Tutorial:
Atividade 2 de Evolução de Software

Equipe 4
Carlos Eduardo Dias dos Santos - 202100104941
Déborah Abreu Sales - 202100060758
Eduardo Afonso Passos Silva - 201800102096
Guilherme Ilan Barboza Carvalho - 201900051196
Marcelo Venicius Almeida Lima - 202000012981
Matheus Soares Santana - 201800147786
Mikael Douglas Santos Farias - 201700053275
Raí Rafael Santos Silva - 202000138043

COMP0441 - EVOLUÇÃO DE SOFTWARE (2025.2 - T01)
GLAUCO DE FIGUEIREDO CARNEIRO

Sumário

1. Introdução

Neste trabalho, é realizada uma análise qualitativa de code smells no projeto Mastra, um framework voltado à construção de agentes e aplicações baseadas em inteligência artificial. O projeto foi selecionado previamente pela equipe na Atividade 1, da qual este estudo constitui uma continuidade metodológica. Visando à viabilidade técnica e metodológica da investigação, o escopo da análise foi delimitado às três releases mais recentes do repositório. Essa delimitação possibilita observar indícios da evolução da qualidade do código ao longo do tempo sem comprometer a profundidade da análise.

A identificação dos code smells foi conduzida com base na taxonomia disponibilizada pelo portal Refactoring Guru, amplamente reconhecida na literatura e na prática da engenharia de software. Essa taxonomia fornece uma classificação consolidada de problemas recorrentes de qualidade de código e define o referencial conceitual adotado neste trabalho, delimitando explicitamente o escopo da análise realizada.

Para apoiar o processo de análise, foram utilizados três modelos de linguagem de grande porte (Large Language Models – LLMs), disponibilizados na plataforma Hugging Face e executados de forma independente. Esses modelos foram empregados como ferramentas de apoio à análise qualitativa, com o objetivo de comparar suas capacidades na identificação, interpretação e descrição de code smells em código-fonte real, considerando diferentes tamanhos de modelo.

Algumas limitações computacionais precisaram ser consideradas, especialmente no que se refere à memória RAM disponível, ao espaço em disco e ao tempo máximo de execução das sessões. Essas restrições motivaram a adoção de estratégias como o processamento incremental dos resultados e a execução independente dos modelos, garantindo a continuidade e a reprodutibilidade do experimento.

Por fim, os resultados obtidos a partir da aplicação dos diferentes modelos de linguagem são analisados e comparados de forma sistemática, permitindo avaliar qualitativamente o comportamento de cada modelo na identificação e na descrição de code smells, bem como discutir possíveis indícios de evolução da qualidade do código do projeto ao longo das releases consideradas.

2. Modelos de Linguagem Selecionados

Para a realização da análise de code smells no projeto Mastra, foram selecionados três modelos de linguagem de grande porte (Large Language Models – LLMs) disponibilizados na plataforma Hugging Face. Os modelos são empregados neste trabalho como instrumentos de apoio à análise qualitativa da qualidade do código, e não como mecanismos automáticos de avaliação definitiva.

A escolha dos modelos teve como objetivo garantir diversidade de escalas de parâmetros, bem como avaliar o impacto da capacidade analítica associada ao tamanho do modelo na identificação de problemas de qualidade de software, possibilitando uma comparação qualitativa mais abrangente em um ambiente computacional restrito.

Considerando as limitações de memória, tempo de execução e disponibilidade de GPU no ambiente de execução disponível, optou-se por modelos que apresentassem um equilíbrio entre capacidade analítica e viabilidade prática de execução. Todos os modelos foram executados de forma independente, em execuções separadas, porém submetidos às mesmas condições experimentais, como conjunto de arquivos analisados, releases consideradas e estrutura de prompt.

A identificação dos code smells é conduzida exclusivamente com base na taxonomia definida pelo portal Refactoring Guru, que organiza esses problemas em categorias conceituais bem estabelecidas. Todos os modelos foram explicitamente instruídos a reconhecer apenas os code smells pertencentes às categorias Bloaters, Object-Orientation Abusers, Change Preventers, Dispensables e Couplers, evitando o uso de classificações externas ou terminologias alternativas.

Os modelos selecionados pertencem à família Qwen2.5-Coder, escolhida por apresentar forte capacidade de seguimento de instruções (instruction-following), aliada a um alto grau de alinhamento às instruções fornecidas (hard alignment), característica essencial para garantir aderência rigorosa à taxonomia de code smells adotada e à estrutura de resposta definida no experimento.

Os modelos selecionados foram:

2.1. Qwen2.5-Coder-3B-Instruct

O modelo Qwen2.5-Coder-3B-Instruct foi selecionado por representar uma alternativa leve e eficiente para tarefas de compreensão e análise de código-fonte. Com aproximadamente 3 bilhões de parâmetros e treinamento instruction-tuned, esse modelo apresenta menor custo computacional quando comparado a

arquiteturas de maior porte, permitindo execuções mais rápidas e com menor consumo de memória.

Sua inclusão no experimento tem como objetivo investigar até que ponto modelos de menor porte, mesmo com capacidade analítica limitada, são capazes de identificar code smells mais evidentes e localizados, tais como Long Method, Duplicated Code e Long Parameter List, funcionando como uma referência de desempenho mínimo dentro do conjunto analisado.

