DETECTANDO ÁTOMOS DE CONFUSÃO UTILIZANDO O GEMINI: PERCEPÇÕES INICIAIS.

Marcus Vinícius Ribeiro Andrade

RESUMO

O presente trabalho tem como objetivo descrever os primeiros insights acerca da eficácia de uma popular inteligência artificial generativa, o Gemini, para a identificação de atoms of confusion (átomos de confusão). Para tanto, serão analisados dois prompts distintos, a saber: i) Um prompt genérico, a fim de verificar se ambos os sistemas supracitados conseguem identificar os átomos de confusão; ii) Um prompt mais específico, com as possibilidades atômicas devidamente enumeradas no extenso dataset utilizado. Destarte, à luz desses fatores, espera-se estabelecer métricas precisas para a avaliação da proficuidade dessa ferramenta na identificação dos meandros que potencialmente tornam um software menos compreensível e de menor qualidade.

Palavras-chave

Átomos de confusão. Gemini. Qualidade de Software. Processamento de linguagem natural. Oportunidades de refatoração. Manutenção de código.

ABSTRACT

This study aims to describe initial insights regarding the effectiveness of a popular generative artificial intelligence, Gemini, in identifying *atoms of confusion*. To this end, two distinct prompts will be analyzed, namely: (i) a generic prompt, intended to assess whether both aforementioned systems are capable of identifying atoms of confusion; and (ii) a more specific prompt, in which the atomic possibilities are explicitly enumerated based on the extensive dataset employed. Therefore, in light of these factors, it is expected that precise metrics can be established to evaluate the proficiency of this tool in identifying the intricacies that potentially reduce software comprehensibility and overall quality.

Keywords
(LANGHOUT; ANICHE, 2021)Atoms of confusion. Gemini. Software Quality. Natural Language Processing. Refactoring Opportunities. Code Maintenance.

1 INTRODUÇÃO

Hodiernamente, os Large Language Models (LLMs), como o Gemini, por exemplo, são amplamente utilizados para automatizar tarefas nos mais diversos domínios, com destaque para a engenharia de software (ALSHAHWAN et al., 2024). Sua capacidade de compreender, gerar e analisar código-fonte abre novas possibilidades para atividades como manutenção, teste, refatoração e detecção de padrões.

Nesse aspecto, é clarividente para desenvolvedores o fato de que diferentes linguagens de programação, bem como diferentes estilos dentro de uma mesma linguagem, podem oferecer múltiplas formas de resolver um mesmo problema. Um exemplo dessa diversidade pode ser observado nas respostas à pergunta “Como converter um valor booleano para inteiro em Java?”, publicada no StackOverflow. Na Figura 1, os autores apresentam oito soluções distintas ordenadas pelo número de votos, revelando uma variação considerável em termos de lógica, legibilidade e compreensibilidade. A opção que utiliza o operador ternário (opção 1) foi a mais votada e considerada a mais legível pelos comentaristas, sendo, portanto, a solução mais aceita pela comunidade. Contudo, tal preferência entra em contraste com as evidências apresentadas por Gopstein et al. (2017), que indicam que o uso do operador condicional pode ser significativamente confuso. Ainda mais curioso é o fato de que a oitava resposta inicia com a frase “Se você quer ofuscar, use isto:”, sugerindo explicitamente que sua intenção não é promover legibilidade, mas sim oferecer uma alternativa pouco usual para resolver o problema (LANGHOUT; ANICHE, 2021).

Nesse contexto, Langhout e Aniche (2021) observaram que muitos erros em software não decorrem de algoritmos complexos, mas sim de trechos pequenos e sutis de código que são mal compreendidos por desenvolvedores. Esses trechos são

chamados de átomos de confusão, pequenas porções de código que induzem a interpretações equivocadas. Castor, por sua vez, ampliou esse conceito ao estabelecer que tais átomos devem ser identificáveis, propensos à confusão, substituíveis por alternativas mais claras e funcionalmente indivisíveis.

Diante disso, o presente trabalho tem como objetivo investigar a eficácia do Gemini na identificação de átomos de confusão em códigos Java, utilizando para tal um conjunto de dados específico: o *Atoms of Confusion Dataset in Java Programs* (2022), que contém exemplos anotados de diversos padrões confusos recorrentes em código. A análise será conduzida com base em dois prompts distintos: um genérico e outro específico, formulado a partir das boas práticas de *prompt engineering*, com o intuito de explorar diferentes abordagens de interação com o modelo.

