UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

ADICLEY DE OLIVEIRA COSTA - 202100104262
ALÍRIA DE SANTANA DE AMORIM CRUZ - 201900050813
HERBERT BARRETO FREIRE - 202200123977
IAGO SOARES DE MACEDO - 202100045644
LETÍCIA CAROLINE DA SILVA OLIVEIRA - 202000047655
LUCAS ARAGÃO DAMACENO - 202100045760
RODRIGO NUNES DE SANTANA - 202000047780

EVOLUÇÃO DE SOFTWARE

ATIVIDADE 2 - Refactoring e Code Smells

SÃO CRISTÓVÃO – SE
2025

Os artefatos necessários para a realização desta atividade, bem como o vídeo tutorial dela, estão disponíveis no seguinte repositório do GitHub Evolucao_Software_2025-2_anything-llm_atividade2.

Link do vídeo:

https://drive.google.com/file/d/1J4SUL8_2iWwux9Xcairc9By-eIWfExpp/view?usp=sharing

Contribuições

Aluno	**Matrícula**	**Contribuição**
Adicley de Oliveira Costa	202100104262	Análise do código-fonte das releases do projeto e identificação das evidências de code smells
Alíria de Santana de Amorim Cruz	201900050813	Comparação dos resultados da análise obtidos entre os modelos para cada um dos code smells identificados
Herbert Barreto Freire	202200123977	Análise do impacto ao longo do tempo
Iago Soares de Macedo	202100045644	Análise do impacto ao longo do tempo
Letícia Caroline da Silva Oliveira	202000047655	Avaliação dos modelos mais efetivos
Lucas Aragão Damaceno	202100045760	Resultados da análise com as respectivas justificativas para a existência dos code smells.
Rodrigo Nunes de Santana	202000047780	Avaliação dos modelos mais efetivos
Leonardo Lima Araujo	201900125781	Não entrou em contato / não contribuiu

Requisitos de Hardware

Para realizar a atividade, o projeto foi executado em um ambiente de nuvem do Google Colab com uma GPU T4 de 16GB (15GB de VRAM disponível), 12.77GB de RAM do sistema, e disco com capacidade de 112GB.

Introdução e Análise do código-fonte do projeto

Para realização da atividade 2, foi extraído o código-fonte do projeto escolhido (no caso da equipe, o anything-llm) com o intuito de analisar os artefatos de código para detectar possíveis presenças de code smells ao longo das releases no projeto utilizando 3 diferentes modelos de linguagens disponibilizados na plataforma Hugging Face.

Com isso, tendo em vista as limitações de hardware disponibilizados no plano free do Google Colab, foram escolhido 3 (três) LLM's do tipo text-generation (entre 1B e 4B de parâmetros) para detecção de code smells nos artefatos de código, onde, para isso, com o intuito de obter uma análise mais precisa da presença de code smells nos artefatos, foi definido o seguinte prompt pela equipe:

```
prompt = """You are code analysis assistant.
            Identify if the source code has any of the following code smells: {}
            Code: {}
            Provide your response in the exact format: "Smell: <name>".
            Do not add explanations.
            Response:"""
```

O prompt acima contém, no primeiro campo da string, a seguinte lista de possíveis code smells listados no site: [Refactoring Guru]

```
code_smells = [
    "Long Method", "Large Class", "Primitive Obsession", "Long Parameter List", "Data Clumps", "Switch Statements",
    "Divergent Change", "Parallel Inheritance Hierarchies", "Shotgun Surgery", "Duplicate Code", "Dead Code",
    "Speculative Generality", "Lazy Class", "Inappropriate Intimacy", "Incomplete Library Class", "Message Chains"
    "Feature Envy"
]
```

Onde, no segundo campo do prompt, será passado o código-fonte alvo para análise.