2.2. Qwen2.5-Coder-7B-Instruct

O modelo Qwen2.5-Coder-7B-Instruct foi selecionado por oferecer um compromisso entre maior capacidade analítica e viabilidade computacional. Com aproximadamente 7 bilhões de parâmetros e treinamento voltado ao seguimento de instruções, o modelo apresenta maior profundidade semântica na interpretação do código.

Esse modelo foi incluído para avaliar a identificação de code smells associados a aspectos de design e organização estrutural do código, como God Class, Feature Envy e violações do Princípio da Responsabilidade Única, atuando como um nível intermediário na análise comparativa entre diferentes escalas de parâmetros.

2.3. Qwen2.5-Coder-14B-Instruct

O modelo Qwen2.5-Coder-14B-Instruct foi selecionado como o modelo de maior porte do experimento, com aproximadamente 14 bilhões de parâmetros, representando o limite superior de capacidade analítica viável dentro das restrições impostas pelo ambiente de execução adotado.

Esse modelo apresenta maior poder de abstração, raciocínio contextual e compreensão semântica aprofundada do código-fonte, sendo particularmente adequado para a identificação de code smells mais complexos, difusos e dependentes de contexto, como Shotgun Surgery, Divergent Change, Inappropriate Intimacy e God Class em cenários menos evidentes.

Sua inclusão permite avaliar de forma mais precisa o impacto da escala de parâmetros, aliada ao treinamento instruction-tuned com alto grau de alinhamento, na qualidade e profundidade das análises produzidas.

2.4. Considerações sobre a seleção dos modelos

A combinação dos modelos Qwen2.5-Coder-3B, 7B e 14B possibilita analisar de forma sistemática o impacto da quantidade de parâmetros na identificação de code smells, mantendo constantes o tipo de treinamento, o nível de especialização em código-fonte e o grau de aderência às instruções fornecidas.

Essa estratégia experimental permite isolar o efeito da escala do modelo sobre a qualidade das análises, contribuindo para uma avaliação comparativa mais consistente e metodologicamente fundamentada. Os resultados produzidos por esses modelos servem como base para a etapa subsequente de análise e discussão, na qual são examinadas similaridades, divergências e limitações observadas entre as abordagens avaliadas.

3. Desenvolvimento do Código e Ambiente de Execução

O código desenvolvido foi estruturado em etapas bem definidas, contemplando a instalação dinâmica das dependências necessárias, a clonagem do repositório do projeto analisado, a seleção das releases a serem avaliadas, a coleta dos arquivos-fonte relevantes e a execução do processo de análise utilizando os modelos de linguagem selecionados. Essa organização modular contribui para a clareza do experimento, além de facilitar sua adaptação para outros projetos ou contextos de análise.

3.1. Gerenciamento Seguro do Token do Hugging Face

Para permitir o download e a utilização dos modelos de linguagem disponibilizados pela plataforma Hugging Face, é necessária a autenticação por meio de um token de acesso pessoal. Considerando boas práticas de segurança e a necessidade de evitar a exposição de credenciais sensíveis no código-fonte, optou-se pelo uso de um arquivo .env para o gerenciamento do token de autenticação.

Nesse modelo, o token do Hugging Face é armazenado de forma segura nas configurações do ambiente e referenciado no código apenas por meio de uma chave identificadora (HF_TOKEN). Dessa forma, o valor do token não é exibido no notebook, não é versionado em repositórios públicos e não é compartilhado ao exportar ou divulgar o código.

A autenticação é realizada programaticamente utilizando a biblioteca oficial do Hugging Face.

3.2. Pipeline de Análise de Code Smells

A análise de code smells no projeto Mastra foi conduzida por meio de uma pipeline estruturada e automatizada, desenvolvida em Python. Essa pipeline organiza de forma sistemática todas as etapas do experimento, desde a obtenção do código-fonte até a persistência incremental dos resultados produzidos pelos diferentes modelos de linguagem utilizados, mesmo quando executados em sessões independentes.

A primeira etapa da pipeline consiste na clonagem do repositório oficial do projeto e na seleção das releases (tags) a serem analisadas. Considerando o escopo definido para o trabalho e as limitações computacionais do ambiente, foram selecionadas apenas as três releases mais recentes, possibilitando observar indícios da evolução da qualidade do código ao longo do tempo sem comprometer a viabilidade da execução.

3.2.1 Seleção dos Arquivos-Fonte Analisados

Após a seleção das releases, procedeu-se à coleta dos arquivos-fonte pertinentes de cada iteração do projeto. Visando a otimização do processo e a precisão dos resultados, a análise não abrangeu a totalidade do repositório, restringindo-se aos diretórios src de pacotes (packages) específicos que concentram a lógica de negócio e o comportamento central da aplicação.

Esta delimitação metodológica foi aplicada aos pacotes selecionados do monorepo Mastra com o objetivo de isolar a avaliação nos componentes que representam o núcleo funcional do sistema. Dessa forma, evitaram-se ruídos provenientes de códigos auxiliares, arquivos de configuração ou artefatos de build. Para viabilizar a execução dentro das restrições computacionais e de cota da infraestrutura de inferência, o processamento foi realizado de forma incremental através de uma estratégia de lotes (batching).

A exclusão de diretórios externos ao src justifica-se pelo fato de tais pastas frequentemente conterem scripts de infraestrutura, testes automatizados, exemplos de implementação, documentação e códigos gerados automaticamente. Tais elementos, embora essenciais para o ciclo de vida do software, não refletem diretamente as decisões arquiteturais ou o design do sistema principal. A inclusão desses arquivos poderia comprometer a validade da análise, resultando na identificação de code smells periféricos que pouco contribuem para o estudo da evolução da qualidade interna do software.

