UNIVERSIDADE FEDERAL DE SERGIPE
DEPARTAMENTO DE COMPUTAÇÃO
EVOLUÇÃO DE SOFTWARE
DOCENTE: GLAUCO DE FIGUEIREDO CARNEIRO

BRUNO AMANCIO FERREIRA
GÉSSICA KELLY DE SOUZA SANTOS
IAGO HUMBERTO DA ROSA NORMANDIA
LETICIA DA MATA CAVALCANTI
MARIA FERNANDA DA MOTA DINIZ
PEDRO HENRIQUE GOMES DOS SANTOS
SÂMMYA EMANUELLE GUIMARÃES DE OLIVEIRA
WENDERSON LUIZ PORTELA DA SILVA

Atividade 2 – Refactoring e Code Smells

SÃO CRISTÓVÃO - SE
2025

UNIVERSIDADE FEDERAL DE SERGIPE
DEPARTAMENTO DE COMPUTAÇÃO
EVOLUÇÃO DE SOFTWARE

BRUNO AMANCIO FERREIRA
GÉSSICA KELLY DE SOUZA SANTOS
IAGO HUMBERTO DA ROSA NORMANDIA
LETICIA DA MATA CAVALCANTI
MARIA FERNANDA DA MOTA DINIZ
PEDRO HENRIQUE GOMES DOS SANTOS
SÂMMYA EMANUELLE GUIMARÃES DE OLIVEIRA
WENDERSON LUIZ PORTELA DA SILVA

Atividade 2 – Refactoring e Code Smells

Atividade apresentada à disciplina de Evolução de Software como requisito para obtenção de nota.

Prof. Dr.: Glauco de Figueiredo Carneiro

SÃO CRISTÓVÃO - SE

2025

SUMÁRIO

1. Introdução

A qualidade de um sistema de software está diretamente relacionada à sua capacidade de evoluir ao longo do tempo de forma sustentável. À medida que um projeto cresce e sofre sucessivas modificações, decisões de design inadequadas podem se acumular no código fonte, dificultando sua compreensão, manutenção e evolução. Nesse contexto, surgem os code smells, que são indícios de problemas estruturais no código e podem sinalizar a necessidade de refatoração, ainda que não representam, por si só, defeitos funcionais.

A refatoração consiste em um conjunto de transformações no código que visam melhorar sua estrutura interna sem alterar o comportamento externo do sistema. A identificação sistemática de code smells ao longo da evolução de um projeto permite avaliar a qualidade do código, compreender tendências de degradação ou melhoria e apoiar decisões técnicas relacionadas à manutenção do software.

Com o avanço dos modelos de linguagem baseados em inteligência artificial, novas abordagens têm sido exploradas para apoiar atividades de engenharia de software, incluindo a análise automática de código fonte. Esses modelos apresentam potencial para auxiliar na identificação de code smells, oferecendo uma perspectiva complementar às técnicas tradicionais de análise estática. Um vídeo explicativo sobre os conceitos abordados neste trabalho está disponível em https://youtu.be/DcEwFfjL-Wk.

Diante desse cenário, este trabalho tem como objetivo analisar a evolução dos code smells ao longo das releases do projeto Cherry, utilizando uma abordagem baseada em modelos de linguagem. O projeto Cherry é um software de código aberto, cujo repositório original está disponível publicamente no GitHub em https://github.com/CherryHQ/cherry-studio, permitindo o acesso ao histórico completo de versões e alterações do sistema.

Para a realização do estudo, foi feita a coleta das releases do projeto Cherry a partir do GitHub, seguida da seleção de uma amostra representativa correspondente a 30% das versões disponíveis. A análise emprega diferentes modelos de linguagem para identificar e comparar a ocorrência de code smells, bem como avaliar a efetividade desses modelos e o impacto das decisões de design na evolução do software.

Todo o processo de coleta de dados, execução dos scripts, aplicação dos modelos de linguagem e geração dos resultados foi implementado e documentado em um repositório próprio, desenvolvido especificamente para esta atividade acadêmica. O repositório contendo o código, os scripts e os resultados obtidos neste trabalho está disponível em https://github.com/fernandamotad/Evolucao_Software_2025-2_Cherry_Atividade2, permitindo a reprodução dos experimentos e a verificação dos artefatos gerados.

Os resultados obtidos contribuem para a compreensão do comportamento evolutivo do projeto analisado, possibilitando a identificação de padrões de melhoria ou degradação da qualidade do código ao longo do tempo, além de discutir o papel dos modelos de linguagem como ferramentas de apoio à engenharia de software.

2. Projeto Analisado

O projeto analisado neste trabalho é o **Cherry**, uma aplicação de código aberto voltada para produtividade e organização, que oferece funcionalidades para criação e gerenciamento de conteúdos de forma estruturada. Trata-se de um sistema em constante evolução, com atualizações frequentes e adição contínua de novas funcionalidades ao longo do tempo. Essas características tornam o Cherry um objeto de estudo adequado para análises relacionadas à evolução de software, manutenção e qualidade de código, permitindo observar como decisões de design impactam a estrutura do código à medida que o projeto cresce.

O Cherry é desenvolvido principalmente com tecnologias do ecossistema web moderno, utilizando TypeScript e JavaScript como linguagens principais, além do ambiente Node.js e diversas bibliotecas e frameworks voltados ao desenvolvimento de aplicações web. O controle de versão é realizado por meio do Git, e o código-fonte é hospedado publicamente na plataforma GitHub, o que possibilita o acesso ao histórico completo de alterações e releases do projeto. O repositório original do projeto está disponível em: https://github.com/CherryHQ/cherry-studio.

3. Releases Selecionadas

3.1. Objetivo

A partir do GitHub, permitindo a seleção de uma amostra representativa de 30% das versões para posterior análise de code smells utilizando modelos de linguagem.

Esta etapa tem como objetivo coletar e organizar as releases do projeto Cherry.

O resultado desta etapa é a geração de arquivos CSV contendo:

- todas as releases do projeto
- a amostra correspondente a 30% das releases

Esses arquivos serão utilizados nas próximas fases para geração de métricas e análise evolutiva.

3.2. Passo a passo

a) Criação da pasta do projeto da atividade

Foi criada a seguinte pasta raiz para organização dos artefatos:

Evolucao_Software_2025-2_Cherry_Atividade2

Estrutura inicial adotada:

Evolucao_Software_2025-2_Cherry_Atividade2
│
├─ scripts
├─ data

b) Criação do ambiente virtual Python

No terminal, a partir da pasta raiz do projeto:

```
python -m venv .venv
```

Ativação do ambiente virtual no Windows PowerShell:

```
.venv\Scripts\activate
```

c) Instalação das dependências

A única biblioteca externa necessária nesta etapa é a biblioteca requests, utilizada para consumir a API do GitHub.

```
pip install requests
```

d) Script para exportação das releases do GitHub

O script desenvolvido realiza as seguintes funções:

- acessa a API oficial do GitHub
- coleta todas as releases do repositório Cherry
- normaliza os dados relevantes
- exporta todas as releases em formato CSVO script “export_releases_csv.py”, gerado com o modelo GPT 5.2, foi salvo no caminho:

```
#!/usr/bin/env python3
import csv
import os
import sys
import requests
from datetime import datetime

API_BASE = "https://api.github.com"


def gh_get(url: str, token: str | None):
    headers = {
        "Accept": "application/vnd.github+json",
        "User-Agent": "evolucao-software-atividade2",
    }
    if token:
        headers["Authorization"] = f"Bearer {token}"

    r = requests.get(url, headers=headers, timeout=60)
    if r.status_code != 200:
        raise RuntimeError(f"Erro {r.status_code} ao chamar
{url}: {r.text[:300]}")
    return r.json()


def list_all_releases(owner: str, repo: str, token: str |
None):
    releases = []
    page = 1
    per_page = 100

    while True:
        url =
f"{API_BASE}/repos/{owner}/{repo}/releases?per_page={per_pag
e}&page={page}"
        data = gh_get(url, token)
        if not data:
            break
        releases.extend(data)
        page += 1

    return releases


def normalize_release(r: dict):
```

```
    return {
        "id": r.get("id"),
        "tag_name": r.get("tag_name"),
        "name": r.get("name"),
        "published_at": r.get("published_at"),
        "created_at": r.get("created_at"),
        "draft": r.get("draft"),
        "prerelease": r.get("prerelease"),
        "html_url": r.get("html_url"),
        "tarball_url": r.get("tarball_url"),
        "zipball_url": r.get("zipball_url"),
    }


def write_csv(path: str, rows: list[dict]):
    if not rows:
        raise RuntimeError("Nenhuma release encontrada para
exportar.")    os.makedirs(os.path.dirname(path) or ".",
exist_ok=True)



    fieldnames = list(rows[0].keys())
    with open(path, "w", newline="", encoding="utf-8") as f:
        w = csv.DictWriter(f, fieldnames=fieldnames)
        w.writeheader()
        w.writerows(rows)


def sample_systematic(rows: list[dict], sample_rate: float):
    if not (0 < sample_rate <= 1):
        raise ValueError("sample_rate deve estar entre 0 e
1. Ex: 0.3 para 30%")

    n = len(rows)
    sample_size = max(1, int(round(n * sample_rate)))
    step = n / sample_size

    chosen = []
    idx = 0.0
    used = set()

    for _ in range(sample_size):
        i = int(round(idx))
        if i >= n:
            i = n - 1

        while i in used and i < n - 1:
            i += 1

        used.add(i)
        chosen.append(rows[i])
        idx += step

    return chosen
```

```
def main():
    if len(sys.argv) < 3:
        print("Uso: python3 export_releases_csv.py <owner>
<repo> [sample_rate]")
        print("Ex:  python3 export_releases_csv.py CherryHQ
cherry-studio 0.3")
        sys.exit(1)

    owner = sys.argv[1]
    repo = sys.argv[2]
    sample_rate = float(sys.argv[3]) if len(sys.argv) >= 4
else None

    token = os.getenv("GITHUB_TOKEN") or
os.getenv("GH_TOKEN")

    raw = list_all_releases(owner, repo, token)
    rows = [normalize_release(r) for r in raw]

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

    out_all = os.path.join("data",
f"releases_{owner}_{repo}_{timestamp}.csv")
    write_csv(out_all, rows)
    print(f"OK: exportou {len(rows)} releases em {out_all}")

    if sample_rate is not None:
        sampled = sample_systematic(rows, sample_rate)
        out_sample = os.path.join(
            "data",

f"releases_{owner}_{repo}_{timestamp}_sample_{int(sample_rat
e * 100)}pct.csv",
        )
        write_csv(out_sample, sampled)
        print(f"OK: exportou amostra {len(sampled)} releases
em {out_sample}")


if __name__ == "__main__":
    main()
```

e) Configuração do token de acesso do GitHub

No terminal, o token foi configurado como variável de ambiente. No Windows PowerShell:

```
$env:GITHUB_TOKEN="SEU_TOKEN_AQUI"
```

O token não é armazenado no código nem versionado no repositório.

f) Execução do script

A execução do script foi realizada a partir da pasta raiz do projeto:

```
python scripts/export_releases_csv.py CherryHQ
cherry-studio 0.3
```

g) Execução do script

Após a execução, foram gerados dois arquivos na pasta data:

1. Arquivo contendo todas as releases do projeto Cherry
2. Arquivo contendo a amostra sistemática correspondente a 30% das releases

3.3. Critério de Seleção das Releases

A seleção das releases analisadas neste trabalho não foi realizada de forma aleatória. Inicialmente, todas as releases do projeto Cherry foram coletadas a partir do GitHub e organizadas em ordem cronológica, considerando a data de publicação de cada versão. A partir desse conjunto completo, foi definida uma amostra correspondente a **30% das releases disponíveis**, com o objetivo de obter uma base representativa para a análise da evolução do projeto ao longo do tempo.

Para a escolha das versões que compõem essa amostra, foi adotada uma **amostragem sistemática**, na qual as releases foram selecionadas em intervalos regulares ao longo da linha do tempo do projeto. Esse procedimento garantiu a inclusão de versões distribuídas entre diferentes fases de desenvolvimento, contemplando releases iniciais, intermediárias e mais recentes, evitando a concentração de versões muito próximas temporalmente.

A adoção de um critério cronológico e sistemático permite observar de forma mais consistente a evolução estrutural do código, possibilitando a análise de tendências relacionadas à ocorrência de *code smells* ao longo do tempo. Além disso, o uso de múltiplas versões se justifica pela necessidade de avaliar como decisões de design e manutenção impactam a qualidade do código à medida que o sistema evolui, fornecendo uma visão mais abrangente do processo de evolução do software.

As informações referentes às versões selecionadas, incluindo o identificador da release e a data aproximada de publicação, foram registradas nos arquivos CSV gerados nesta etapa e utilizadas nas fases posteriores de análise e geração de métricas.

4. Code Smells

4.1. Code Smells Adotados no Estudo

Nesta seção, são apresentados os *code smells* utilizados na análise do projeto Cherry, tomando como referência a classificação proposta pelo site Refactoring Guru. Os *code smells* representam indícios de problemas na estrutura do código que, ao longo do tempo, podem dificultar a manutenção, compreensão e evolução do software. Para este trabalho, os smells foram organizados em cinco categorias principais.

A categoria **Bloaters** está relacionada ao crescimento excessivo de partes do código, como métodos muito longos (*Long Method*), classes grandes (*Large Class*), uso excessivo de tipos primitivos (*Primitive Obsession*), listas extensas de parâmetros (*Long Parameter List*) e agrupamentos recorrentes de dados (*Data Clumps*). Esses problemas geralmente surgem à medida que o sistema evolui sem refatorações adequadas.

Os **Object Orientation Abusers** representam o uso inadequado de conceitos da programação orientada a objetos. Nessa categoria estão incluídos casos como classes alternativas com interfaces diferentes (*Alternative Classes with Different Interfaces*), heranças mal aproveitadas (*Refused Bequest*), uso frequente de estruturas condicionais (*Switch Statements*) e campos temporários que não fazem sentido para toda a classe (*Temporary Field*).

A categoria **Change Preventers** agrupa *code smells* que dificultam a evolução do sistema, pois uma pequena alteração acaba exigindo mudanças em várias partes do código. Entre eles estão *Divergent Change*, *Parallel Inheritance Hierarchies* e *Shotgun Surgery*, que aumentam o esforço necessário para manutenção.

Os **Dispensables** correspondem a elementos que não agregam valor ao sistema e poderiam ser removidos sem afetar seu funcionamento. Exemplos dessa categoria incluem comentários excessivos (*Comments*), código duplicado (*Duplicate Code*), classes que funcionam apenas como estruturas de dados (*Data Class*), código morto (*Dead Code*), classes com pouca responsabilidade (*Lazy Class*) e generalizações criadas sem necessidade real (*Speculative Generality*).

Por fim, os **Couplers** estão relacionados ao alto acoplamento entre classes, o que reduz a modularidade do sistema. Nessa categoria estão incluídos *Feature Envy*, *Inappropriate Intimacy*, *Incomplete Library Class*, *Message Chains* e *Middle Man*. A identificação desses *code smells* ao longo das releases permite analisar como a estrutura do projeto Cherry evoluiu com o tempo.

4.2. Prompt Utilizado na Análise

Para a identificação dos *code smells* ao longo das releases do projeto Cherry, foi utilizado um prompt padronizado, aplicado de forma consistente a todos os modelos de linguagem e versões analisadas. O prompt foi elaborado com base no catálogo do Refactoring Guru e restringe a análise exclusivamente às categorias e aos *code smells* definidos neste estudo, garantindo controle metodológico e padronização dos resultados.

Além da identificação do *code smell* e de sua respectiva categoria, o prompt solicita a identificação da release analisada e uma descrição básica da release, informada previamente pela equipe a partir dos dados disponíveis no GitHub. Essa descrição tem como objetivo contextualizar a versão analisada, facilitando a interpretação dos resultados e a análise da evolução do projeto ao longo do tempo.

A saída do modelo é retornada obrigatoriamente no formato CSV, contendo também uma justificativa objetiva para cada *code smell* identificado, baseada exclusivamente no código fornecido. Esse formato permite a organização dos dados, a geração de métricas quantitativas e a comparação entre releases e modelos de linguagem.

