado em RAG (*Retrieval-Augmented Generation*). Essa mudança permitiu explorar de forma mais viável os desafios propostos neste trabalho.

REFERÊNCIAS

ALVES, Nathalia da Cruz; VON WANGENHEIM, Christiane Gresse. Uma Análise em Larga-Escala das Funcionalidades de Aplicativos criados com App Inventor. In: Anais do III Simpósio Brasileiro de Educação em Computação, online, Brasil, 2023.

ARCELLI FONTANA, F. et al. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, v. 21, n. 3, p. 1143–1191, 6 jun. 2015.

AZEEM, M. I. et al. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, v. 108, p. 115–138, abr. 2019.

BROWN, Tom B. et al. Language Models are Few-Shot Learners. 2020. Disponível em: <<http://arxiv.org/abs/2005.14165>>.

CARAM, F. L. et al. Machine Learning Techniques for Code Smells Detection: A Systematic Mapping Study. *International Journal of Software Engineering and Knowledge Engineering*, v. 29, n. 02, p. 285–316, fev. 2019.

CRUZ, D.; SANTANA, A.; FIGUEIREDO, E. Evaluating a continuous feedback strategy to enhance machine learning code smell detection. *Science of Computer Programming*, v. 247, p. 103346, jan. 2026.

DEEPSEEK AI. *DeepSeek Chat* [ferramenta de inteligência artificial]. Disponível em: <https://chat.deepseek.com/>. Acesso em: 4 jul. 2025.

DEVLIN, J. et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2019. Disponível em: <<https://arxiv.org/pdf/1810.04805>>.

DI NUCCI, D. et al. Detecting code smells using machine learning techniques: Are we there yet? 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), mar. 2018.

ELDAR SULTANOW et al. Machine Learning based Static Code Analysis for Software Quality Assurance. 1 set. 2018.

FISHELSON, R. I; HERSHKOVITZ, A. Studying interrelations of computational thinking and creativity: A scoping review (2011–2020). *Computers & Education*. n. 186, p. 104353, 2022.

FOWLER, M.; BECK, K. Refactoring : improving the design of existing code. Boston: Addison-Wesley, 2018.

GALANTE, G. F.; HAUCK, J. C. R. Análise de Code Smells em Linguagens Visuais de Programação no Contexto da Educação: Um Mapeamento Sistemático da Literatura. Anais do XXXIII Workshop sobre Educação em Computação (WEI 2025), p. 758–771, 20 jul. 2025.

GOODFELLOW, Ian et al. Deep Learning. Cambridge, MA: MIT Press, 2016.

GOOGLE. *Gemini* [ferramenta de inteligência artificial]. Disponível em: <https://gemini.google.com/>. Acesso em: 4 jul. 2025.

GOOGLE. *NotebookLM* [ferramenta de inteligência artificial]b. Disponível em: <https://notebooklm.google/>. Acesso em: 4 jul. 2025.

GUGGULOTHU, T.; MOIZ, S. A. Code smell detection using multi-label classification approach. Software Quality Journal, v. 28, n. 3, p. 1063–1086, 4 abr. 2020.

HERMANS, F.; EFTHIMIA AIVALOGLOU. Do code smells hamper novice programming? A controlled experiment on Scratch programs. 1 maio 2016.

HERMANS, F.; STOLEE, K. T.; HOEPELMAN, D. Smells in block-based programming languages. p. 68–72, 1 set. 2016.

KIYAK, E. O.; BIRANT, D.; BIRANT, K. U. Comparison of multi-label classification algorithms for code smell detection. 2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT). Anais...IEEE, 2019.

LACERDA, G. et al. Code smells and refactoring: A tertiary systematic review of challenges and observations. Journal of Systems and Software, v. 167, p. 110610, set. 2020.

LANDIS, R. J.; KOCH, G. G. The Measurement of Observer Agreement for Categorical Data. Biometrics, v. 33, n. 1, p. 159–174, mar. 1977.

LEWIS, P. et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Disponível em: <<https://arxiv.org/abs/2005.11401>>.

LI, I.; TURBAK, F.; ENI MUSTAFARAJ. Calls of the wild: Exploring procedural abstraction in app inventor. p. 79–86, 1 out. 2017.

MARTIN, R. C. Clean Code a Handbook of Agile Software craftsmanship. [s.l.] Prentice Hall, 2008.