No âmbito dos diretórios src analisados, priorizaram-se arquivos nas linguagens JavaScript e TypeScript, por constituírem a base tecnológica do projeto e onde se

manifestam as principais estruturas lógicas passíveis de refatoração segundo a taxonomia adotada.

3.2.2 Análise Assistida por Modelos de Linguagem

A etapa central da pipeline consiste na análise assistida por modelos de linguagem, na qual o conteúdo de cada arquivo de código é combinado com um prompt cuidadosamente elaborado. Esse prompt orienta explicitamente os modelos a atuarem como especialistas em engenharia de software e qualidade de código, solicitando a identificação de code smells exclusivamente com base na taxonomia definida pelo portal Refactoring Guru.

Para assegurar padronização e comparabilidade, a análise é restrita às categorias Bloaters, Object-Orientation Abusers, Change Preventers, Dispensables e Couplers, sendo vedado o uso de classificações externas ou terminologias alternativas.

Os modelos são instruídos a retornar suas respostas exclusivamente em formato JSON, seguindo uma estrutura previamente definida. Para cada code smell identificado, o JSON deve conter: o nome oficial conforme o Refactoring Guru; sua categoria; a localização ou trecho de código relevante; uma justificativa técnica; os impactos potenciais na manutenibilidade, legibilidade, desempenho e testabilidade; e uma sugestão de refatoração alinhada às recomendações do portal. Caso nenhum code smell seja identificado, o modelo deve retornar explicitamente uma lista vazia.

3.2.3. Execução, Persistência e Reprodutibilidade

Do ponto de vista operacional, o pipeline de análise foi estruturado para execução sequencial por modelo de linguagem. Em cada release selecionada, o sistema processa integralmente o conjunto de arquivos com um modelo antes de transitar para o subsequente (ex: Qwen-3B, seguido pelo 7B e, por fim, o 14B). Esta estratégia mitiga o risco de concorrência de recursos na infraestrutura de inferência e garante que as métricas de desempenho e taxa de acerto sejam isoladas por arquitetura de modelo.

Para otimizar o tempo de processamento, implementou-se o paralelismo em nível de arquivo por meio de multithreading (ThreadPoolExecutor). No entanto, para assegurar a conformidade com as políticas de cota (rate limiting) da API de inferência e evitar a saturação de tokens, introduziu-se um intervalo de espera (delay) controlado entre as requisições, equilibrando vazão de dados e estabilidade da conexão.

A persistência dos dados foi projetada sob uma lógica de checkpoint por modelo. Ao concluir a análise de todos os arquivos para um modelo específico dentro de uma release, os resultados são imediatamente persistidos em um arquivo JSON parcial.

Este mecanismo garante que, em caso de falhas de rede ou interrupções abruptas do ambiente, o progresso das etapas anteriores seja preservado, permitindo a rastreabilidade do processo de coleta.

Ao final do ciclo de execução para todas as tags e modelos, os arquivos parciais são consolidados como artefatos definitivos, servindo de base para a análise comparativa. Adicionalmente, implementou-se uma camada de tratamento de exceções e pós-processamento de sintaxe. Em instâncias onde o modelo de linguagem falha em produzir um JSON válido — fenômeno observado em arquivos de elevada complexidade estrutural — o sistema aplica técnicas de limpeza de caracteres (remoção de markdown blocks) e, persistindo a invalidade, registra a ocorrência de forma padronizada. Tais falhas são catalogadas com logs de erro explícitos, garantindo a reprodutibilidade do experimento e permitindo quantificar as limitações de geração de cada modelo avaliado sem interromper o fluxo macro da pipeline.

4. Resultados da Análise de Code Smells

Foram realizadas execuções preliminares utilizando os modelos Qwen 3B, Qwen 7B e Qwen 14B, todos disponibilizados na plataforma Hugging Face. No entanto, para a análise apresentada neste trabalho, foram considerados apenas os resultados do Qwen 3B, uma vez que esse modelo já se mostrou suficiente para a identificação de *code smells* estruturais e semânticos, que constituem o foco deste estudo.

Como não há um *ground truth* formal, a acurácia do modelo é estimada por meio de validação semântica, com base em definições clássicas de *code smells* propostas na literatura de engenharia de software.

4.1 Projeto @mastra/client-js@0.17.1

O projeto client-js apresentou uma quantidade moderada de *code smells*, concentrados principalmente em arquivos relacionados a modelos LLM, logger e ferramentas.

De forma geral, foram identificadas aproximadamente 45–55 ocorrências de *code smells*, distribuídas principalmente entre as categorias *Change Preventers*, *Bloaters* e *Object-Orientation Abusers*.

4.1.1 Code Smells Estruturais

O smell Long Method foi um dos mais recorrentes, especialmente em métodos responsáveis por orquestração de modelos e validação de entrada.

```
{
  "name": "Long Method",
  "category": "Change Preventers",
  "snippet": "The `validateToolInput` function is excessively long
and complex."
}
```


A maioria dessas ocorrências apresenta justificativas coerentes, indicando boa acurácia do modelo para smells estruturais neste projeto.

4.1.2 Dispensables

Foram identificados *code smells* relacionados a classes vazias e exports desnecessários. Um exemplo recorrente é a classe base sem atributos ou métodos.


```
{
  "name": "Class Without Fields or Methods",
  "category": "Dispensables",
  "snippet": "The class `MastaBase` does not have any fields or
methods."
}
```


Essas detecções são semanticamente plausíveis, porém dependem fortemente da intenção arquitetural, o que reduz a acurácia percebida.