O dataset utilizado apresenta exemplos de 14 tipos distintos de átomos de confusão, tais como: Precedência de Operadores Infixos, Pós- e Pré-Incremento/Decremento, Variáveis Constantes, Remoção de Identação, Operador Condicional, Aritmética como Lógica, Lógica como Fluxo de Controle, Reutilização de Variáveis, Código Morto ou Repetido, Mudanças na Codificação Literal, Colchetes Omitidos, Conversão de Tipo e problemas relacionados à Identação. Esses elementos serão fundamentais para a avaliação da proficiência do modelo na detecção de padrões que afetam negativamente a legibilidade e a manutenibilidade do código.

Com isso, busca-se responder às seguintes perguntas de pesquisa, adaptadas do trabalho de Silva et al. (2024), considerando o Gemini como objeto de análise:

- **RQ.1**: O Gemini é capaz de identificar átomos de confusão com um prompt específico, baseado em engenharia de prompting?
- **RQ.2**: O Gemini consegue identificar átomos de confusão utilizando um prompt genérico?
- **RQ.3**: Há diferença na eficácia do Gemini entre o uso de prompt genérico e específico?
- **RQ.4 (exploratória)**: A frequência de ocorrência de determinados átomos de confusão influencia as métricas de *precision* e *recall* na classificação realizada pelo Gemini?

Figura 1 – Respostas à pergunta “Como converter booleano para int em Java?” no StackOverflow

```
1. i = b ? 1 : 0;

2. i = (Boolean b).compareTo(false);

3. i = Boolean.compare(b, false);

4. i = -("false".indexOf(""+b));

5. i = 5 - b.toString().length;

6. import org.apache.commons.lang3.
                BooleanUtils;
   i = BooleanUtils.toInteger(b);

7. if(b){
      i = 1;
   } else{
      i = 0;
   }

8. i = 1 & Boolean.hashCode( b ) >> 1;
```

Fonte: LANGHOUT; ANICHE (2021).

2 METODOLOGIA

2.1 Dataset

Para este trabalho, foi utilizado os dados presentes em um dataset (ATOMS of Confusion Dataset in Java Programs, 2022) que contém informações como a classificação do átomo de confusão presente, o link para o código java no github, o snippet do código e a linha onde está localizado, precisamente, a partícula em questão. Assim, todos os 14 átomos enumerados por Gopstein et al estão presentes em algum nível nos códigos enumerados neste dataset.

A saber, os átomos de código presentes no dataset são:

- **Precedência de Operadores Infixos:** Exemplos que demonstram como a

precedência dos operadores pode afetar o resultado das operações aritméticas.

- **Pós-Incremento/Decremento:** Casos que mostram o uso dos operadores de

incremento e decremento após a variável, e as implicações no valor da variável.

- **Pré-Incremento/Decremento:** Exemplos que ilustram o uso dos operadores de

incremento e decremento antes da variável, e como isso altera o valor da variável

antes da expressão ser avaliada.

- **Variáveis Constantes:** Casos que demonstram a utilização de variáveis constantes

e as diferenças entre a atribuição direta e a alteração de valores constantes.

- Remoção de Identação: Exemplos que ilustram a remoção de identação e o impacto

na clareza do código.

- **Operador Condicional:** Demonstração do uso do operador condicional ternário e

sua equivalência com uma estrutura condicional mais explícita.

- **Aritmética como Lógica:** Casos que mostram a utilização de operações aritméticas

como substituto para expressões lógicas e sua interpretação.

- **Lógica como Fluxo de Controle:** Exemplos que mostram a utilização de

expressões lógicas para controle de fluxo e as possíveis confusões que podem surgir.

- **Variáveis Reutilizadas**: Exemplos que demonstram o uso de variáveis em loops e

como a reutilização pode afetar a legibilidade e o comportamento do código.

- **Código Morto, Inalcançável, Repetido:** Casos que ilustram código que não é

executado ou é repetido de maneira desnecessária, e como isso pode ser removido para melhorar a eficiência.