MIT APP INVENTOR COMMUNITY. AIA file structure. MIT App Inventor Community, 2019. Disponível em: <<https://community.appinventor.mit.edu/t/aia-file-structure/219>>. Acesso em: 18 abr. 2025.

OUYANG, L. et al. Training language models to follow instructions with human feedback. 2022. Disponível em: <<https://arxiv.org/pdf/2203.02155>>.

OPENAI. *ChatGPT* [ferramenta de inteligência artificial]. Disponível em: <https://chat.openai.com/>. Acesso em: 4 jul. 2025.

PATTON, Evan; TISSENBAUM, Michael; HARUNANI, Farzeen. MIT App Inventor: Objectives, Design, and Development. In: ABELSON, Harold (Org.). Computational Thinking Education. Springer. 2019.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic Mapping Studies in Software Engineering. *EASE – 12th International Conference on Evaluation and Assessment in Software Engineering*, 2008

RADFORD, A. et al. Improving Language Understanding by Generative Pre-Training. [s.l.] OpenAI, 2018. Disponível em: <https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf>.

RUSSELL, S; NORVIG, P. Artificial intelligence: a modern approach. 3. ed. Chennai, TN, India: Pearson Education India, 2015.

SADIK, A. R.; GOVIND, S. Benchmarking LLM for Code Smells Detection: OpenAI GPT-4.0 vs DeepSeek-V3. Disponível em: <<https://arxiv.org/abs/2504.16027v1>>. Acesso em: 4 jul. 2025.

SUTTON, R. S.; BARTO, A. G. Reinforcement Learning: An Introduction. 2. ed. Cambridge, MA, USA: Bradford Books, 2018.

TECHAPALOKUL, P.; TILEVICH, E. Code Quality Improvement for All: Automated Refactoring for Scratch. 2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), out. 2019.

TRICCO, A. C. et al. A Scoping Review of Rapid Review Methods. BMC Medicine, v. 13, n. 1, 16 set. 2015.

TSAI, C.-Y. Improving students' understanding of basic programming concepts through visual programming language: The role of self-efficacy. Computers in Human Behavior, v. 95, p. 224–232, jun. 2019.

VASWANI, A. et al. Attention Is All You Need. Disponível em: <<https://arxiv.org/abs/1706.03762>>.

WOHLIN, Claes et al. Experimentation in Software Engineering. Springer, 2012.

WU, D. et al. ISMELL: Assembling LLMs with expert toolsets for code smell detection and refactoring. International Conference on Automated Software Engineering, p. 1345–1357, 2024.

XU, B. et al. ExpertPrompting: Instructing Large Language Models to be Distinguished Experts, 2025. Disponível em: <<https://arxiv.org/abs/2305.14688>>.

YADAV, Aman; COOPER, Steve. Communications of the ACM, 60(2), p. 31-33, 2017.

YAHYA, H. M.; TAHA, D. B. Detection Bad Code Smells By Using Deep Machine Learning Approaches. p. 281–286, 14 out. 2023.

YANG, Z. et al. The impact of a block-based visual programming curriculum: Untangling coding skills and computational thinking. Learning and Instruction, v. 95, p. 102041–102041, 29 out. 2024.

YIN, Robert K. *Case Study Research and Applications: Design and Methods*. 6. ed. SAGE Publications, 2017.

APÊNDICES

Análise de códigos usando large language models para identificação de code smell em aplicativos criados com App Inventor

Marco Antônio Machado de Arruda¹, Jean Carlo Rossa Hauck¹

¹Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brasil

marco.antonio.machado.arruda@grad.ufsc.br, jean.hauck@ufsc.br

Abstract. *In introductory computing education, visual programming languages are often used as more accessible tools for learning. To reduce errors and prevent the formation of poor programming habits, this work investigates the automatic detection of code smells in projects developed with the App Inventor platform, employing a Retrieval Augmented Generation (RAG)-based approach combined with Large Language Models (LLMs). Several local models were evaluated; however, computational limitations and the simplicity of the available models prevented satisfactory results in identifying these smells.*