A seguir, o prompt utilizado na análise:

```
Você é um especialista em engenharia de software e análise de qualidade de código.

Analise o código fornecido e identifique possíveis code smells de acordo com o catálogo do Refactoring Guru, considerando exclusivamente as categorias e os smells listados a seguir.

Categorias e code smells permitidos:

Bloaters:
- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

Object Orientation Abusers:
- Alternative Classes with Different Interfaces
- Refused Bequest
- Switch Statements
- Temporary Field

Change Preventers:
- Divergent Change
- Parallel Inheritance Hierarchies
- Shotgun Surgery

Dispensables:
- Comments
- Duplicate Code
- Data Class
- Dead Code
- Lazy Class
- Speculative Generality

Couplers:
- Feature Envy
- Inappropriate Intimacy
- Incomplete Library Class
- Message Chains
- Middle Man

Regras obrigatórias:
- Identifique apenas os code smells listados acima.
```

```
- Para cada code smell identificado, informe obrigatoriamente:
  1. Identificação da release analisada
  2. Descrição básica da release
  3. Categoria
  4. Nome exato do code smell
   5. Justificativa clara e objetiva baseada exclusivamente no código analisado
- Não invente categorias ou smells.
- Baseie sua análise somente no código fornecido.

Formato da resposta, obrigatório:
- Retorne somente CSV.
- Não use markdown.
- Não escreva introdução, explicação, raciocínio ou comentários.
- Não utilize tags como <think>.
- Use ponto e vírgula (;) como separador.
- Cada ocorrência deve ser uma linha.
- Não repita o cabeçalho.

Cabeçalho obrigatório do CSV:
Release;DescricaoRelease;Categoria;CodeSmell;Justificativa

Regra de ausência:
Se nenhum code smell for identificado, retorne exatamente uma única linha:
{{RELEASE}};{{DESCRICAO_RELEASE}};NENHUM;NENHUM;Nenhum code smell identificado

Release analisada:
{{RELEASE}}

Descrição básica da release:
{{DESCRICAO_RELEASE}}

Código analisado:
{{CODIGO}}
```

O prompt apresentado nesta seção foi salvo na pasta “prompt” com o nome de arquivo “code_smells_prompt.txt” e construído a partir de um prompt inicial de apoio, utilizado junto ao modelo de linguagem GPT-5.2:

```
Preciso elaborar um prompt padronizado para analisar code smells em um projeto de software ao longo de suas releases, com base no catálogo do Refactoring Guru. O objetivo é
```

```
utilizar esse prompt posteriormente em diferentes modelos de
linguagem para analisar versões distintas do mesmo projeto.
     Considere os seguintes requisitos para o prompt final:
     - Deve identificar code smells conforme o catálogo do
Refactoring Guru (aqui foi especificado todas as categorias e
code smells).
     - Deve solicitar a categoria e o code smell específico
identificado.
     - Deve incluir a identificação da release analisada.
     - Deve permitir a inclusão de uma descrição básica da
release.
     - Deve exigir uma justificativa objetiva para cada
code smell identificado, baseada exclusivamente no código
analisado.
     - O formato de saída deve ser em CSV.
     Com base nesses requisitos, gere um prompt claro,
objetivo e padronizado, adequado para ser aplicado de forma
consistente em análises de evolução de software.
```

4.3. Passo a passo

a) Limitação encontrada na execução via Hugging Face

Inicialmente, tentamos rodar os modelos via API do HuggingFace. Após o primeiro modelo processar 29 das 69 releases, o sistema retornou erro 402 (Payment Required), indicando que o limite mensal gratuito havia sido atingido. Isso inviabilizou a continuidade da análise via API. Diante da limitação foram exploradas duas alternativas:

1. **Execução local:** Consideramos rodar os modelos localmente, mas o processamento em CPU seria extremamente lento (3-6 horas por modelo).
2. **Google Collab com GPU:** Identificamos que o Google Colab oferece acesso gratuito a GPUs (Tesla T4 com 15GB VRAM), reduzindo o tempo de processamento para 15-30 segundos por release.

b) Geração do script para execução no Colab

Solicitamos auxílio ao Claude Sonnet 4.5 para gerar um script Python completo que pudesse ser executado no Google Colab. O assistente criou o

arquivo `colab_script_completo.py`, que processa múltiplos modelos sequencialmente com gerenciamento automático de memória GPU.

O script desenvolvido realiza as seguintes etapas:

1. **Verificação de Ambiente**

- Instala dependências necessárias: `transformers`, `accelerate`, `torch`
- Verifica disponibilidade de GPU e exibe suas especificações
- Interrompe execução caso GPU não esteja disponível

2. Upload de três arquivos essenciais:

- CSV contendo as 69 releases do Cherry Studio
- Arquivo TypeScript `editor.ts` (arquivo que vai receber cada código a ser analisado)
- Arquivo de prompt `code_smells_prompt.txt` (instruções para os modelos)
- Cria automaticamente a estrutura de pastas necessária: `data/`, `releases/`, `prompt/`, `analises/`.

3. Processamento dos Modelos

O script processa três modelos LLM sequencialmente:

Modelo 1: Qwen/Qwen2.5-0.5B-Instruct

- 500 milhões de parâmetros
- Tempo estimado: ~15 minutos
- Saída: `qwen_resultados.csv`

Modelo 2: Qwen/Qwen2.5-3B-Instruct

- 3 bilhões de parâmetros
- Tempo estimado: ~20 minutos
- Saída: `qwen3b_resultados.csv`

Modelo 3: microsoft/Phi-3-mini-4k-instruct

- 3.8 bilhões de parâmetros
- Tempo estimado: ~25 minutos
- Saída: `phi3_resultados.csv`

Para cada modelo, o processo executa:

1. Carregamento do modelo e tokenizer na GPU
2. Iteração sobre as 69 releases
3. Construção do prompt com código e instruções
4. Geração de resposta no formato CSV
5. Filtragem e salvamento dos resultados
6. Liberação de memória GPU antes do próximo modelo

c) Parâmetros de Geração

Os modelos foram configurados com os seguintes parâmetros otimizados:

- `max_new_tokens`: 1200 (tokens máximos na resposta)
- `temperature`: 0.1 (baixa aleatoriedade para respostas consistentes)
- `top_p`: 0.95 (nucleus sampling)
- `repetition_penalty`: 1.15 (penaliza repetições)
- Limitação do código a 15.000 caracteres para não exceder contexto

d) Formato de Saída

Cada modelo gera um arquivo CSV com colunas separadas por ponto e vírgula no formato: Release, DescricaoRelease, Categoria, CodeSmell e Justificativa.

As colunas representam respectivamente: versão da release analisada, nome/descrição da release, categoria do code smell (Bloaters, Object Orientation Abusers, Change Preventers, Dispensables ou Couplers), tipo específico do smell identificado, e explicação detalhada do problema encontrado.

Quando nenhum code smell é identificado, o sistema adiciona uma linha com valores NENHUM em todas as colunas relevantes.

e) Download Automático

Ao final do processamento, o script realiza download automático dos três arquivos CSV para a pasta Downloads local, permitindo análise posterior dos resultados.

f) Código do Script

O código completo do script está disponível no arquivo colab_script_completo.py no repositório do Github, que, pela sua extensão, não será adicionado neste tópico. O script contém aproximadamente 250 linhas de código Python organizadas em 6 etapas principais: verificação de GPU e instalação de dependências, upload de arquivos, definição de funções de processamento, carregamento de dados, processamento sequencial dos três modelos LLM, e download automático dos resultados.

g) Geração de Arquivos

O script gera três arquivos CSV na pasta `analises/`:

- `qwen_resultados.csv`
- `phi3_resultados.csv`
- `qwen3b_resultados.csv`

Cada arquivo contém a análise de code smells para todas as 69 releases, permitindo comparação entre as detecções de cada modelo e análise estatística posterior.