Como dito anteriormente, tendo em vista as limitações de hardware para a atividade, a equipe optou por analisar os artefatos do código-fonte manualmente e, após isso, selecionar os módulos do projeto que contenham artefatos de código com possíveis evidências de code smells. A partir da análise feita pela equipe, notamos que os módulos que compõe o back-end do projeto (como os módulos endpoints, api e src(onde este é ligado ao front-end)), totalizando 15 artefatos analisados de 1070 do projeto, possuem alguns métodos longos (alguns com mais de 1000 linhas de código em um único método), uma evidência forte de presença do Long Method,

ou seja, um método único que contém, internamente, muitas funcionalidades embutidas em sua lógica.

Um outro possível code smell identificado pela equipe foi o “duplicate code”, onde foi encontrado artefatos que continha, em sua lógica, trechos de código que poderiam ser refatorados (por meio do Extract Method) para sua reutilização, ao invés de replicar inúmeras vezes o mesmo código com poucas ou nenhuma mudança, além da existência de muitos trechos de códigos comentados nos artefatos, ou seja a presença de código não utilizado (Unreachable Code) no projeto, evidenciando, assim, um possível não tratamento dessas problemáticas.

Tendo isso em vista, os artefatos de código foram passados para a lista de modelos selecionados onde, ao final, foi produzido um arquivo de saída contendo a identificação do(s) code smells encontrado(s) pelos modelos de acordo com o prompt montado.

Ainda sobre a análise dos artefatos de código, percebeu-se, por meio das PR’s criadas ao longo das releases do projeto, que as PR possuem um template com o campo identificando o seu tipo, a exemplo do “Refactor” para indicar a presença de uma PR de refatoração de código. Nesse sentido, notamos que algumas PR ligadas a esse tipo não eram, em alguns casos, vinculados a correção dos code smells, por meio da refatoração por exemplo, mas sim, a atualizações de funcionalidades do código.

Nesse sentido, a equipe não conseguiu identificar a presença de mecanismos automatizados no projeto para uma possível detecção e correção de code smells, mas sim algumas correções manuais feitas por meio de PR’s do tipo refatoração.

1 - Escolha dos modelos

Para a etapa 1, foram selecionados os seguintes modelos de linguagem do tipo text-generator:

```
models = [
    "microsoft/Phi-3-mini-128k-instruct",
    "microsoft/Phi-4-mini-instruct",
    "deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B"
]
```

1) Modelo: microsoft/Phi-3-mini-128k-instruct

O primeiro modelo escolhido foi o “microsoft/Phi-3-mini-128k-instruct”, um modelo do tipo “text-generator” com 4B de parâmetros e janela de contexto de comprimento 128K de tokens que foi treinado utilizando o dataset Phi-3, um dataset que, além dos dados brutos, foram adicionados variações desses dados (dados sintéticos) com o objetivo de aumentar sua capacidade de raciocínio após o treinamento do modelo.

2) Modelo: microsoft/Phi-4-mini-instruct

O segundo modelo escolhido foi o “microsoft/Phi-4-mini-instruct”, um outro modelo do tipo “text-generator” com 4B de parâmetros com janela de contexto de 128K de tokens que foi treinado utilizando dados sintético e dados públicos filtrados, com foco em dados denso com o intuito de aprimorar o raciocínio do modelo após o treinamento.

3) Modelo: deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B

Por fim, o terceiro modelo de geração de texto escolhido foi o “deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B”, um modelo com 2B de parâmetros e janela de contexto de 128K de tokens, o qual, diferente dos demais, foi treinado utilizando a técnica de Aprendizado por Reforço, uma técnica de Aprendizado de Máquina que consiste em “ensinar” o modelo com base nos dados observados, para após isso, realizar alguma tomada de decisão. Ou seja, diferente dos modelos listados anteriormente, que foram treinados com dados rotulados (Aprendizado supervisionado), o DeepSeek-R1 foi o único modelo escolhido que foi treinado utilizando Reinforcement Learning sem ajuste fino supervisionado.

2 - Nesta etapa, submetemos 15 artefatos de código (distribuídos entre os módulos `api`, `endpoint` e `src`) à análise dos três modelos selecionados. A diversidade e o tamanho dos arquivos revelaram comportamentos distintos de "raciocínio" e aderência ao prompt em cada LLM.