Resumo. *No ensino introdutório de computação, linguagens visuais são empregadas como ferramentas mais acessíveis ao aprendizado. Para reduzir erros e evitar a formação de maus hábitos, este trabalho investiga a detecção automática de code smells em projetos desenvolvidos na plataforma App Inventor, empregando a técnica de Retrieval Augmented Generation (RAG) combinada a Large Language Models (LLMs). Diferentes modelos locais foram avaliados; entretanto, limitações computacionais e a simplicidade dos modelos disponíveis impediram resultados satisfatórios na identificação dos smells.*

1. Introdução

O uso de linguagens de programação visuais (Visual Programming Languages - VPLs) tem recebido atenção na área de ensino de computação, especialmente por facilitar o primeiro contato de estudantes com conceitos fundamentais da programação. Pesquisas indicam que essas linguagens são amplamente utilizadas por alunos jovens, entre 12 e 14 anos [Hermans and Aivaloglou 2016], evidenciando o acesso à tecnologia por crianças e adolescentes. Nesse contexto, o desenvolvimento de aplicativos móveis surge como uma oportunidade de introduzir noções de computação desde a educação básica, apoiado por ferramentas como o App Inventor, que torna a programação mais intuitiva e acessível [Patton et al. 2019].

Além de seu papel educacional, o App Inventor atua como um instrumento de democratização do desenvolvimento de aplicativos, alcançando mais de 23 milhões de usuários e tendo sido utilizado para criar mais de 115 milhões de aplicações [Li et al. 2017]. Entretanto, apesar de seu caráter introdutório e acessível, é comum que iniciantes produzam soluções que apresentam fragilidades. Nesse cenário, destacam-se os code smells, conceito amplamente difundido por Fowler (1999), definidos como padrões

que, embora não constituam erros formais, sinalizam possíveis problemas de manutenção e evolução do software. Identificar esses indícios desde os primeiros contatos dos estudantes com programação contribui para a formação de desenvolvedores mais conscientes de boas práticas [Fowler 2018].

Considerando essa necessidade, este trabalho investiga a identificação automática de code smells em ambientes de programação visual utilizados por iniciantes. Para isso, são exploradas abordagens baseadas em aprendizado de máquina, com ênfase em técnicas de prompt-based learning [Brown et al. 2020] e RAG utilizando modelos de linguagem de larga escala (LLMs) [Lewis et al. 2021]. O objetivo é avaliar a eficácia dessas técnicas na análise de projetos desenvolvidos no App Inventor e compreender seu potencial para apoiar o ensino de boas práticas de desenvolvimento desde as etapas iniciais da formação em computação.

2. Estado da Arte

Na etapa de trabalhos correlatos, foi conduzida uma revisão rápida da literatura, conforme as diretrizes de [Tricco et al. 2015]. Para embasar as decisões do projeto, realizaram-se buscas nas principais bases científicas acessíveis a estudantes da UFSC, incluindo IEEE, ScienceDirect, Scopus e Google Scholar. Inicialmente, foram analisadas as dez primeiras páginas de resultados de cada base utilizando a string de busca ("code smell"OR "bad smell") AND ("machine learning"OR "LLM") AND ("VPL"OR "visual programming language"OR "App Inventor"). Como o número de trabalhos relevantes foi pequeno, a expressão de busca foi ampliada com a remoção dos termos referentes às VPLs.

A análise preliminar resultou na seleção de 13 trabalhos correlatos, dos quais nove abordam code smells associados a técnicas de machine learning, um discute machine learning aplicado à análise estática de código e três tratam de smells em linguagens de programação visual. Após a seleção, os estudos foram avaliados segundo as questões definidas no protocolo de pesquisa, considerando: (i) quais smells foram identificados; (ii) quais métodos de machine learning foram utilizados na detecção; (iii) como os code smells foram tratados ou corrigidos; (iv) qual o impacto na identificação e análise dos code smells na qualidade do software; e (v) Quais métricas foram adotadas na avaliação do estudo. A partir dessa análise, considerando também a inexistência de um dataset previamente rotulado, foram destacadas duas abordagens: fine-tuning de um modelo ou uso de LLM com RAG, sendo esta última a técnica selecionada para o estudo.

Não foram identificados trabalhos voltados para a detecção de smells especificamente em linguagens visuais. Dessa forma, o presente trabalho baseou-se em características gerais da análise tradicional, complementadas por processamentos adicionais decorrentes das particularidades dos códigos em VPLs, especialmente no App Inventor. Como base para a definição dos smells analisados, adotaram-se os trabalhos de Galante e Hauck [Galante and Hauck 2025], que destacam smells recorrentes em linguagens de programação visual, juntamente com a definição clássica de code smells apresentada por Fowler [Fowler 2018]. Além disso, conceitos foram adaptados ao contexto do App Inventor de maneira semelhante ao realizado em [Hermans et al. 2016].