- **Mudança na Codificação Literal:** Exemplos que mostram como a mudança na representação literal de valores pode impactar o código.
- **Colchetes Omitidos:** Casos onde os colchetes são omitidos em estruturas condicionais e o impacto na execução do código.
- **Conversão de Tipo:** Exemplos que ilustram a conversão de tipos de dados e como diferentes métodos de conversão podem ser aplicados.
- **Identação:** Casos que mostram como a identação correta ou incorreta pode impactar a legibilidade e a execução do código.

2.2 Justificativa da Escolha do Modelo

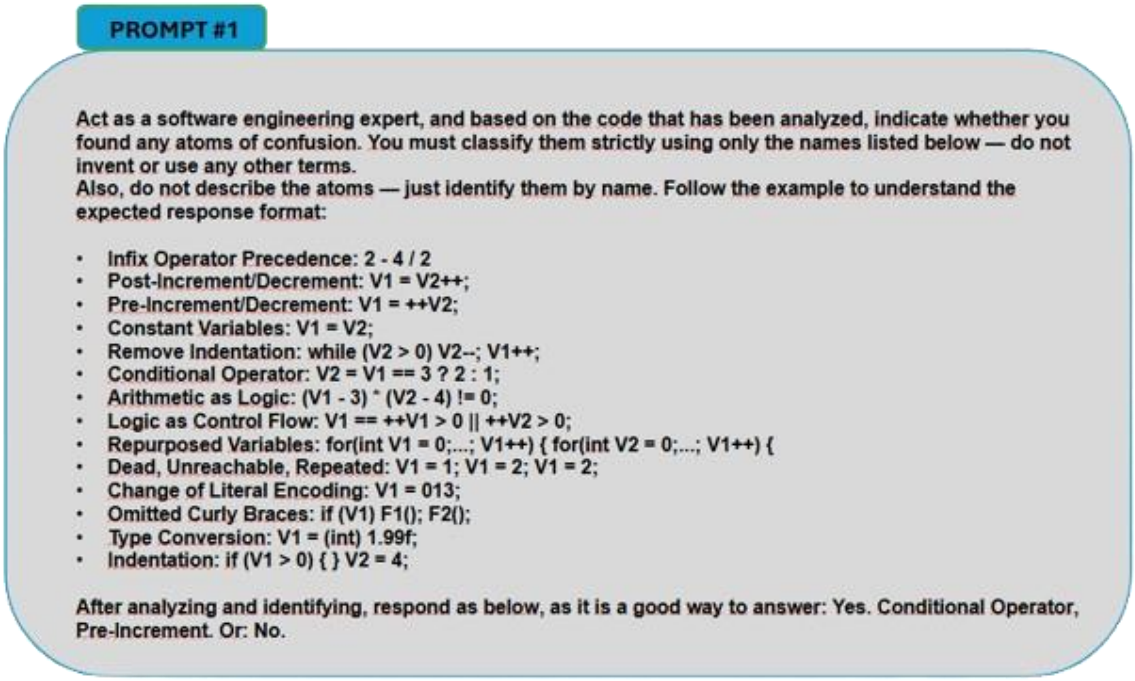
A escolha pelo uso do modelo Gemini, da Google, fundamenta-se em critérios técnicos e funcionais alinhados às demandas do presente estudo. Inicialmente, observou-se que as classes do dataset utilizado apresentam, no máximo, 31.360 caracteres, volume compatível com os limites de entrada da versão gratuita do Gemini, que se mostram superiores aos da versão gratuita do ChatGPT, permitindo, assim, a análise completa dos dados sem a necessidade de truncamento ou segmentação. Além disso, conforme discutido por Rane, Choudhary e Rane (2024), o ChatGPT é um modelo voltado principalmente à geração imaginativa de texto, o que o torna mais indicado para aplicações conversacionais. O Gemini, por outro lado, apresenta maior integração com a infraestrutura do Google, o que favorece uma maior precisão factual e o posiciona como uma ferramenta mais robusta para fins de análise de conhecimento e extração de informações. Tais características foram decisivas para a adoção do Gemini como modelo de linguagem base nesta pesquisa.