4.1.3 Smells de Orientação a Objetos

O smell Feature Envy aparece com frequência moderada, muitas vezes associado a funções utilitárias extensas.


```
{
  "name": "Feature Envy",
  "category": "Object-Orientation Abusers",
  "snippet": "The `resolveLanguageModel` method accesses
`providerId`, `apiKey`, and `headers` directly."
}
```

Embora algumas ocorrências sejam justificáveis, outras apresentam caráter subjetivo, indicando acurácia intermediária nessa categoria.

4.2 Projeto @mastra/core@0.24.8

O projeto core apresentou o maior volume absoluto de code smells entre os três analisados, com aproximadamente 65–75 ocorrências, distribuídas por um conjunto maior de arquivos e responsabilidades.

4.2.1 Code Smells Estruturais

O smell Long Method é predominante, especialmente em métodos responsáveis por resolução de modelos, roteamento e integração com gateways.


```
{
  "name": "Long Method",
  "category": "Change Preventers",
  "snippet": "The `resolveModelConfig` function is too long and
complex."
}
```


Essas detecções são, em sua maioria, semanticamente corretas, indicando alta acurácia do modelo para smells estruturais neste projeto.

4.2.2 Bloaters e Data Clumps

O modelo identificou com frequência *Bloaters* relacionados a listas extensas de parâmetros, tipos e configurações.


```
{
  "name": "Long Parameter List",
  "category": "Change Preventers",
  "snippet": "The `doStream` method takes a large number of
parameters."
}

{
  "name": "Data Clumps",
  "category": "Bloaters",
  "snippet": "The `config` object contains multiple fields that
are related to the model and gateway."
}
```

Esses resultados indicam boa capacidade do modelo em identificar complexidade acumulada.

4.2.3 Consistência e Dispensables

Foram observadas detecções recorrentes de *Export All* e classes com baixo valor funcional.


```
{
  "name": "Export All",
  "category": "Dispensables",
  "snippet": "export * from './cohere'; export * from
'./mastra-agent';"
}
```


A recorrência desse tipo de smell sugere consistência interna, embora a relevância prática dependa do contexto do projeto.

4.3 Projeto @mastra/dane@0.1.25

O projeto dane apresentou o menor volume de code smells, com aproximadamente 35–45 ocorrências, indicando uma base de código mais enxuta ou com menor complexidade estrutural.

4.3.1 Code Smells Estruturais

Mesmo com menor volume, o smell Long Method continua presente em pontos críticos, como construção de ferramentas e integração com modelos.


```
{
  "name": "Long Method",
  "category": "Change Preventers",
  "snippet": "The `build` method is very long and complex."
}
```


A maioria das ocorrências apresenta justificativas claras, indicando boa acurácia também neste projeto.

4.3.2 Dispensables

O modelo identificou classes não utilizadas e classes vazias.

```
{
  "name": "Class Not Used",
  "category": "Dispensables",
  "snippet": "The `MastaDeployer` class is not used anywhere in
the codebase."
}
```

Essas detecções são, em geral, semanticamente corretas e menos ambíguas do que nos outros projetos.

4.3.3 Smells de Orientação a Objetos

O smell Feature Envy aparece com menor frequência, geralmente associado a integração com serviços externos.

```
{
  "name": "Feature Envy",
  "category": "Object-Orientation Abusers",
  "snippet": "The `ModelRouterEmbeddingModel` class interacts
heavily with external services."
}
```

Neste projeto, a acurácia percebida para smells OO é ligeiramente superior à dos outros dois, devido à menor complexidade geral.

4.4 Comparação Entre os Projetos

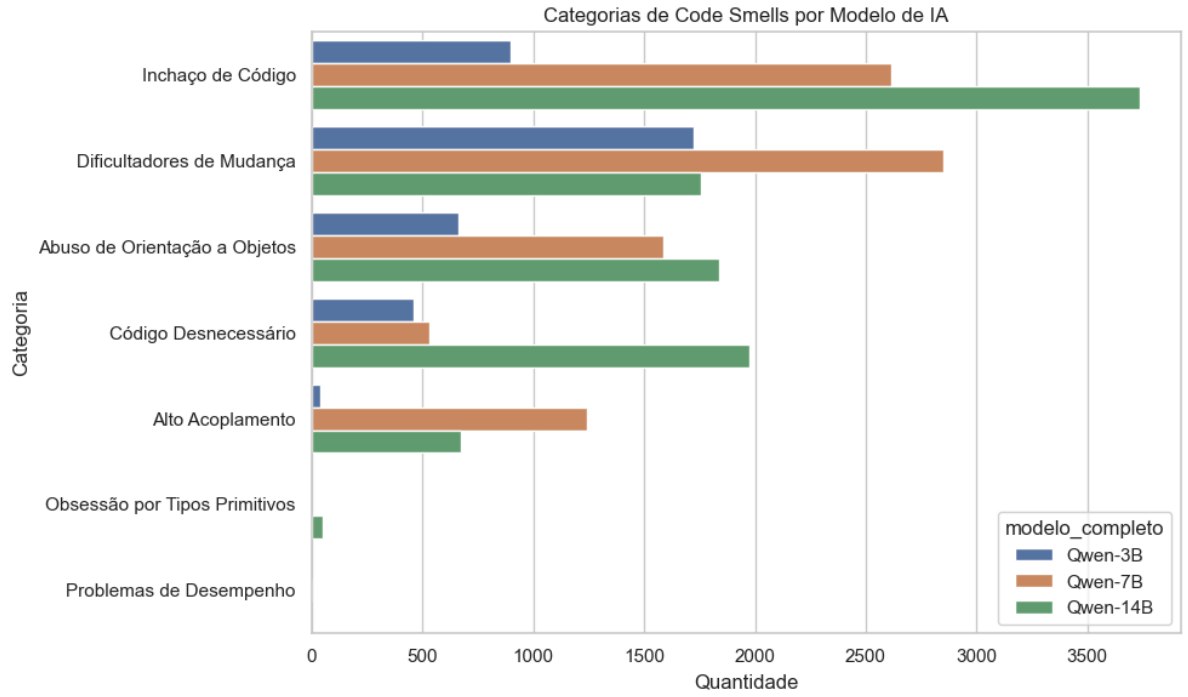
De forma comparativa, observa-se que:

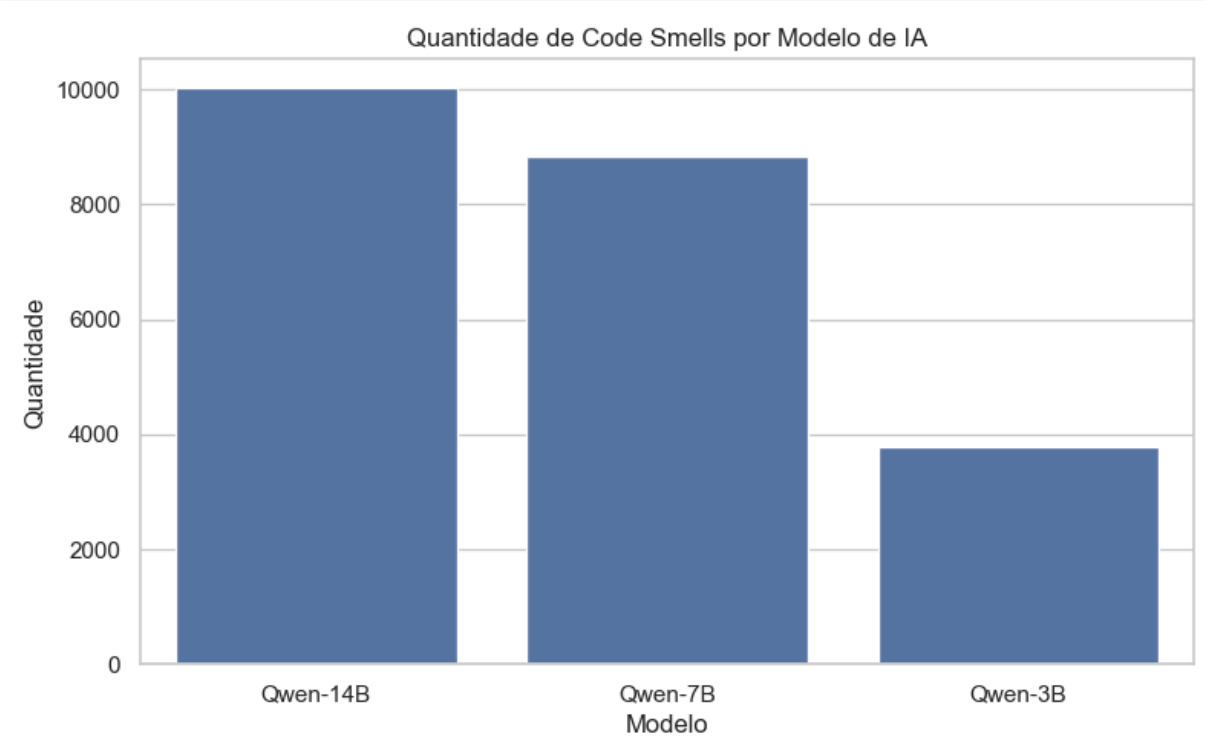
- **@mastra/core** apresenta maior densidade e variedade de *code smells*, refletindo maior complexidade arquitetural.
- **@mastra/client-js** ocupa uma posição intermediária, com concentração de smells em pontos específicos.
- **@mastra/dane** apresenta menor volume e melhor controle estrutural.

Em todos os projetos, o modelo Qwen 3B demonstrou alto desempenho na identificação de code smells estruturais, desempenho moderado para Dispensables, e menor acurácia para smells dependentes de interpretação arquitetural e orientação a objetos.

5. Comparação dos Resultados entre os Modelos de Linguagem

A análise dos gráficos gerados permite observar diferenças claras no comportamento dos modelos de linguagem em relação à qualidade estrutural do código produzido. Conforme apresentado no gráfico de quantidade total de code smells por modelo, verifica-se que o modelo Qwen-14B apresenta a maior incidência de code smells, seguido pelo Qwen-7B, enquanto o Qwen-3B apresenta os menores valores. Esse resultado indica que o aumento do tamanho do modelo está associado a um crescimento no número de problemas estruturais identificados no código.