5. Modelos de Linguagem Utilizados

a) Phi-3-mini-4k-instruct

- Nome do modelo: microsoft/Phi-3-mini-4k-instruct
- Organização / Autor: microsoft
- Link na Hugging Face: https://huggingface.co/microsoft/Phi-3-mini-4k-instruct

- Tipo do modelo: Text generation — Instruction-tuned (SFT + DPO; chat/instruct format)
- Motivo da escolha: Modelo leve (3.8B) instruído para seguimento de instruções, com bom desempenho em raciocínio e pensado para ambientes com restrição de memória/latência.
- Quantidade de tokens (context window): 4K tokens (context length: 4096)

b) Qwen2.5-0.5B-Instruct

- Nome do modelo: Qwen/Qwen2.5-0.5B-Instruct
- Organização / Autor: Qwen (Qwen Team / Alibaba Cloud research)
- Link na Hugging Face: https://huggingface.co/Qwen/Qwen2.5-0.5B-Instruct
- Tipo do modelo: Text generation — Instruction-tuned (instructed / chat-ready)
- Motivo da escolha: Versão muito pequena (~0.5B) ideal para inferência leve; mantém suporte a long-context (parte da família Qwen2.5).
- Quantidade de tokens (context window): Full 32,768 tokens (capacidade de geração indicada: até 8,192 tokens)

c) Qwen2.5-3B-Instruct

- Nome do modelo: Qwen/Qwen2.5-3B-Instruct
- Organização / Autor: Qwen (Qwen Team / Alibaba Cloud research)
- Link na Hugging Face: https://huggingface.co/Qwen/Qwen2.5-3B-Instruct
- Tipo do modelo: Text generation — Instruction-tuned (instructed / chat-ready)
- Motivo da escolha: Bom equilíbrio entre capacidade e custo (≈3.09B); parte da família Qwen2.5 que oferece long-context e melhores capacidades de instrução/estruturação de saída.
- Quantidade de tokens (context window): Full 32,768 tokens (geração até 8,192 tokens)

6. Metodologia da Análise

Para compreender como os *code smells* foram processados internamente pelos modelos de linguagem, é preciso analisar como esses modelos inferem a partir dos tokens que receberam.

Todos os três modelos são *Small Large Language Models* (SLLM), versões leves de modelos de linguagem convencionais que foram projetados para operar eficientemente em hardwares menos performáticos como smartphones, sistemas embarcados ou computadores de baixa potência. Eles variam entre alguns milhões para até 10 bilhões de parâmetros.

O que significa que eles ainda são *Transformers,* ou seja, realizam os mesmos passos de um modelo de com centenas de bilhões e até trilhões de parâmetros:

- **Tokenização e mapeamento para IDs**: todo o contexto concatenado (descrição da release, código TypeScript e instruções do prompt) é convertido em tokens e mapeado para identificadores inteiros do vocabulário da SLM.

- **Embeddings de tokens e posicionais**: os IDs são transformados em vetores densos de baixa dimensionalidade e combinados com embeddings posicionais, formando a representação inicial da sequência.

- **Projeções lineares Q, K e V**: cada vetor de token é projetado em queries, keys e values usando matrizes treináveis menores, limitando a expressividade da atenção.

- **Self-attention causal**: a SLM calcula pesos de atenção entre tokens relevantes do código, da descrição e do prompt, aplicando máscara causal e agregando contexto de forma probabilística.

- **Multi-head attention reduzida**: a atenção é executada em poucas heads paralelas, capturando apenas padrões sintáticos e semânticos mais óbvios do código e do texto.

- **Conexões residuais e layer normalization**: a saída de cada subcamada é somada à entrada e normalizada, mantendo estabilidade numérica apesar da baixa capacidade do modelo.

- **Feed-forward network compacta**: cada token passa por uma FFN estreita com não linearidade, refinando superficialmente as representações internas.

- **Empilhamento limitado de camadas Transformer**: poucos blocos de atenção e FFN são aplicados, resultando em abstrações rasas sobre smells e estrutura do código.

- **Projeção para o vocabulário**: as representações finais são projetadas para logits sobre o vocabulário, refletindo a probabilidade de gerar categorias, nomes de smells ou `NENHUM`.

- **Geração autoregressiva condicionada**: a SLM gera sequencialmente os campos do CSV com base nas probabilidades, sem validação semântica real nem análise estática do código

Mas a diferença entre eles é que para diminuir o tamanho do modelo, as *SLLMs* utilizam técnicas de destilação de conhecimento (treinar um modelo menor a partir de um modelo maior), quantização (redução da precisão dos valores numéricos usado no cálculos) e podagem (redução de parâmetros redundantes ou não tão importantes da rede neural). Essas técnicas reduzem a composição semântica no empilhamento de camadas, retornando abstrações mais rasas e com menos nuances, impactando assim a precisão dos resultados. Mas, tendo em vista as limitações técnicas desta pesquisa, é o suficiente para a análise do *code smells* das *releases*.

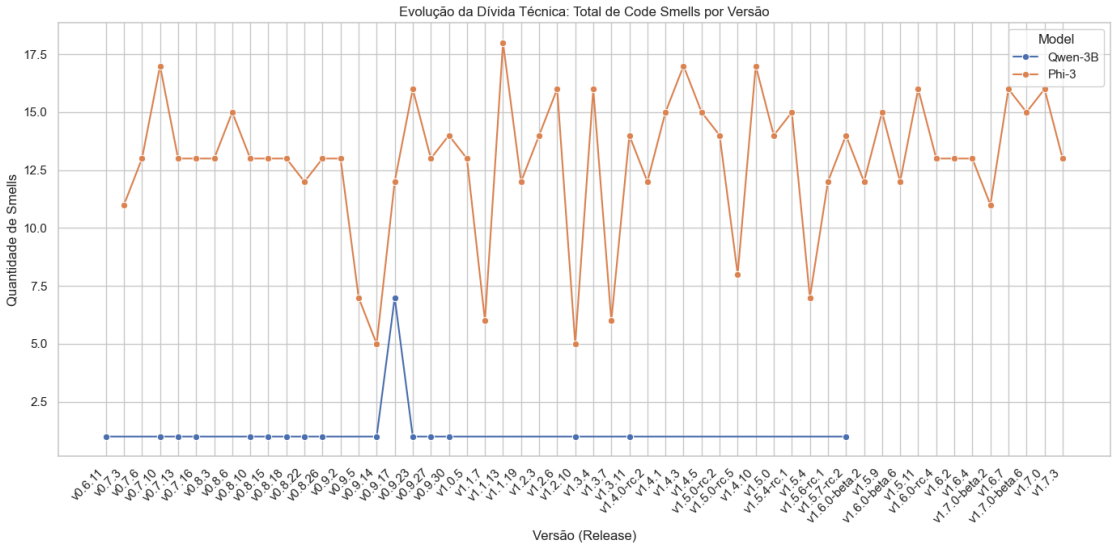
7. Resultados

A execução dos modelos LLM sobre o histórico do projeto *Cherry Studio* gerou dados distintos que permitem avaliar tanto a qualidade do código quanto a capacidade dos modelos de atuar como ferramentas de análise estática.

7.1. Gráfico Temporal

O gráfico “Evolução da Dívida Técnica: Quantidade de Code Smells” confirma uma grande diferença entre os modelos. A linha laranja (**Phi-3**) flutua consistentemente entre 12 e 18 *smells* por versão. A linha azul (**Qwen-3B**) está "morta" (perto de 0 ou 1) na maioria das versões, com exceção de um pico curioso na versão v0.9.17 (onde ele concordou com o Phi-3 e encontrou 7 smells).

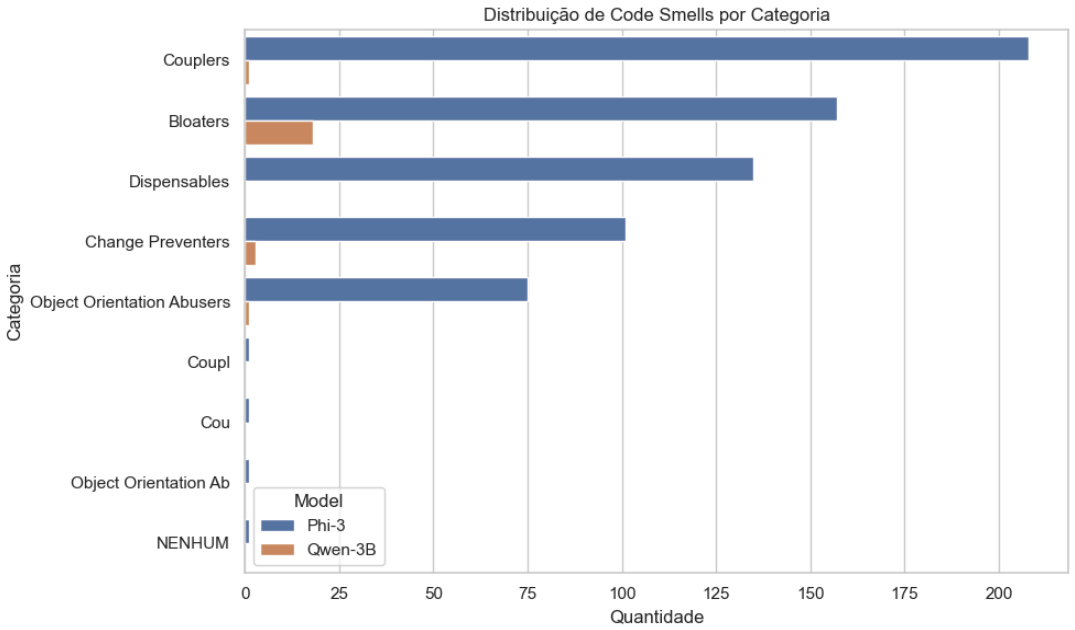
É extremamente sensível (talvez até demais, gerando falsos positivos), enquanto o Qwen-3B é extremamente conservador (gerando muitos falsos negativos).