2.3 Prompts para o Gemini

Para a realização da pesquisa, foram elaborados dois prompts: um genérico e um específico, seguindo os padrões de prompt engineering, ou seja, o segundo é feito de forma livre, menos elaborado e não induz a um direcionamento mais consciente da resposta esperada. Já o primeiro prompt, apresenta algumas técnicas de engenharia de prompt como (Prompting Guide, 2023):

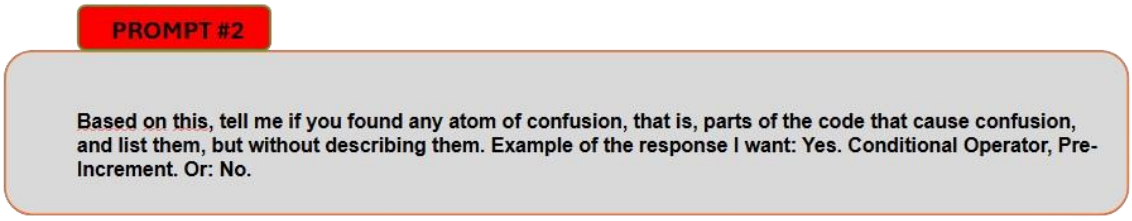
- **Estrutura P.R.O.M.P.T**: o prompt foi elaborado com clareza de persona, roteiro, objetivo, modelo de resposta e panorama, o que orienta o modelo sobre como responder e com qual finalidade.
- **Fornecimento de Exemplos**: foram incluídos exemplos e referências de respostas esperadas, ajudando o modelo a produzir resultados mais alinhados ao desejado.
- **Estímulos Direcionais**: o prompt foi escrito com instruções específicas e palavras-chave, guiando a resposta para atender melhor aos objetivos da tarefa.

Figura 1: Prompt específico, com exemplos e técnicas de Prompt engineering



Fonte: Elaborado pelo autor (2025).

Figura 2: Prompt genérico.



Fonte: Elaborado pelo autor (2025).

Ademais, é relevante salientar que, inicialmente, foi inserido o código a ser analisado e, em seguida, apresentadas as questões propostas no prompt. Essa estrutura favorece uma análise mais eficiente por parte do modelo, resultando em respostas mais precisas e contextualizadas.

No entanto, dos 489 trechos de código presentes no dataset — os quais pertencem a bibliotecas Java amplamente reconhecidas e utilizadas em aplicações industriais, como a classe Arrays da biblioteca *fastutil* — apenas 319 encontram-se atualmente disponíveis. Isso se deve ao fato de que os códigos estão hospedados no GitHub e, em virtude de atualizações, reestruturações ou remoções, parte deles se tornou indisponível. Consequentemente, ao executar o processo de *scraping* por meio de um script escrito em linguagem Go (Golang), utilizando os links fornecidos no dataset, 170 desses códigos resultaram em erro de status 404.

Outrossim, é imprescindível mencionar que, considerando a classificação de átomos supracitada e levando em conta apenas os arquivos efetivamente acessíveis por meio dos links ainda ativos, a proporção observada é a seguinte:

Tabela 1: Proporção de átomos presentes no dataset por classificação.

Átomo	Quantidade	Frequência relativa
Change of Literal Encoding	9	2.8%
Logic as Control Flow	98	30.7%
Pre Increment Decrement	11	3%
Conditional Operator	80	25%
Omitted Curly Braces	1	0,3%
Post Increment Decrement	32	10%
Type Conversion	40	12.6%
Arithmetic as Logic	4	1.3%
Infix Operator Precendece	44	13.8%

Fonte: Elaborado pelo autor (2025).

2.4 Arquitetura do projeto

O projeto foi desenvolvido utilizando a linguagem de programação Go (Golang), escolhida por suas poderosas funcionalidades nativas para programação concorrente e paralela. Essa escolha permitiu uma implementação eficiente da integração com a

API do Gemini, respeitando as restrições de taxa de requisição (rate limiting) impostas pela API.

Para isso, foi implementado um pool de workers responsável pelo processamento concorrente das tarefas, com mecanismos de controle para evitar condições de corrida (race conditions) e suporte a pausas globais no processamento — ativadas automaticamente ao atingir o limite de requisições permitido. Essa arquitetura garantiu maior desempenho e estabilidade na comunicação com a API.

Como se não bastasse, o projeto também adotou o modelo arquitetural conhecido como **Arquitetura Limpa (Clean Architecture)**. Essa abordagem é amplamente utilizada no mercado de trabalho devido às suas diversas vantagens, especialmente no que se refere à **manutenibilidade**, **testabilidade** e **organização** do código.