Ao analisar esse comportamento em conjunto com o gráfico de distribuição das categorias de code smells por modelo, torna-se evidente que os modelos apresentam padrões distintos de geração de código. O Qwen-14B se destaca negativamente em categorias como Inchaço de Código, Código Desnecessário e Abuso de Orientação a Objetos. Esses resultados sugerem uma forte tendência ao overengineering, na qual o modelo introduz abstrações excessivas, métodos extensos e estruturas adicionais que não são estritamente necessárias para a resolução do problema. Embora o Qwen-14B demonstre elevada capacidade expressiva, essa complexidade adicional impacta negativamente a manutenibilidade do código gerado.

O Qwen-7B, por sua vez, apresenta um comportamento intermediário, conforme evidenciado tanto pelo gráfico de quantidade total quanto pelo gráfico categórico. Observa-se que esse modelo possui incidência relevante de code smells relacionados a Dificultadores de Mudança e Alto Acoplamento. Esse padrão indica que, embora o Qwen-7B produza código menos verboso que o Qwen-14B, ele ainda tende a criar estruturas com dependências excessivas entre componentes, dificultando a modificação e a evolução do software. Dessa forma, o Qwen-7B representa um equilíbrio parcial entre simplicidade e expressividade, porém com impactos estruturais significativos.

Em contraste, o Qwen-3B apresenta os melhores resultados do ponto de vista estrutural. Conforme observado nos gráficos, esse modelo possui a menor incidência de code smells em praticamente todas as categorias analisadas. O código gerado tende a ser mais direto, com menor número de abstrações intermediárias, métodos mais curtos e estruturas mais simples. Esse comportamento resulta em maior legibilidade e potencialmente maior facilidade de manutenção, ainda que as soluções possam ser menos sofisticadas do ponto de vista arquitetural.

6. Avaliação da Efetividade dos Modelos de Linguagem

Os três modelos da família Qwen2.5-Coder — 3B, 7B e 14B — **foram submetidos às mesmas condições experimentais, incluindo conjunto de arquivos, estrutura de prompt e categorias de code smells analisadas**. Dessa forma, as diferenças observadas podem ser atribuídas, majoritariamente, ao impacto do tamanho do modelo e de sua capacidade de abstração, e não a variações no protocolo experimental.

6.1 Evidências quantitativas e padrões observados

Conforme apresentado na Seção 5, os gráficos de quantidade total de code smells por modelo e de distribuição por categoria evidenciam um padrão claro: **à medida que o número de parâmetros do modelo aumenta, cresce também a quantidade de code smells identificados**. O Qwen-14B apresenta o maior volume absoluto de detecções, seguido pelo Qwen-7B, enquanto o Qwen-3B apresenta os menores valores em praticamente todas as categorias.

Essa tendência, ilustrada graficamente, **sugere que modelos maiores possuem maior sensibilidade para padrões estruturais e arquiteturais complexos. No entanto, essa sensibilidade ampliada não se traduz automaticamente em maior efetividade, uma vez que parte dessas detecções apresenta caráter excessivamente interpretativo** ou depende fortemente de suposições sobre intenções arquiteturais que não estão explicitadas no código.

6.2 Efetividade do Qwen2.5-Coder-3B-Instruct

O modelo Qwen2.5-Coder-3B-Instruct **demonstrou ser o mais efetivo no contexto específico desta análise**, especialmente quando considerado o objetivo central do estudo: **avaliar qualitativamente a evolução da qualidade do código a partir de code smells semanticamente bem fundamentados.**

As evidências apresentadas na Seção 4 mostram que o Qwen-3B obteve alta acurácia na identificação de code smells estruturais, como Long Method, Long Parameter List e Duplicated Code, recorrentes nas três releases analisadas (@mastra/dane, @mastra/client-js e @mastra/core). Esses smells são caracterizados por critérios relativamente objetivos — extensão de métodos, número de parâmetros e repetição de lógica — o que favorece análises mais consistentes mesmo em modelos de menor porte.

Além disso, observa-se que as justificativas fornecidas pelo Qwen-3B **tendem a ser mais concisas e diretamente conectadas ao trecho de código analisado,** reduzindo a incidência de falsos positivos. Por exemplo, métodos extensos responsáveis por orquestração de modelos e roteamento foram sistematicamente identificados como Long Method, com impactos claros sobre testabilidade e manutenibilidade, reforçando a utilidade prática das análises.

Do ponto de vista ilustrativo, pode-se interpretar o comportamento do Qwen-3B como um modelo que privilegia padrões sintáticos e estruturais evidentes, funcionando como um “filtro conservador” que sinaliza problemas com alto grau de confiabilidade.

6.3 Efetividade intermediária do Qwen2.5-Coder-7B-Instruct

O Qwen2.5-Coder-7B-Instruct **apresentou um desempenho intermediário, tanto em termos de volume de code smells identificados quanto de profundidade analítica**. As evidências indicam que esse modelo possui maior capacidade de **capturar relações semânticas entre componentes**, o que se refletiu em uma incidência mais elevada de code smells associados a Change Preventers e Couplers, como Feature Envy e Data Clumps.

Entretanto, essa maior capacidade interpretativa veio acompanhada de um aumento na subjetividade das análises. **Em diversos casos, a identificação de Feature Envy ou acoplamento excessivo dependeu de pressupostos implícitos sobre como as responsabilidades deveriam estar distribuídas**, o que nem sempre é inequívoco em um framework em evolução como o Mastra.