3. Experimentos de Identificação de Code Smells

O primeiro experimento teve como objetivo avaliar a exequibilidade do projeto. Para isso, utilizaram-se modelos web de LLMs. Realizou-se um teste preliminar empregando one-shot prompting. Nesta etapa inicial, não houve uso de RAG; os arquivos de programação do App Inventor foram fornecidos diretamente no prompt, juntamente com exemplos inseridos manualmente. Os resultados indicaram que os modelos são capazes, de modo geral, de interpretar os blocos de código e identificar possíveis code smells.

O segundo e mais prolongado experimento consistiu no uso de modelos locais para identificar três code smells selecionados: Lazy Class, Feature Envy e God Class. Para isso, foram rotulados manualmente 383 aplicativos, selecionados de um conjunto maior de 88.863 aplicações do App Inventor, indicando a presença ou ausência desses smells. A partir dessa rotulagem, foram extraídos seis exemplos positivos para cada smell, bem como seis exemplos negativos, que foram incorporados ao banco vetorial utilizado na etapa de RAG.

A implementação envolveu três componentes principais: (i) o processamento e tratamento dos arquivos de código em blocos do App Inventor; (ii) a construção de uma arquitetura RAG utilizando o banco vetorial FAISS; e (iii) o desenvolvimento da estrutura de prompts para as requisições aos modelos LLM.

No design do prompt, empregaram-se técnicas de roleplaying [Xu et al. 2025], com a inserção de metadados destinados a enriquecer os exemplos com explicações, justificativas e formatos de saída esperada. Também foram incluídas definições específicas dos smells adaptadas ao contexto do App Inventor, bem como modelos padronizados de resposta para orientar o comportamento dos modelos de linguagem.

O terceiro experimento teve como objetivo otimizar a qualidade das respostas obtidas no segundo experimento. Para mitigar problemas decorrentes das restrições de hardware, especialmente a janela de contexto reduzida dos modelos locais, que frequentemente ultrapassava o limite de 2.048 tokens, foi introduzido um processo de chunking baseado em “blocos raiz”, isto é, blocos do App Inventor que não possuem elementos que os encapsulem na interface. A estratégia foi motivada pelo fato de que, no experimento anterior, diversas respostas eram geradas sem referência adequada ao próprio código sob análise, indicando perda de contexto devido ao excesso de tokens.

A divisão dos códigos em blocos exigiu adaptações adicionais no sistema, especialmente o envio de uma requisição ao modelo para cada bloco individual. Essa abordagem aumentou significativamente o tempo de execução, aproximadamente quadruplicando a duração total do processo, que passou a demandar cerca de 24 horas para o modelo Llama 3.1 8B. Além disso, foi incorporado um novo metadado obtido por meio de uma varredura prévia dos arquivos antes do particionamento, identificando se variáveis globais, funções ou procedimentos são utilizados no código. Quando identificadas declarações sem uso, o prompt recebe um aviso adicional de “possível Lazy Class”.

Complementarmente ao chunking, foram aplicados truncamentos mais restritivos nos exemplos utilizados na RAG. Cada requisição passou a incluir no máximo dois exemplos, limitados a 500 tokens cada. O prompt textual propriamente dito ocupou cerca de 400 tokens, enquanto o bloco de código avaliado também foi truncado para aproximadamente 500 tokens, buscando manter a soma dentro do limite de 2.048 tokens na maioria

dos casos.

4. Análise dos Resultados

Os experimentos que empregaram RAG, o segundo e o terceiro, foram avaliados utilizando a base de dados rotulada previamente. As métricas de análise foram selecionadas a partir dos trabalhos correlatos e incluem Precision, Recall, F1-Score e Accuracy. Considerando a natureza subjetiva da identificação de code smells, adicionou-se também o índice Kappa de Cohen, métrica utilizada para avaliar o grau de concordância entre especialistas.