Entre os principais benefícios, destacam-se:

- **Separação de responsabilidades:** o código é dividido em camadas bem definidas (como domínio, casos de uso, interfaces e infraestrutura), o que facilita entender e modificar partes específicas do sistema sem afetar o todo.
- **Facilidade de testes unitários:** com a lógica de negócio isolada em camadas independentes de frameworks, torna-se simples criar testes automatizados de forma confiável e eficiente.
- **Independência de frameworks e tecnologias:** a aplicação não depende diretamente de bibliotecas externas. Por exemplo, é possível trocar o banco de dados, o framework web ou o provedor de nuvem com impacto mínimo na regra de negócio.
- **Alta escalabilidade e flexibilidade:** novas funcionalidades podem ser adicionadas sem comprometer a estrutura existente, permitindo que o sistema evolua com segurança.
- **Reutilização de código:** regras de negócio e entidades centrais podem ser reaproveitadas por diferentes interfaces, como APIs, CLIs ou interfaces gráficas, sem duplicação.

"A intenção da arquitetura limpa é separar o código de forma que os detalhes sejam isolados da lógica de negócio. Isso permite que o sistema evolua com mudanças nos detalhes sem afetar suas regras essenciais." (MARTIN, 2018, p. 23).

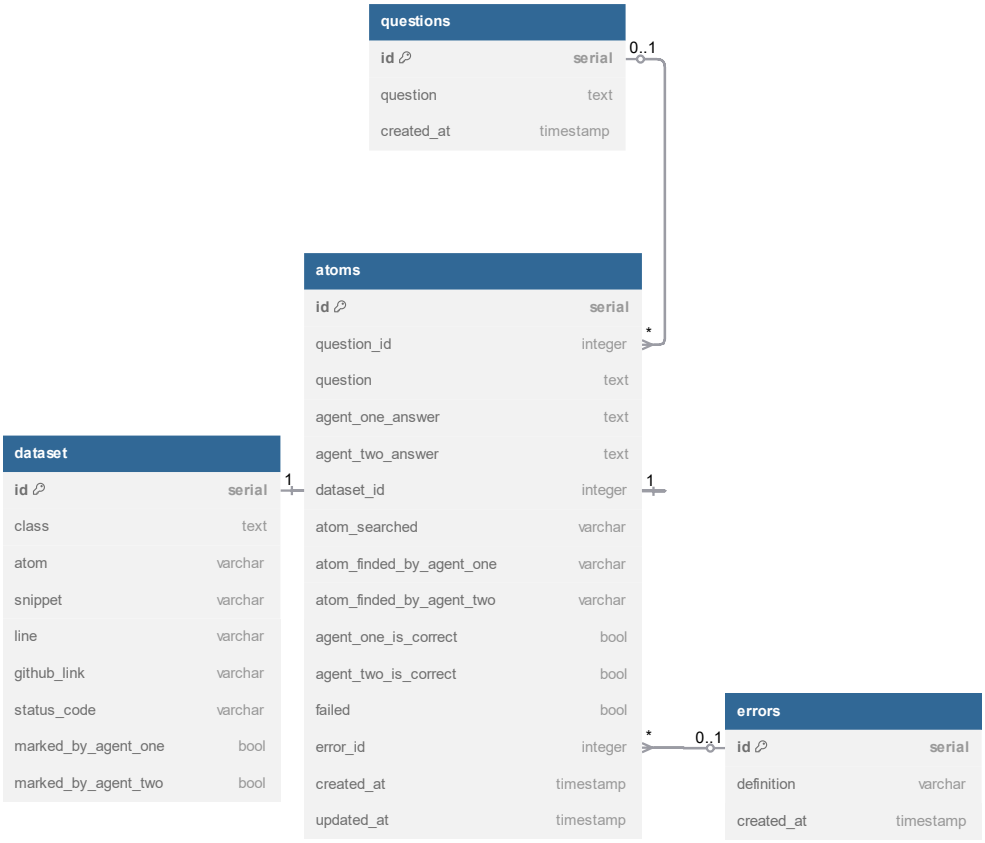
2.5 Banco de dados

O banco de dados **PostgreSQL** foi adotado como solução para o armazenamento de dados do projeto, incluindo **resultados processados, erros capturados, conjuntos de dados (datasets)** e os **prompts utilizados nas requisições**. A escolha se justifica pela robustez e flexibilidade do PostgreSQL, que oferece suporte avançado para consultas complexas, integridade referencial e fácil integração com aplicações escritas em Golang.