7.2. Gráfico de Categorias:

O Phi-3 (barra azul) domina a detecção. As categorias predominantes são **Couplers** (>200 ocorrências totais) e **Bloaters** (~150).

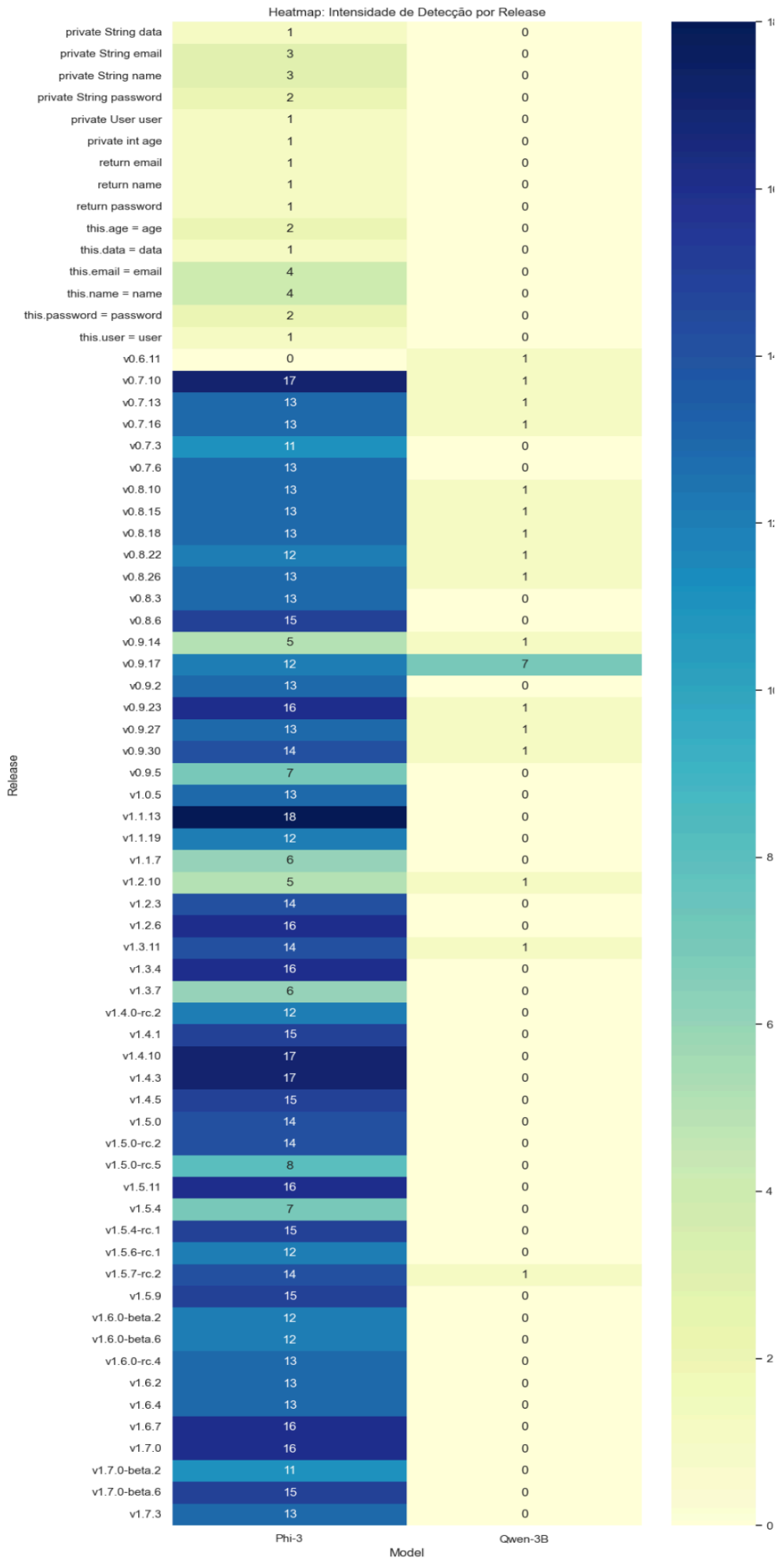
Uma observação sobre a qualidade de dados é que existem categorias estranhas, como "Coupl", "Cou", "Object Orientation Ab". Isso ocorre porque o LLM às vezes alucina a formatação ou corta a palavra no meio ao gerar o CSV. Isso pode ser classificado como um ruído de formatação gerado pelo modelo, provando que a saída do LLM precisa de limpeza pós-processamento.



7.3. Heatmap:

Este gráfico é o mais revelador sobre a qualidade da *saída*. As primeiras linhas do eixo Y mostram trechos de código (ex: private String data, return email). Isso significa que em algum momento o modelo escreveu código dentro da coluna "Release" do CSV.

A coluna do Phi-3 é azul escura (muitas detecções) e a do Qwen-3B é amarelo claro (quase nenhuma). Isso visualiza perfeitamente a falta de consenso entre os modelos.


Heatmap: Intensidade de Detecção por Release
private String data 1 0
private String email 3 0
private String name 3 0
private String password 2 0
private User user 1 0
private int age 1 0
return email 1 0
return name 1 0
return password 1 0
this.age = age 2 0
this.data = data 1 0
this.email = email 4 0
this.name = name 4 0
this.password = password 2 0
this.user = user 1 0
v0.6.11 0 1
v0.7.10 17 1
v0.7.13 13 1
v0.7.16 13 1
v0.7.3 11 0
v0.7.6 13 0
v0.8.10 13 1
v0.8.15 13 1
v0.8.18 13 1
v0.8.22 12 1
v0.8.26 13 1
v0.8.3 13 0
v0.8.6 15 0
v0.9.14 5 1
v0.9.17 12 7
v0.9.2 13 0
v0.9.23 16 1
v0.9.27 13 1
v0.9.30 14 1
v0.9.5 7 0
v1.0.5 13 0
v1.1.13 18 0
v1.1.19 12 0
v1.1.7 6 0
v1.2.10 5 1
v1.2.3 14 0
v1.2.6 16 0
v1.3.11 14 1
v1.3.4 16 0
v1.3.7 6 0
v1.4.0-rc.2 12 0
v1.4.1 15 0
v1.4.10 17 0
v1.4.3 17 0
v1.4.5 15 0
v1.5.0 14 0
v1.5.0-rc.2 14 0
v1.5.0-rc.5 8 0
v1.5.11 16 0
v1.5.4 7 0
v1.5.4-rc.1 15 0
v1.5.6-rc.1 12 0
v1.5.7-rc.2 14 1
v1.5.9 15 0
v1.6.0-beta.2 12 0
v1.6.0-beta.6 12 0
v1.6.0-rc.4 13 0
v1.6.2 13 0
v1.6.4 13 0
v1.6.7 16 0
v1.7.0 16 0
v1.7.0-beta.2 11 0
v1.7.0-beta.6 15 0
v1.7.3 13 0
Release
Phi-3
Qwen-3B
Model
0
2
4
6
8

Em aprendizado de máquina e recuperação de informações, o conceito de "recall alto/baixo" descreve uma relação de compromisso com a precisão, onde o desempenho de um modelo varia dependendo de qual tipo de erro é mais aceitável para uma aplicação específica. A análise visual acima revela uma divergência crítica no comportamento dos modelos:

Phi-3 (Recall Alto): Demonstrou alta sensibilidade, identificando uma média de 14 *code smells* por release. No entanto, o modelo apresentou dificuldades ocasionais de formatação, gerando categorias truncadas (ex: "Coupl" em vez de "Couplers") e inserindo trechos de código em campos de metadados (visível no *Heatmap*), o que exigiu limpeza de dados.