4.1. Análise Segundo Experimento

O segundo experimento evidenciou principalmente as limitações impostas pela execução local dos modelos. Utilizando uma GPU GTX 1660 com 6 GB, o desempenho dos modelos foi significativamente comprometido. Entre os três modelos avaliados, DeepSeek R1, Gemma 3 e Llama 3.1, nenhum alcançou resultados satisfatórios, como pode ser observado nas Tabelas 1, 2 e 3, respectivamente.

De forma geral, o valor elevado de accuracy ocorre devido ao desbalanceamento da base rotulada. Como a maioria das amostras não contém determinados smells, a simples não identificação por parte do modelo já resulta em um “acerto” em muitos casos, mesmo quando outras previsões são incorretas.

Tabela 1. Métricas da Execução - DeepSeek R1

Code Smell	Precision	Recall	F1-Score	Accuracy
Lazy Class	0,111	0,063	0,080	0,833
Feature Envy	0,150	0,160	0,155	0,806
God Class	0	0	0	0,893
Nenhum	0,872	0,675	0,761	0,673

Fonte: Autor (2025)

Para o modelo DeepSeek (Tabela 1), observa-se um desempenho particularmente ruim na identificação do smell God Class, apresentando precision e recall iguais a zero, o que indica que nenhuma ocorrência foi corretamente reconhecida. A classe “Nenhum”, correspondente à ausência de smells, apresentou o melhor desempenho do modelo. Esse resultado, contudo, pode ser explicado pela natureza introdutória dos projetos desenvolvidos no App Inventor, que frequentemente contém poucos blocos de lógica ou até mesmo nenhum. Essa característica tende a aumentar artificialmente o número de instâncias classificadas como “Nenhum”, favorecendo essa categoria mesmo quando o modelo falha em identificar outros smells.

Tabela 2. Métricas da Execução - Gemma 3

Code Smell	Precision	Recall	F1-Score	Accuracy
Lazy Class	0,083	0,045	0,058	0,832
Feature Envy	0,132	0,774	0,226	0,411
God Class	0,077	0,759	0,139	0,470
Nenhum	0,800	0,081	0,147	0,274

Fonte: Autor (2025)

O modelo Gemma 3 apresentou também resultados fracos (Tabela 2). Contudo, tanto para God Class quanto para Feature Envy, registrou valores razoavelmente altos de recall (0,759 e 0,774, respectivamente). Isso indica que o modelo raramente deixou de identificar instâncias que realmente pertenciam a esses smells. No entanto, ao observar a precision desses mesmos smells, nota-se valores muito baixos, revelando que o modelo superestimou sua ocorrência. Em outras palavras, o Gemma classificou muitos trechos como God Class ou Feature Envy quando, na verdade, não eram, o que explica o recall elevado, porém acompanhado de uma elevada taxa de falsos positivos.

Tabela 3. Métricas da Execução - Llama 3.1

Code Smell	Precision	Recall	F1-Score	Accuracy
Lazy Class	0,061	0,189	0,092	0,566
Feature Envy	0,125	0,132	0,128	0,801
God Class	0,117	0,889	0,206	0,613
Nenhum	0,714	0,054	0,101	0,252

Fonte: Autor (2025)

Para o Llama 3.1, a avaliação segue um padrão semelhante ao observado no Gemma 3. Entretanto, o modelo apresentou valor alto de recall somente para God Class (Tabela 3). Assim como no caso anterior, a precision baixa indica a ocorrência de muitos falsos positivos, ou seja, o modelo inferiu a presença de God Class com frequência, classificando como tal diversos casos que não estavam rotulados com esse smell.

De forma geral, observam-se valores de F1-Score baixos, o que evidencia uma grande discrepância entre precision e recall. Essa diferença em relação aos dados rotulados indica uma ruptura na concordância entre as previsões dos modelos e os rótulos de referência. Tal interpretação é reforçada pela Tabela 4, referente ao coeficiente Kappa de Cohen, que aponta predominantemente para uma concordância fraca, e, em alguns casos, desempenho pior do que o acaso.