Além disso, o uso de um banco relacional permite **facilidade na análise dos dados** armazenados e contribui significativamente para **o monitoramento e rastreamento das interações realizadas com a API**, o que foi essencial para avaliar a qualidade e eficiência do sistema.

A **Figura 3** a seguir apresenta o diagrama relacional com as **principais tabelas e suas relações de dependência**, ilustrando como os dados são organizados e conectados dentro da base.

Figura 3: Diagrama do banco de dados



Fonte: Elaborado pelo autor (2025).

2.6 Resposta do Gemini

Foram utilizados os parâmetros padrão de **temperatura igual a 1**, o que representa um equilíbrio entre a geração determinística e criativa de respostas por parte do modelo. Com isso, as respostas tenderam a manter-se próximas do assunto principal, mas com certa variabilidade na forma de apresentação. Na maioria dos casos, as respostas seguiram o formato solicitado, como, por exemplo:

"Yes. Indentation, Omitted Curly Braces, Repurposed Variables."

Para garantir a extração adequada dos elementos relevantes nas respostas — como os nomes dos *code smells* — foi empregada a técnica de **expressões regulares (regex)**, a fim de filtrar e padronizar os termos retornados pelo modelo com maior precisão.

Figura 4 – Padrão Regex utilizado para extração de *atoms of confusion* nas respostas do modelo e para filtro de partes desnecessárias nos códigos enviados.

```
const (
 SINGLE_COMMENTS string = `(?m)//[^\n]*\n`
 MULTI_COMMENTS string = `(?s)/\*.*?\*/`
 IMPORTS string = `(?m)^import\s+[^\n]+;\s*`
 HEADERS string = `(?m)^package\s+[^\n]+;\s*`
 ANSWER  string = `(?i)\byes\.\s+[A-Za-z ]+(?:,\s*[A-Za-z ]+)*\.`
)
```

3 JUSTIFICATIVA PARA AS MÉTRICAS UTILIZADAS

Para avaliar o desempenho do sistema proposto, foram utilizadas métricas amplamente adotadas nas áreas de **Recuperação da Informação** e **Aprendizado de Máquina**, conforme abordagem similar à apresentada por Silva et al. (2024) no estudo sobre detecção de *code smells* com auxílio do ChatGPT. As métricas escolhidas foram:

- **Precisão (Precision)**: mede a proporção de verdadeiros positivos entre todos os casos identificados como positivos pelo sistema. Ou seja, quantifica **o quanto as detecções feitas são realmente corretas**.
- **Revocação (Recall)**: avalia a capacidade do sistema em identificar corretamente os casos positivos existentes no conjunto de dados, isto é, **quantos dos casos relevantes foram efetivamente detectados**.
- **F1-Score (F-measure)**: é a **média harmônica entre precisão e revocação**, proporcionando um equilíbrio entre essas duas métricas e permitindo uma avaliação mais abrangente do desempenho.

Essas métricas são aplicadas aos resultados obtidos com o intuito de medir a **efetividade da abordagem adotada** e permitir comparações consistentes com estudos relacionados.

4 RESULTADOS

4.1 O Gemini consegue identificar átomos de confusão utilizando um prompt genérico?

Para responder a essa pergunta, foi necessário utilizar uma **matriz de confusão**, que permite visualizar o desempenho do modelo na identificação dos átomos. Essa matriz resume as previsões corretas e incorretas em comparação com os valores reais, fornecendo uma visão clara da acurácia do modelo.

Nesse contexto, considerando os cálculos realizados a partir do **primeiro prompt** — que é mais específico e segue os princípios da engenharia de prompt — observou-se um desempenho superior em relação ao **segundo prompt**, de natureza mais genérica, que apresentou uma quantidade ínfima de acertos. Diante disso, a Tabela 2 apresenta a seguinte relação:

Tabela 2: Resultados do primeiro prompt, o específico.