2.1. Análise do Modelo: Microsoft Phi-3 Mini (128k Instruct)

- **Resultados Observados:** O modelo apresentou um comportamento instável. Em arquivos menores (como `workspace_index.js`), ele foi assertivo ao identificar apenas o *Long Method*. No entanto, em arquivos de maior complexidade (como `admin_index.js`), o modelo sofreu de "alucinação repetitiva", gerando uma lista infinita e redundante de smells (repetindo *Long Method*, *Switch Statements*, *Inappropriate Intimacy* dezenas de vezes na mesma resposta).
- **Justificativa Técnica:**
    - **Long Method:** O modelo identificou corretamente a principal característica do código legado do projeto: funções massivas que acumulam responsabilidades de validação, controle de acesso e regra de negócio (como visto em `apiWorkspaceEndpoints`).
    - **Falha de Contexto:** A repetição cíclica de smells no `admin_index.js` indica que o modelo "se perdeu" na geração dos tokens ao tentar processar o contexto longo do arquivo, entrando em

um loop de geração de texto, um problema comum em modelos menores (4B parâmetros) quando saturados com prompts complexos e códigos extensos.

2.2. Análise do Modelo: Microsoft Phi-4 Mini (Instruct)

- **Resultados Observados:** Este foi o modelo mais "sensível" (alto *Recall*, baixa *Precision*). Ele ignorou consistentemente a instrução negativa do prompt ("Do not add explanations") e listou uma quantidade excessiva de code smells para quase todos os arquivos.
    - **Smells frequentes apontados:** *Long Method*, *Long Parameter List*, *Data Clumps*, *Switch Statements*, *Speculative Generality*.
- **Justificativa Técnica:**
    - **Excesso de zelo (False Positives):** O Phi-4 classificou trechos de desestruturação de objetos (`const { a, b, c } = req.body`) como *Data Clumps* ou *Long Parameter List*. Embora tecnicamente possa ser debatido, no contexto de JavaScript moderno/Express, isso é um padrão comum, e não necessariamente um smell.
    - **Switch Statements:** O modelo apontou este smell em arquivos que não continham a palavra-chave `switch`. A justificativa técnica é que o modelo interpretou cadeias longas de `if/else` ou lógicas de roteamento condicionais complexas como equivalentes lógicos a um *Switch Statement*, demonstrando uma análise semântica, mas imprecisa na terminologia.

2.3. Análise do Modelo: DeepSeek R1 Distill Qwen (1.5B)

- **Resultados Observados:** O modelo apresentou as maiores dificuldades em seguir a formatação estrita solicitada ("Smell: <name>").
    - **Erro de Interpretação de Linguagem:** Em `workspace_index.js`, o modelo afirmou incorretamente: *"The code is from a Python file"*, tentando analisar sintaxe JavaScript como se fosse Python.
    - **Falha na Resposta (Template):** Em diversos casos (como `workspace_thread_index.js`), o modelo devolveu o próprio template do prompt ("Smell: <name>") em vez de preenchê-lo, ou

iniciou tags de pensamento (`</think>`) expondo sua dúvida interna sobre o código.

- **Justificativa Técnica:** Sendo um modelo "Distill" (destilado) e focado em *Reasoning* (Raciocínio), o DeepSeek tentou "pensar" sobre o problema antes de responder. O prompt rígido, que proibia explicações, parece ter conflitado com o treinamento de *Chain-of-Thought* (Cadeia de Pensamento) do modelo. Isso resultou em saídas onde o modelo "travava" tentando decidir se devia explicar ou apenas listar, resultando em respostas incoerentes ou loops de texto.

3 - Compare através de uma tabela ou algo similar os resultados da análise obtidos entre os modelos para cada um dos code smells identificados.

3.1 Comparativo de detecção de code-smells

Esta tabela mostra a sensibilidade de cada modelo para os principais smells identificados nos artefatos

Code Smell	Phi-3 Mini	Phi-4 Mini	DeepSeek R1
Long Method	Detectou (seletivo/repetitivo)	Detectou em 95%	Citou definições apenas
Long Parameter List	Raro (apenas no admin)	Frequente	Não identificado
Data Clumps	Não identificado	Frequente (Falso Positivo)	Não identificado
Switch Statements	Alucinou em loops	Semântico	Erro de sintaxe (Python)
Spec. Generality	Não identificado	Detectou	Não identificado

3.2 Comparação por módulo

Comparação da performance qualitativa em cada uma das três áreas do projeto.