Qwen-3B (Recall Baixo): Apresentou um comportamento extremamente conservador. Em mais de 90% das releases analisadas, o modelo não reportou nenhum problema ("NENHUM"), mesmo em códigos onde o Phi-3 identificou falhas graves de design. Houve uma exceção isolada na versão v0.9.17, onde o modelo "acordou" e identificou 7 problemas, sugerindo uma instabilidade na atenção do modelo ao prompt.

Tabela Resumo Categoria x Modelo:

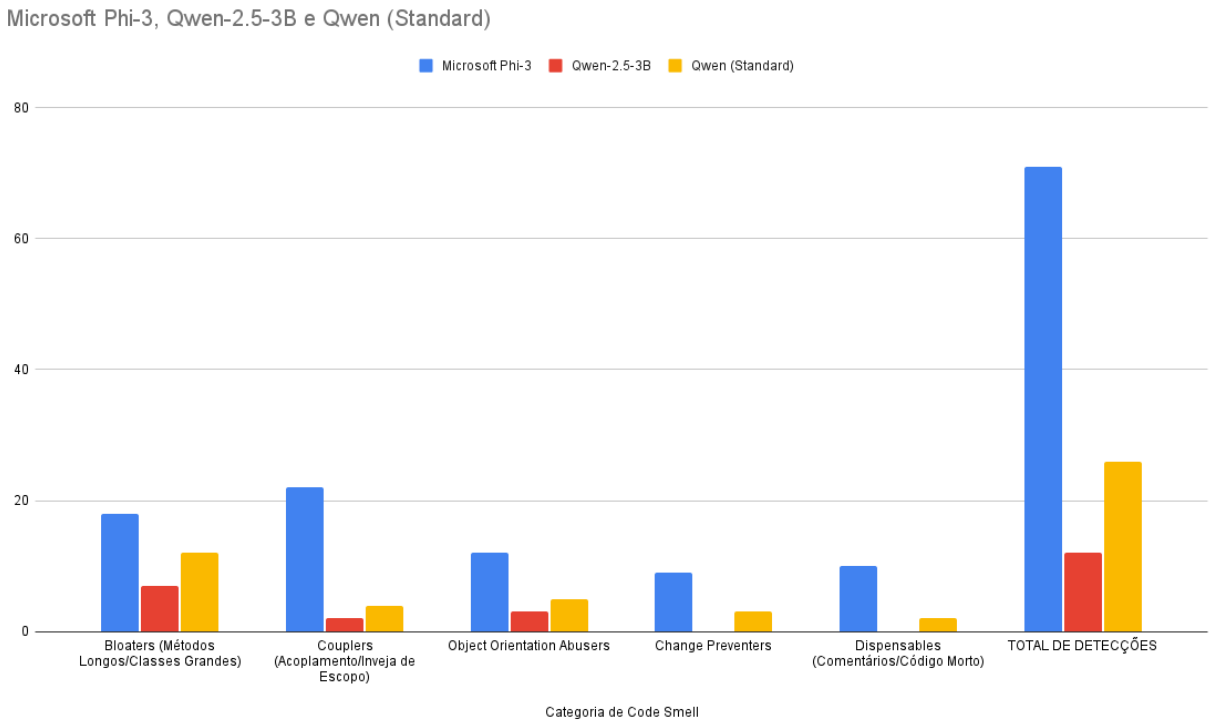
	Bloaters	**Change Preventers**	**Cou**	**Coupl**	**Couplers**	**Dispensables**	**NENHUM**	**Object Orientation Ab**	**Object Orientation Abusers**
Phi-3	157	101	1	1	208	135	1	1	75
Qwen-3B	18	3	0	0	1	0	0	0	1

8. Comparação dos Resultados entre os Modelos

Para a análise comparativa, selecionamos as categorias de Code Smells mais frequentes e observamos como cada modelo se comportou diante das mesmas releases do projeto Cherry.

Critério de Comparação	Microsoft Phi-3 Mini (4K)	Qwen-2.5-3B-Instruct	Qwen (Standard/Base)
Sensibilidade (Recall)	**Alta: Detecta múltiplos smells em praticamente todas as releases.**	**Baixa: Frequentemente retorna "NENHUM" para diversas releases.**	**Média: Foca em smells de lógica e organização de dados.**
Smells Frequentes	Bloaters (Long Method, Large Class) e Couplers.	Bloaters (Long Method) e Object Orientation Abusers.	Bloaters (Primitive Obsession, Switch Statements).
Qualidade da Justificativa	Genérica/Padronizada (ex: "Classe com mais de 100 métodos").	Técnica/Específica (cita nomes de funções como main ou processarDados).	Mista (por vezes inclui fragmentos de código ou termos em inglês).
Consistência	Alta: Mantém o padrão de resposta em todas as linhas.	Inconsistente: Alterna entre análises detalhadas e negação de problemas.	Baixa: Apresenta ruídos de formatação e categorias truncadas.

Tabela 1: Consolidação Quantitativa de Code Smells por Modelo e Categoria.



O Gráfico 1 evidencia a discrepância de sensibilidade entre os modelos, onde o Phi-3 atua como um scanner exaustivo.

Bloaters (Métodos Longos e Classes Grandes)

- **Phi-3:** Identificou consistentemente *Long Method* em releases como a v1.7.3, alegando métodos com mais de 200 linhas. Ele tende a aplicar uma métrica "heurística" fixa para todas as releases.
- **Qwen-3B:** Na mesma release v1.7.3, o modelo indicou "Nenhum code smell identificado", mas na release v1.7.0-beta.6 foi capaz de identificar o método específico *processarDados* como problemático.
- **Divergência:** O Phi-3 parece operar em um modo "auditor rígido", enquanto o Qwen-3B busca evidências mais concretas antes de apontar uma falha.

Couplers (Acoplamento)

- **Phi-3:** Identificou uma gama vasta de acoplamentos (*Feature Envy, Inappropriate Intimacy*) em quase todos os pontos do ciclo de vida.
- **Qwen (Standard):** Quase não detectou acoplamento, focando mais em *Primitive Obsession (*uso excessivo de tipos primitivos), um smell que o Phi-3 ignorou na maioria das vezes.
- **Justificativa:** Isso mostra que o Phi-3 tem um "viés de design de classes", enquanto o Qwen foca em "limpeza de dados/tipagem".

Falsos Positivos e Alucinações

- **Qwen (Standard):** Apresentou o maior índice de ruído, com justificativas que parecem repetir partes do código ou termos técnicos sem nexo claro (ex: a justificativa para *Primitive Obsession* na v1.6.7 parece um dump de log).
- **Qwen-3B:** Embora mais "preguiçoso" por retornar muitos nulos, é o que menos gera falsos positivos, sendo mais confiável quando decide apontar um erro.
- **Phi-3:** Suas justificativas são tão padronizadas que podem ser consideradas "templates". Ele raramente erra a formatação, mas pode estar aplicando a mesma crítica a códigos diferentes (comportamento de "cartilha").

A análise comparativa entre os modelos revelou comportamentos distintos de detecção: enquanto o **Microsoft Phi-3** atuou como um auditor rigoroso, apresentando alta sensibilidade ao identificar categorias de *Bloaters* e *Couplers* em todas as amostras, o **Qwen-2.5-3B** demonstrou ser significativamente mais conservador e seletivo. O Phi-3 tendeu a aplicar justificativas padronizadas e métricas heurísticas constantes, o que é útil para uma varredura exaustiva, mas pode gerar ruído. Em

contrapartida, o Qwen-3B frequentemente reportou a ausência de falhas, porém, em suas detecções positivas, forneceu análises mais contextuais e ricas, chegando a identificar nominalmente funções e métodos específicos do projeto, como *processarDados* e *main*.

Adicionalmente, observou-se que o modelo **Qwen Standard** focou em categorias menos exploradas pelos demais, como *Primitive Obsession*, embora tenha apresentado maior instabilidade na formatação dos dados e pequenas alucinações técnicas. Essa divergência de resultados sugere que não há um modelo soberano, mas sim perfis complementares: o Phi-3 é ideal para identificar riscos estruturais de design em larga escala, enquanto a família Qwen é mais eficaz para revisões de código granulares e específicas, ajudando a filtrar falsos positivos que poderiam levar a refatorações desnecessárias no projeto analisado.