Tabela 4. Cohen's Kappa - Todos os Modelos

Modelo	Smell	Kappa	Interpretação
DeepSeek-R1	Lazy Class	-0.004	Concordância pior que o acaso
	Feature Envy	0.046	Fraca
	God Class	-0.056	Concordância pior que o acaso
	Nenhum	0.266	Razoável
Gemma 3	Lazy Class	-0.025	Concordância pior que o acaso
	Feature Envy	0.045	Fraca
	God Class	0.041	Fraca
	Nenhum	0.006	Fraca
Llama 3.1	Lazy Class	-0.102	Concordância pior que o acaso
	Feature Envy	0.016	Fraca
	God Class	0.118	Fraca
	Nenhum	-0.009	Concordância pior que o acaso

Fonte: Autor (2025)

4.2. Análise Terceiro Experimento

O terceiro experimento consistiu em uma tentativa de otimizar o processo utilizado no segundo experimento. Nessa etapa, apenas o modelo Llama 3.1 foi avaliado. A Tabela 5 evidencia uma melhoria considerável em comparação à execução anterior do mesmo modelo (Tabela 3), indicando que as modificações aplicadas, como o uso de chunking por blocos raiz e ajustes no prompt, contribuíram para resultados mais consistentes.

Tabela 5. Métricas da Terceira Execução - Llama 3.1

Code Smell	Precision	Recall	F1-Score	Accuracy
Lazy Class	0,170	0,892	0,286	0,482
Feature Envy	0,176	0,981	0,300	0,489
God Class	0,152	0,685	0,248	0,766
Nenhum	0,994	0,215	0,354	0,392

Fonte: Autor (2025)

Os resultados da terceira execução, apresentados na Tabela 5, também demonstram desempenho insatisfatório de forma geral. Entretanto, alguns pontos se destacam quando comparados à execução anterior (Tabela 3). O caso mais evidente é o valor excepcionalmente alto de precision para a classe “Nenhum”, que atingiu 0,994. Esse resultado indica que o modelo, apesar do recall baixo, foi capaz de identificar corretamente, quase sem erros, os arquivos que realmente não continham code smells.

O recall baixo revela que muitos arquivos sem smells deixaram de ser classificados como “Nenhum”, mas a elevada precision mostra que, nos casos em que o modelo decidiu pela classe “Nenhum”, essa decisão foi correta na grande maioria das vezes.

Outras métricas também apresentaram melhora significativa em sua maioria, com exceção do accuracy para Lazy Class e Feature Envy, e do recall para God Class. Alguns valores importantes, como o F1-Score, chegaram a triplicar para as classes “Nenhum” e “Lazy Class”. Embora os valores absolutos ainda permaneçam baixos, a melhoria observada é evidente, especialmente quando analisada em conjunto com o índice de Kappa de Cohen (Tabela 6). Métricas que anteriormente indicaram concordância pior que o acaso passaram a atingir níveis de concordância fraca, demonstrando um avanço na consistência das predições do modelo.

Tabela 6. Cohen's Kappa – Terceira Execução – Llama 3.1

Modelo	Smell	Kappa	Interpretação
Llama 3.1	Lazy Class	0,113	Fraca
	Feature Envy	0,136	Fraca
	God Class	0,172	Fraca
	Nenhum	0,108	Fraca

Fonte: Autor (2025)

5. Ameaças à Validade

Diversas ameaças à validade foram identificadas ao longo deste estudo. Primeiramente, observou-se a ocorrência de respostas imprecisas e desconexas por parte dos modelos, o que sugere limitações impostas pela janela de contexto reduzida (2048 tokens).

Essa restrição pode ter causado truncamentos tanto nos exemplos recuperados pelo RAG quanto no próprio código analisado, prejudicando a qualidade das inferências. Embora tenha sido adotada uma mitigação restringindo a recuperação do FAISS a apenas três documentos (e dois documentos na terceira execução), ainda há indícios de que o limite de contexto tenha sido ultrapassado em várias consultas.

Adicionalmente, o hardware utilizado impôs limitações significativas: a GPU disponível não permitiu manter módulos de raciocínio de modelos como o DeepSeek-R1 ativos, reduzindo o potencial de desempenho dos modelos locais. O modelo de embeddings empregado (all-MiniLM-L6-v2), também com janela reduzida (256 tokens), pode ter sofrido truncamentos tanto na construção do banco vetorial quanto durante a recuperação, afetando a qualidade das similaridades calculadas.

No pré-processamento dos arquivos “.bky”, a utilização da biblioteca XML do Python resultou em alterações estruturais nos documentos, como a remoção ou modificação de tags internas. Esses ajustes podem ter alterado a representação semântica dos blocos, impactando a interpretação dos modelos. Ademais, arquivos com sintaxe quebrada, frequentes no conjunto original e muitas vezes associados à Lazy Class, não foram inseridos na base usada no RAG, embora estivessem presentes na fase de rotulação. Essa discrepância pode ter comprometido a coerência entre os dados usados para treino, recuperação e avaliação.