Módulo	Característica dos Artefatos	Performance Phi-3	Performance Phi-4	Performance DeepSeek
API	Alta complexidade e lógica de rotas.	**Gerou loops cíclicos em arquivos grandes.**	**Listou múltiplos smells por arquivo.**	**Confundiu JS com Python no workspace_index.**
Endpoint	Lógica de processamento e middlewares.	**Frequentemente "No smell detected".**	**Classificou desestruturação como Data Clump.**	**Ficou preso em reflexões internas (<think>).**
SRC (UI)	Componentes React (main.jsx).	**Não encontrou evidências.**	**Viu Long Method na configuração de rotas.**	**Repetiu trechos do código sem analisar.**

3.3 Comparação entre os pontos fortes e fracos de cada modelo

Modelo	Pontos Fortes	Pontos Fracos e Limitações
Microsoft Phi-4 Mini	Alta capacidade de detecção semântica; identifica lógica de "Switch" mesmo sem a palavra-chave; muito detalhista.	Gerou muitos falsos positivos (ex: tratou desestruturação de objetos como erro); ignorou a proibição de explicações no prompt.
Microsoft Phi-3 Mini	Eficiente e assertivo em arquivos pequenos; identifica o "Long Method" de forma direta quando o contexto é limitado.	Instabilidade em contexto longo; sofre de "alucinação repetitiva", entrando em loops infinitos ao processar arquivos complexos (como o admin_index.js).
DeepSeek R1 (1.5B)	Possui uma forte base de raciocínio interno (Chain-of-Thought), tentando entender a arquitetura antes de responder.	Falha de instrução e sintaxe; confundiu JavaScript com Python e não conseguiu seguir o formato de saída "seco", resultando em respostas incompletas ou tags de pensamento expostas.

4 - Dentre os modelos selecionados pela sua equipe, avalie quais foram os mais efetivos na análise. Justifique sua resposta.

Dentre os modelos selecionados, o mais efetivo foi, com vantagem, o *microsoft/Phi-4-mini-instruct.* Essa vantagem pode ser observada por meio da análise dos seguintes aspectos:

- **Variedade e Profundidade**: O *Phi-4-mini-instruct* identificou uma gama maior de code smells em quase todos os artefatos analisados, incluindo não apenas Long Method e Large Class, mas também Long Parameter List, Data Clumps, Switch Statements, Speculative Generality, entre outros. Isso demonstra uma cobertura significativamente mais ampla das possíveis fragilidades estruturais do código.
- **Detecção Mais Fina**: Este modelo detectou smells onde outros modelos, especialmente o *DeepSeek-R1-Distill-Qwen-1.5B*, foram superficiais ou inconsistentes. A exemplo disso, o Phi-4-mini-instruct apontou com clareza a presença de Data Clumps e Long Parameter List – smells clássicos que apareceram com menor frequência nos outros dois modelos.
- **Justificativas Detalhadas**: A resposta do Phi-4-mini-instruct, além de indicar a presença do smell, geralmente traz pequenas explicações ou exemplos, facilitando a compreensão do porquê aquele smell foi sinalizado no contexto do artefato.
- **Menos Alucinações e Redundâncias:** Apesar de identificar mais smells, o modelo se manteve relativamente consistente com as evidências reais do código, ao contrário do DeepSeek, que por vezes exibiu raciocínio “solto” e respostas fora do formato solicitado, ou do Phi-3, que foi conservador demais, se limitando ao Long Method na maioria dos casos.
- **Relevância**: Em artefatos como agentFlows.js e document.js, o Phi-4 foi o único a apontar smells como Data Clumps e Large Class com justificativas coerentes, evidenciando melhor entendimento do contexto do código.