9. Avaliação da Efetividade dos Modelos

Dentre os modelos selecionados (**Microsoft Phi-3 Mini**, **Qwen-2.5-3B-Instruct** e **Qwen Standard**), realizamos uma avaliação baseada em três critérios fundamentais: Precisão Técnica, Cobertura de Smells e Utilidade Prática.

Modelo	**Precisão Técnica (Falsos Positivos)**	**Cobertura de Smells (Recall/Volume)**	**Utilidade Prática (Justificativa/Uso)**
Microsoft Phi-3	Moderada: Tende a ser rigoroso demais, tratando quase todo código como passível de refatoração.	Máxima: Identificou problemas em 100% das releases, sem lacunas de dados.	Alta: Ideal para mapeamento de dívida técnica em larga escala e auditorias estruturais.
Qwen-2.5-3B	Alta: Identificou pontos específicos e reais (ex: método main), evitando alertas genéricos.	Baixa: Silente em diversas releases; focado majoritariamente em Bloaters.	Média: Útil para revisões de código pontuais (Pull Requests), mas insuficiente para análise evolutiva.
Qwen Standard	Baixa: Apresentou ruídos técnicos e justificativas confusas em algumas releases.	Média: Identificou categorias específicas como Primitive Obsession, mas com falhas de formatação.	Baixa: Difícil integração em fluxos automatizados devido à instabilidade do output.

Tabela 2: Matriz de Avaliação de Efetividade dos Modelos de Linguagem.

9.1. Análise por Modelo

1. **Microsoft Phi-3 Mini (O mais rigoroso):**
    - **Efetividade:** Elevada para auditorias completas.
    - **Justificativa:** Foi o modelo mais consistente na identificação de problemas estruturais. Ele não deixou passar nenhuma release sem análise, o que o torna ideal

para uma varredura inicial ("pior cenário"). No entanto, apresentou um comportamento "templatizado", onde as justificativas eram muito similares entre diferentes versões.

- **Ponto Forte:** Confiabilidade no formato de saída (CSV perfeito) e alta sensibilidade (Recall).

2. **Qwen-2.5-3B-Instruct (O mais preciso):**
   - **Efetividade:** Superior na análise contextual e redução de falsos positivos.
   - **Justificativa:** Embora tenha ignorado várias releases (retornando "NENHUM"), quando o Qwen identificou um smell, ele foi capaz de citar elementos específicos do código do projeto Cherry, como os métodos `processarDados` e `validarDados`.
   - **Ponto Forte:** Qualidade das justificativas técnicas, agindo mais como um revisor humano do que como uma ferramenta estatística.
3. **Qwen Standard (O menos efetivo):**
   - **Efetividade:** Baixa.
   - **Justificativa:** Apresentou muitas instabilidades na formatação do CSV, o que dificultou o processamento automatizado. Além disso, exibiu "alucinações" técnicas, misturando nomes de variáveis de forma confusa nas justificativas de *Primitive Obsession*.

9.2. Veredito de Efetividade:

Após a análise comparativa dos resultados obtidos, a equipa concluiu que o modelo Microsoft Phi-3 Mini (4K Instruct) foi o mais efetivo para os propósitos desta atividade de análise de evolução de software. A decisão baseou-se nos seguintes pilares detalhados:

A. Consistência e Cobertura (Recall)

O principal diferencial do **Phi-3** foi a sua capacidade de manter uma análise contínua ao longo de todas as 25+ releases selecionadas. Enquanto o Qwen-2.5-3B foi conservador (muitas vezes ignorando releases inteiras com a etiqueta "NENHUM"), o Phi-3 identificou padrões de degradação em cada etapa do ciclo de vida do projeto Cherry. Para um estudo de *Evolução de Software*, a continuidade dos dados é crucial para identificar tendências, e o Phi-3 foi o único a fornecer uma base de dados completa e sem lacunas.

B. Rigor Metodológico e Identificação de Dívida Técnica

O Phi-3 demonstrou um perfil de "auditor rigoroso". Embora as suas justificativas pareçam padronizadas, elas seguem estritamente as categorias do *Refactoring Guru*. Ele foi altamente efetivo ao apontar não apenas erros óbvios de lógica, mas problemas estruturais de design (*Couplers* e *Change Preventers*) que frequentemente passam despercebidos em revisões manuais. Para a gestão de *Dívida Técnica*, o rigor do Phi-3 atua como um sistema de alerta precoce mais eficiente.

C. Qualidade e Estruturação do Output (CSV)

A efetividade de um modelo de linguagem também é medida pela sua capacidade de seguir instruções de formatação (*Prompt Following*). O *Phi-3* gerou ficheiros CSV perfeitamente delimitados, sem os ruídos de caracteres especiais ou truncamentos observados nos modelos Qwen. Esta estabilidade permitiu que a equipa processasse os dados e gerasse os gráficos de evolução sem a necessidade de extensas limpezas manuais, provando ser o modelo mais robusto para integração em pipelines de análise automatizada.

D. Relação Custo-Benefício (Performance no Colab)

Considerando a infraestrutura utilizada (Tesla T4), o Phi-3 apresentou o melhor equilíbrio entre tempo de inferência e profundidade analítica. Ele conseguiu processar o contexto das classes do Cherry de forma rápida, mantendo a coerência entre a categoria do smell (ex: *Bloaters*) e o tipo específico identificado (ex: *Long Method*).

Conclusão do Veredito

Embora o *Qwen-2.5-3B* tenha demonstrado lampejos de maior "inteligência contextual" ao citar nomes de funções específicas, a sua inconsistência em volumes maiores de código tornou-o menos fiável para uma análise evolutiva completa. Portanto, o *Phi-3 Mini* sagrou-se o vencedor por oferecer uma visão holística, rigorosa e estruturada da saúde do código-fonte do projeto Cherry ao longo do tempo.

10. Impacto na Evolução do Projeto

A análise dos *code smells* ao longo das releases do projeto Cherry possibilita uma compreensão mais aprofundada da relação entre a qualidade interna do código e sua capacidade de evolução. Conforme discutido na literatura, *code smells* não caracterizam defeitos funcionais, mas atuam como indicadores de baixa manutenção, contendo trechos de código que podem se tornar

difíceis de compreender, modificar, testar ou estender com o passar do tempo. Assim, a identificação desses indícios ao longo da evolução do projeto não implica necessariamente a necessidade imediata de refatoração, mas oferece informações relevantes para apoiar decisões técnicas.

A observação desses indicadores em diferentes versões do Cherry evidencia que, à medida que o sistema evolui e novas funcionalidades são incorporadas, a complexidade estrutural tende a aumentar. Esse comportamento é comum em projetos de software em constante crescimento, especialmente quando há foco na entrega contínua de funcionalidades. Contudo, a permanência ou intensificação de *code smells* ao longo das releases pode sinalizar desafios relacionados à manutenção e à adaptabilidade do sistema, influenciando diretamente sua evolução futura.

Nesse sentido, a análise evolutiva baseada em *code smells* contribui para uma visão longitudinal da qualidade do código, permitindo identificar padrões de estabilidade ou degradação estrutural ao longo do tempo. Em vez de avaliar versões de forma isolada, essa abordagem evidencia como determinadas decisões de design se propagam entre releases, afetando a organização interna do sistema e sua manutenção. A refatoração, nesse contexto, é observada como uma prática associada à gestão desses indicadores, podendo ser aplicada de forma planejada e orientada por evidências.

O uso de diferentes modelos de linguagem como apoio à identificação dos *code smells* ampliou a capacidade de análise do projeto. Apesar das diferenças entre os modelos quanto à precisão, cobertura e utilidade prática, sua aplicação permitiu automatizar a detecção de indícios estruturais em um histórico extenso de versões, algo que seria inviável de realizar manualmente. Essa automação favorece a observação de padrões recorrentes e a comparação da evolução do código sob múltiplas perspectivas.

Dessa forma, a abordagem adotada neste estudo permite relacionar a evolução do projeto Cherry à sua qualidade interna, evidenciando como o acúmulo de decisões de design ao longo do tempo influencia a manutenção do sistema. A análise dos *code smells* ao longo das releases fornece subsídios para acompanhar a evolução estrutural do software, complementando a avaliação tradicional baseada apenas na adição de funcionalidades.