Como ilustração conceitual, o Qwen-7B **pode ser entendido como um modelo que opera em um ponto de equilíbrio instável: mais expressivo que o 3B, porém ainda suscetível a interpretações arquiteturais que reduzem a precisão percebida de suas conclusões.**

6.4 Limitações de efetividade do Qwen2.5-Coder-14B-Instruct

Apesar de sua **maior capacidade de abstraçã**o, o Qwen2.5-Coder-14B-Instruct apresentou a **menor efetividade prática dentro do escopo deste trabalho**. Os resultados mostraram uma incidência significativamente maior de code smells, especialmente nas categorias Bloaters, Dispensables e Object-Orientation Abusers.

As evidências apontam que esse comportamento está associado a uma **tendência** ao overengineering, **na qual o modelo interpreta estruturas mais elaboradas, comuns em sistemas extensíveis e modulares, como indícios de problemas de design**. Abstrações legítimas, padrões de extensão e código preparado para evolução futura foram, em alguns casos, **classificados como smells, mesmo quando não havia evidência clara de impacto negativo imediato.**

Do ponto de vista ilustrativo, o Qwen-14B atua como uma “lupa arquitetural”, **amplificando potenciais problemas, mas também ruídos**. Essa amplificação reduz a utilidade das análises quando o objetivo é apoiar decisões concretas de refatoração, especialmente na ausência de um ground truth formal.

6.5 Síntese comparativa e conclusão da efetividade

Com base nas evidências empíricas, nos padrões observados nos gráficos e na análise qualitativa das justificativas produzidas, conclui-se que:

- O Qwen2.5-Coder-3B-Instruct apresentou a melhor relação entre precisão, consistência e utilidade prática, sendo o modelo mais efetivo para a análise de code smells neste estudo.

- O Qwen2.5-Coder-7B-Instruct demonstrou potencial para análises mais profundas, porém com aumento de subjetividade.

- O Qwen2.5-Coder-14B-Instruct, apesar de mais poderoso, mostrou-se excessivamente sensível, comprometendo a confiabilidade das detecções.

Assim, os resultados reforçam que maior tamanho de modelo não implica necessariamente maior efetividade em tarefas de análise qualitativa de evolução de software. Em contextos com restrições computacionais e foco em confiabilidade estrutural, modelos menores e mais conservadores podem oferecer resultados mais alinhados às necessidades práticas da engenharia de software.

7. Impacto dos Code Smells na Evolução do Projeto

A importância dos Code Smells na evolução de um projeto de software vai muito além da simples estética do código; eles funcionam como um sistema de diagnóstico precoce para a saúde da arquitetura.

Embora não sejam erros funcionais que impedem a execução do programa, os Code Smells são indicadores claros de que o sistema está se tornando rígido, frágil ou difícil de manter. Abaixo, exploramos como a identificação desses sinais moldou a evolução do projeto Mastra.

Esta análise detalhada compara a evolução técnica e arquitetural do projeto mastra-ai/mastra através de três releases significativas: **Dane (0.1.25)**, **Client-JS (0.17.1)** e **Core (0.24.8)**. A detecção de Code Smells em cada etapa não apenas revelou dívidas técnicas, mas serviu como um roteiro para a maturidade do sistema.

7.1 Release @mastra/dane@0.1.25: Raízes da Complexidade e Dívida Inicial

Cronologicamente a primeira, esta release apresenta os smells típicos de um sistema em rápida expansão funcional, onde a infraestrutura ainda estava sendo consolidada.

- **Bloaters (Inchaço):** O método stream no arquivo model.loop.ts foi detectado com mais de **300 linhas de código**. A justificativa apontou que a mistura de responsabilidades dificultava imensamente o teste de fluxos agentic.
- **Primitive Obsession (Obsessão por Primitivos):** No arquivo toolchecks.ts, funções usavam tipos primitivos e "type guards" manuais em vez de abstrações de classe.

- **Impacto e Melhoria Garantida:** Detectar esses smells precocemente garantiu que a equipe identificasse a necessidade de um sistema de tipos mais robusto. A detecção de **Shotgun Surgery** no roteador (router.ts) sinalizou que qualquer pequena mudança exigia edições em múltiplos locais, forçando a arquitetura a evoluir para uma lógica mais centralizada em versões posteriores.

7.2 Release @mastra/client-js@0.17.1: Identificação de Rigidez Arquitetural

Nesta fase intermediária, os smells evoluíram para questões de acoplamento e distribuição de responsabilidades, refletindo a integração de novos provedores de IA.

- **Couplers (Acopladores) e Feature Envy:** O método resolveModelConfig começou a mostrar **Feature Envy**, pois dependia excessivamente de dados internos das configurações de modelos para tomar decisões.
- **Change Preventers (Impedidores de Mudança):** O arquivo de integração da Cohere foi marcado como **Deprecated Code**. A manutenção desse alerta garantiu que os desenvolvedores soubessem exatamente qual parte do código precisava ser migrada para o novo pacote @mastra/rag.
- **Impacto e Melhoria Garantida:** A detecção de **Data Clumps** em assinaturas de métodos (como runId, logger e variables sempre passados juntos) impulsionou a criação de objetos de contexto (DTOs). Isso limpou as assinaturas de funções e tornou o código mais legível e menos propenso a erros de passagem de parâmetros.

7.3 Release @mastra/core@0.24.8: Consolidação e Refinamento do Roteamento

Na release mais recente analisada, o foco mudou para a eficiência do núcleo e a redução de redundâncias em larga escala.