Por fim, o conjunto de dados utilizado é um subconjunto previamente selecionado de 2.822 aplicativos entre um total de 88.863, predominantemente em português e com poucos exemplos em outros idiomas. Essa seleção limita a diversidade linguística e estrutural das amostras, reduzindo a generalização dos resultados obtidos para outros contextos ou populações.

6. Conclusão e Trabalhos Futuros

Este trabalho desenvolveu um sistema local de RAG combinado com LLMs para auxiliar na identificação de code smells em projetos App Inventor. Diferentes modelos leves foram avaliados, mas nenhum apresentou desempenho satisfatório para uso prático. A análise evidenciou que limitações de hardware, janelas de contexto reduzidas, restrições no processo de recuperação e pré-processamento impactaram significativamente os resultados.

A revisão da literatura mostrou que não há estudos que integrem simultaneamente code smells, App Inventor e LLMs. Além disso, os trabalhos que investigam smells em VPLs geralmente discutem apenas seus efeitos, e não sua identificação direta, indicando que essa área ainda carece de pesquisa. Os experimentos conduzidos neste trabalho contribuem para essa lacuna, embora seus resultados não tenham se mostrado eficazes. De modo geral, o estudo conclui que, embora o uso de LLMs em conjunto com RAG seja promissor, melhorias substanciais ainda devem ser realizadas no sistema, no pré-processamento e nas estratégias de prompting para tornar a abordagem viável.

A seguir são destacados alguns pontos de trabalhos futuros.

- **Exploração de novas técnicas de chunking**, incluindo segmentação com *overlap* e limites de tokens mais estáveis, utilizando modelos de embedding maiores, de

modo a melhorar a qualidade da recuperação sem comprometer a coerência das inferências.

- **Refinamento do pré-processamento dos arquivos .bky**, possivelmente traduzindo a estrutura XML para formatos mais compactos ou estáveis, reduzindo o número de tokens e diminuindo distorções estruturais introduzidas pela biblioteca XML.
- **Desenvolvimento de prompts especializados**, adaptados às características de cada modelo, para reduzir alucinações e melhorar a aderência às regras de saída.
- **Treinamento via fine-tuning** de modelos como Gemma e DeepSeek para detecção específica de *code smells* em App Inventor, retomando uma etapa originalmente planejada, mas não implementada neste estudo.
- **Avaliação com hardware mais robusto**, permitindo a execução de modelos com janelas de contexto maiores ou com módulos de raciocínio ativados, o que pode impactar significativamente a qualidade das previsões.

Referências

- 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.
- Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2nd edition.
- Galante, G. and Hauck, J. (2025). Análise de code smells em linguagens visuais de programação no contexto da educação: Um mapeamento sistemático da literatura. In *Anais do XXXIII Workshop sobre Educação em Computação*, pages 758–771, Porto Alegre, RS, Brasil. SBC.
- Hermans, F. and Aivaloglou, E. (2016). Do code smells hamper novice programming? a controlled experiment on scratch programs. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10.
- Hermans, F., Stolee, K. T., and Hoepelman, D. (2016). Smells in block-based programming languages. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 68–72.
- Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., tau Yih, W., Rocktäschel, T., Riedel, S., and Kiela, D. (2021). Retrieval-augmented generation for knowledge-intensive nlp tasks.
- Li, I., Turbak, F., and Mustafaraj, E. (2017). Calls of the wild: Exploring procedural abstraction in app inventor. In *2017 IEEE Blocks and Beyond Workshop (BB)*, pages 79–86.
- Patton, E. W., Tissenbaum, M., and Harunani, F. (2019). *MIT App Inventor: Objectives, Design, and Development*, pages 31–49. Springer Singapore, Singapore.

Tricco, A., Antony, J., Zarin, W., Striffler, L., Ghassemi, M., Ivory, J., Perrier, L., Hutton, B., Moher, D., and Straus, S. (2015). A scoping review of rapid review methods. *BMC medicine*, 13:224.

Xu, B., Yang, A., Lin, J., Wang, Q., Zhou, C., Zhang, Y., and Mao, Z. (2025). Expert-prompting: Instructing large language models to be distinguished experts.