Portanto, o modelo *microsoft/Phi-4-mini-instruct* foi o mais efetivo porque conseguiu equilibrar completude (vasto espectro de smells identificados), precisão (consistência com as evidências do código) e clareza nas respostas. Isso potencializou a etapa de diagnóstico e fundamentou melhor a discussão sobre a evolução da qualidade estrutural do código ao longo do projeto, permitindo uma visualização mais precisa dos pontos críticos a serem refatorados.

5 - Baseado na análise dos code smells ao longo do tempo, qual impacto pode ser constatado na evolução do projeto analisado? Justifique sua resposta.

Com base na análise dos code smells identificados nas releases do projeto, é possível constatar que sua evolução apresenta indícios claros de acúmulo de responsabilidades em determinados artefatos centrais do sistema. A recorrência de smells como Long Method e Large Class indica que, ao longo do tempo, novas funcionalidades foram incorporadas principalmente por meio da extensão de métodos e arquivos já existentes, em vez da criação de abstrações ou componentes mais especializados. Esse padrão sugere uma evolução funcional do sistema sem o acompanhamento proporcional de práticas de refatoração.

A presença contínua desses problemas impacta diretamente a manutenibilidade do projeto, tornando o código mais difícil de compreender, testar e modificar. À medida que métodos crescem e assumem múltiplas responsabilidades, aumenta-se o acoplamento e reduz-se a clareza da lógica de negócio, o que pode elevar o risco de introdução de defeitos em futuras alterações. Além disso, smells como Long Parameter List e Data Clumps, quando presentes, indicam uma modelagem de dados que poderia ser aprimorada, reforçando a necessidade de uma melhor organização estrutural do código conforme o projeto evolui.

Especificamente, a predominância do *Long Method* — muito apontada na análise automatizada — indica um aumento crítico na dificuldade de entendimento e na quantidade de fluxos lógicos do sistema. Desse modo, isso acaba criando barreiras significativas para a implementação de testes unitários eficazes, uma vez que métodos extensos exigem cenários de teste muito mais complexos para garantir uma cobertura adequada. Sem essa proteção, o desenvolvimento fica mais lento, já que a equipe terá de gastar mais tempo corrigindo erros e testando manualmente para garantir que novas mudanças não afetem o que já funciona.

Observando a evolução do software, nota-se um possível desequilíbrio: o crescimento funcional parece ter superado a organização estrutural. Embora o

sistema demonstre resiliência ao continuar recebendo novos recursos, os *smells* encontrados sinalizam uma dívida técnica crescente. Isso sugere que a complexidade atual pode ser reflexo de um foco em entregas rápidas. Diante disso, há uma oportunidade de melhorar a agilidade futura ao tratar as refatorações não como algo opcional, mas como um investimento necessário na qualidade do código.

Uso de modelos auxiliares para geração dos texto;

1) Para a parte 1 da atividade, não foi utilizado nenhum tipo de LLM para geração do texto e análise de artefatos coletados.
2) Para a parte 2 foi utilizado o modelo: Gemini 3 para análise dos resultados contendo as identificações de code smells nos módulos do projeto para cada modelo escolhido para realização da atividade.
3) Para a parte 3 foi utilizado o modelo: Gemini 3 para geração do texto e das tabelas contendo as identificações de code smells por cada modelo, avaliando detecção, performance e pontos fortes e fracos encontrados por cada modelo na avaliação dos code smells.
4) Para a parte 4 foi utilizado o modelo: gpt-4.1 para geração do texto sobre avaliação dos modelos mais efetivos.
5) Para a parte 5 foi utilizado o modelo: Gemini 3 para geração do texto de avaliação de impacto no projeto causado pelos code smells identificado pelos modelos selecionados.

Referências

[1] Gemini 3 (Chat): utilizado na extração de dados do arquivo json com os resultados das análises de code smells pelos modelos.
[2] Gpt-4.1: Utilizado para geração dos textos sobre a avaliação gerada por cada modelo.
[3] REFRACTORING GURU. Code Smells. Disponível em: https://refactoring.guru/refactoring/smells