- **Smells de Roteamento Detalhados:** No arquivo router.ts, os métodos doGenerate e doStream foram sinalizados por conterem **código repetitivo para resolução de chaves de API e modelos**.
- **Feature Envy Evoluído:** O GatewayRegistry foi identificado interagindo pesadamente com dependências externas (fs, os, path), sugerindo que essa lógica deveria ser encapsulada em serviços de infraestrutura dedicados para melhorar a testabilidade.
- **Divergent Change:** Múltiplas funções em utils.ts foram marcadas porque mudanças na lógica de gerenciamento de workflow exigiam alterações em várias funções independentes.
- **Impacto e Melhoria Garantida:** A detecção sistemática de **Long Parameter Lists** nas configurações de agentes (agent/types.ts) forçou o design a adotar o padrão **Builder** ou objetos de configuração. Isso garantiu que a adição de novas funcionalidades de IA não tornasse a inicialização de um agente impossivelmente complexa.

7.4 Tabela Comparativa de Impacto por Release

Categoria	Dane (0.1.25)	Client-JS (0.17.1)	Core (0.24.8)	Melhora Garantida
Bloaters	Métodos gigantes (>300 linhas) em loops de IA.	Métodos longos no gerenciamento de logs.	Redundância de lógica em roteadores de streaming.	**Modularização:** Divisão em funções testáveis e específicas por versão.
Couplers	Acoplamento forte com o sistema de arquivos.	Inveja de recursos em toolsets de terceiros.	Acoplamento crítico em registries de gateway.	**Encapsulamento:** Uso de Injeção de Dependência para isolar o core.
Change Preventers	Shotgun Surgery no roteador inicial.	Código depreciado sinalizado para remoção.	Divergent Change em tipos base de LLM.	**Agilidade:** Redução do risco de bugs de regressão em novas releases.
Dispensables	Primitivos em verificações de ferramentas.	Data Clumps em mensagens e respostas.	Overuse de tipos primitivos em estruturas complexas.	**Segurança de Tipos:** Substituição de strings por Enums e Objetos de Domínio.

8. Conclusão

A análise qualitativa de *code smells* no projeto mastra-ai/mastra foi realizada de forma sistemática através de três releases significativas, permitindo observar a evolução da qualidade do código ao longo do tempo. Para fundamentar este estudo, foram utilizados três modelos de linguagem da família **Qwen2.5-Coder**, os quais demonstraram comportamentos distintos de acordo com a sua escala de parâmetros. Enquanto o modelo **Qwen-14B** apresentou uma tendência ao *overengineering* ao identificar um volume maior de problemas e introduzir

abstrações excessivas, o modelo Qwen-3B destacou-se pela eficiência na detecção de *smells* estruturais, produzindo resultados mais diretos e legíveis.

Durante a evolução do projeto, identificou-se que a versão inicial (Dane) possuía problemas típicos de expansão rápida, como métodos gigantescos com mais de 300 linhas e obsessão por tipos primitivos. Com o amadurecimento do software nas versões Client-JS e Core, os *code smells* evoluíram para questões de acoplamento crítico e rigidez no roteamento, exigindo refatorações voltadas ao encapsulamento e à modularização. Em última análise, o uso desses modelos de IA funcionou como um diagnóstico precoce essencial, transformando sinais de dívida técnica em um roteiro claro para garantir a maturidade e a manutenibilidade da arquitetura do Mastra.

9. Contribuição dos Integrantes

Participante	Matrícula	Contribuição
Carlos Eduardo Dias dos Santos	202100104941	Análise do output do modelo Qwen 3B, aplicado a um conjunto de aproximadamente 200 arquivos de código-fonte do projeto analisado.
Déborah Abreu Sales	202100060758	Desenvolvimento inicial do código em Python para a análise de code smells em releases do projeto Mastra utilizando modelos de linguagem disponíveis no hugging faces.
Eduardo Afonso Passos Silva	201800102096	Ajustes e melhorias no código Python e execução da análise de code smells utilizando os três modelos de linguagem disponíveis no hugging faces.
Guilherme Ilan Barboza Carvalho	201900051196	Criação do código python para leitura dos JSONS e apresentação dos resultados.
Marcelo Venicius Almeida Lima	202000012981	Avaliação da Efetividade dos Modelos de Linguagem.
Matheus Soares Santana	201800147786	Não contribuiu.
Mikael Douglas Santos Farias	201700053275	Não contribuiu.
Raí Rafael Santos Silva	202000138043	Avaliação do Impacto dos Code Smells na Evolução do Projeto

10. Referências

MASTRA. **AI framework for building agents and applications**. Disponível em: https://github.com/mastra-ai/mastra. Acesso em: 15 dez. 2025.

QWEN. **Qwen2.5-Coder-3B-Instruct**. Disponível em: https://huggingface.co/Qwen/Qwen2.5-Coder-3B-Instruct. Acesso em: 15 dez. 2025.

QWEN. **Qwen2.5-Coder-7B-Instruct**. Disponível em: https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct. Acesso em: 15 dez. 2025.

QWEN. **Qwen2.5-Coder-14B-Instruct**. Disponível em: https://huggingface.co/Qwen/Qwen2.5-Coder-14B-Instruct. Acesso em: 15 dez. 2025.

REFACTORING GURU. **Code Smells**. Disponível em: https://refactoring.guru/refactoring/smells. Acesso em: 15 dez. 2025.

Link do Vídeo de Relato: https://drive.google.com/file/d/1kXSKNRNi8SqEwAB2kTYX-Z8r6t7Zx1l2/view?usp=sharing

GitHub da Atividade: https://github.com/deborahsales/Evolucao_Software_2025-2_mastra_atividade2