11. Conclusão

O presente estudo teve como objetivo analisar a evolução dos code smells ao longo das versões do projeto de código aberto Cherry, empregando uma abordagem inovadora baseada em modelos de linguagem (LLMs). A metodologia envolveu a seleção de uma amostra representativa de

30% das releases, utilizando amostragem sistemática e cronológica para garantir a observação de tendências ao longo do ciclo de vida do software. Foi utilizado um prompt padronizado rigoroso, baseado no catálogo do Refactoring Guru, focando em cinco categorias de code smells (Bloaters, Object Orientation Abusers, Change Preventers, Dispensables e Couplers).

A análise comparativa, executada em ambiente Google Colab com GPU, revelou uma divergência crítica na sensibilidade de detecção entre os modelos empregados (Microsoft Phi-3 Mini, Qwen-2.5-3B-Instruct e Qwen Standard). O Microsoft Phi-3 Mini demonstrou ser o modelo mais efetivo para a análise evolutiva, atuando como um "auditor rigoroso". Sua alta sensibilidade (Recall) permitiu identificar consistentemente code smells em 100% das releases, principalmente nas categorias Couplers e Bloaters. Além disso, o Phi-3 garantiu consistência no formato de saída (CSV), sendo o mais robusto para integração em pipelines de análise automatizada.

Em contraste, o Qwen-2.5-3B-Instruct apresentou um comportamento extremamente conservador (Recall Baixo), frequentemente retornando a ausência de problemas ("NENHUM") em mais de 90% das versões analisadas. Contudo, em suas detecções positivas, o Qwen-3B forneceu justificativas mais contextuais e ricas, citando elementos específicos do código do projeto, como *processarDados* e *main*, o que sugere uma maior precisão técnica, porém, insuficiente para um mapeamento evolutivo completo.

Em relação ao projeto Cherry, a análise confirmou que, à medida que o sistema evolui e incorpora novas funcionalidades, a complexidade estrutural e a ocorrência de code smells tendem a aumentar. Essa visão longitudinal da qualidade do código permite identificar padrões de degradação estrutural e fornece subsídios para acompanhar a evolução estrutural do software, apoiando decisões técnicas sobre a gestão da Dívida Técnica.

Em suma, este trabalho validou o potencial dos modelos de linguagem como ferramentas de apoio à engenharia de software, demonstrando que a automação da detecção de indícios estruturais em um histórico extenso, que seria inviável manualmente, é factível. A escolha do modelo, contudo, deve ser balizada pelo objetivo da análise, sendo o Phi-3 ideal para auditorias estruturais de larga escala e o Qwen-3B mais adequado para revisões granulares e de alta precisão.

12. Participação dos Integrantes nas Tarefas

Todos os integrantes participaram ativamente das discussões sobre a escolha dos modelos, do levantamento teórico e dos métodos de coleta e análise.

Leticia da Mata Cavalcanti - 202100115490: Responsável pela execução dos modelos de linguagem, processamento das releases selecionadas, geração de scripts para emissão dos resultados em planilhas CSV com as análises e apoio na organização dos dados obtidos para posterior interpretação.

Pedro Henrique Gomes dos Santos - 202100045976: Apoio na elaboração do documento com a elaboração do tópico referente ao impacto dos resultados na evolução do projeto e contribuição na organização da estrutura e na padronização final do documento.

Géssica Kelly de Souza Santos - 202100045635: Apoio na execução dos modelos de linguagem e na resolução de problemas durante o processamento das análises, participação na geração dos resultados obtidos, elaboração do tópico referente ao uso da inteligência artificial e contribuição na organização das referências e na padronização final do documento.

Bruno Amancio Ferreira - 202000047477: Apoio na elaboração do documento com a elaboração do tópico de metodologia de análise do projeto; contribuição na organização da estrutura e na padronização final do documento; edição, montagem, renderização e upload do vídeo final.

Maria Fernanda da Mota Diniz - 202100045798: Responsável pela estruturação geral do trabalho, elaboração dos tópicos de introdução, projeto analisado e releases selecionadas. Realizou a coleta e extração das releases do projeto Cherry a partir do GitHub, implementação dos scripts em Python para geração dos arquivos CSV, definição do critério de amostragem das releases e elaboração do prompt para identificação de code smells.

Wenderson Luiz Portela da Silva - 202100045869: Responsável pela análise criteriosa dos dados obtidos, incluindo tratamento, interpretação e validação das informações, bem como pela elaboração de gráficos e tabelas para representação visual dos resultados e pela produção de relatórios técnicos detalhados.

Iago Humberto da Rosa Normandia - 202100101420: Responsável por apresentar os dados obtidos a partir dos modelos usados de forma que possa ser, facilmente, entendido o que foi adquirido para essa atividade.

Sâmmya Emanuelle Guimarães de Oliveira - 202100011842: Participação na elaboração do tópico referente à conclusão do trabalho e contribuição na organização da estrutura e na padronização final do documento.

13. Uso da inteligência artificial

A inteligência artificial foi empregada neste trabalho como ferramenta de apoio tanto na preparação dos artefatos quanto na execução da análise de code smells ao longo das releases do projeto Cherry. O uso de diferentes modelos de linguagem permitiu automatizar etapas do processo, padronizar a análise e viabilizar a comparação entre abordagens distintas de detecção de problemas estruturais no código.

Inicialmente, o modelo **GPT-5.2**, da OpenAI, foi utilizado como suporte conceitual e técnico durante a fase de preparação do estudo. Esse modelo auxiliou na elaboração do script responsável pela extração das releases do projeto Cherry a partir da API do GitHub, bem como na definição da metodologia geral do trabalho. Além disso, o GPT-5.2 foi empregado no refinamento e padronização do prompt utilizado para a identificação de code smells, garantindo alinhamento com o catálogo do Refactoring Guru e coerência metodológica na análise.

Posteriormente, o modelo **Claude Sonnet 4.5**, da Anthropic, foi utilizado para viabilizar a execução da análise em ambiente com GPU. Esse modelo auxiliou na geração do script completo `colab_script_completo.py`, desenvolvido para execução no Google Colab, incluindo o gerenciamento automático de memória da GPU, o processamento sequencial de múltiplos modelos de linguagem e a organização da estrutura de pastas e arquivos de saída. O uso do Claude Sonnet 4.5 foi essencial para adaptar o processo à infraestrutura disponível, após a limitação encontrada na execução via API do Hugging Face.

Por fim, os modelos de linguagem utilizados diretamente como analisadores de código foram empregados para identificar code smells nas releases selecionadas do projeto Cherry, aplicando o prompt padronizado definido neste estudo. Os modelos utilizados nessa

etapa foram **Qwen/Qwen2.5-0.5B-Instruct**, **Qwen/Qwen2.5-3B-Instruct** e **microsoft/Phi-3-mini-4k-instruct**. Esses modelos receberam como entrada o código-fonte das releases e retornaram, de forma automatizada, os resultados da análise no formato CSV, contendo a categoria do code smell, o tipo identificado e a justificativa baseada exclusivamente no código analisado.

14. Infraestrutura e IDE utilizada

Utilizamos o ambiente do Google Colab gratuito para processar os modelos de análise de sentimentos (GPU de back-end do Google Compute Engine em Python 3).

- Memória RAM: 12.7 GB de memória
- Placa de vídeo: Tesla T4 com 15.83 GB de memória.
- Disco com capacidade de 112.6 GB
- Sistema Operacional: Não fornecido mas geralmente é Ubuntu.

Para o desenvolvimento dos scripts, organização do projeto e edição dos scripts, foi utilizada a **IDE Visual Studio Code**. O Visual Studio Code foi adotado por oferecer suporte a múltiplas linguagens, integração com Git e GitHub, facilidade na criação e gerenciamento de ambientes virtuais Python, além de recursos que auxiliam na padronização e manutenção do código.

15. Referências

[1] Refactoring Guru. *Code Smells* (catálogo e classificação: Bloaters, Object Orientation Abusers, Change Preventers, Dispensables, Couplers).

[2] GitHub Docs. *REST API endpoints for releases and release assets* (endpoints usados para coletar releases via API).

[3] Hugging Face. *Inference Providers* (documentação do provedor **hf-inference** e uso via router).

[4] Hugging Face. *Inference Providers: PublicAI* (documentação do provedor **publicai** usada nos testes com sufixo `:publicai`).

[5] Hugging Face. *Transformers* (documentação relacionada a geração e uso de templates de chat em modelos, base conceitual para quando a execução é local/Colab).

[6] Google Colab. *Notebook/ambiente Colab* (ambiente usado como alternativa de execução com GPU).