Átomo	Quantidade Esperada	Quantidade encontrada	Precision	Recall	F-meansure
Change of Literal Encoding	9	0	0.0	0.0	0.0
Logic as Control Flow	98	1	0.25	0.0102	0.009800154
Pre Increment Decrement	11	0	0.0	0.0	0.0
Conditional Operator	80	17	0.4473	0.2125	0.1440607
Omitted Curly Braces	1	0	0.0	0.0	0.0

Post Increment Decrement	32	0	0.0	0.0	0.0
Type Conversion	40	20	0.1739	0.5	0.129025078
Arithmetic as Logic	4	2	0.2857	0.5	0.181812397
Infix Operator Precendece	44	5	0.2631	0.1136	0.079342076

Fonte: Elaborado pelo autor (2025).

Portanto, para se alcançar esses resultados, os cálculos foram realizados com base nas seguintes fórmulas, considerando os casos de verdadeiro positivo, falso negativo e falso positivo:

$$Recall = \frac{VP}{VP + FN}$$

$$precision = \frac{VP}{VP + FP}$$

$$Fmeansure = 2 \times \frac{Precision\ x\ Recall}{Precision + Recall}$$

Vale salientar que os valores quantitativos de falsos positivos foram obtidos a partir da quantidade de vezes em que o Gemini identificou incorretamente um átomo, atribuindo-o a outro. Essa identificação foi realizada por meio da seguinte consulta:

Figura 5: Quantidade de falsos positivos.

```
-- falsos positivos.
SELECT
  SUM(CASE WHEN LOWER(atom_finded) LIKE LOWER('%Type Conversion%') THEN 1 ELSE 0 END) - 20 AS total_type_conversion,
  SUM(CASE WHEN LOWER(atom_finded) LIKE LOWER('%Conditional Operator%') THEN 1 ELSE 0 END) - 17 AS total_conditional_operator,
  SUM(CASE WHEN LOWER(atom_finded) LIKE LOWER('%Infix Operator Precedence%') THEN 1 ELSE 0 END) - 5 AS total_infix_operator_precedence,
  SUM(CASE WHEN LOWER(atom_finded) LIKE LOWER('%Arithmetic as Logic%') THEN 1 ELSE 0 END) - 2 AS total_arithmetic_as_logic,
  SUM(CASE WHEN LOWER(atom_finded) LIKE LOWER('%Logic as Control Flow%') THEN 1 ELSE 0 END) - 1 AS total_logic_as_control_flow
FROM
  atoms
WHERE
  question_id = 1;
```

total_type_conversion	total_conditional_operator	total_infix_operator_precedence	total_arithmetic_as_logic	total_logic_as_control_flow
115	21	14	5	3

Fonte: Elaborado pelo autor (2025).

Observa-se, com base nos **resultados** obtidos, que o desempenho do Gemini na identificação de átomos de confusão em códigos Java é **insatisfatório**. A acurácia ficou significativamente abaixo de 50%, o que compromete sua efetividade como ferramenta para aprimorar a detecção de trechos potencialmente confusos no trabalho dos desenvolvedores.

4.1 O Gemini consegue identificar átomos de confusão utilizando um prompt genérico?

Após a análise inicial da capacidade do modelo em identificar corretamente os átomos, procedeu-se à avaliação da segunda questão proposta. Ainda com o auxílio da matriz de confusão, buscou-se compreender de forma mais ampla o comportamento geral do modelo diante das diferentes categorias envolvidas, verificando sua assertividade e os possíveis padrões de erro que poderiam impactar sua aplicabilidade prática. Abaixo, o resultado na figura 6:

Tabela 3: Resultados do segundo prompt, o genérico.

Átomo	Quantidade Esperada	Quantidade encontrada	Precision	Recall	F-meansure
Change of Literal Encoding	9	0	0.0	0.0	0.0
Logic as Control Flow	98	0	0.0	0.0	0.0
Pre Increment Decrement	11	0	0.0	0.0	0.0
Conditional Operator	80	4	0.8	0.05	0.094
Omitted Curly Braces	1	0	0.0	0.0	0.0

Post Increment Decrement	32	0	0.0	0.0	0.0
Type Conversion	40	1	1.0	0.025	0.049
Arithmetic as Logic	4	0.0	0.0	0.0	0.0
Infix Operator Precendece	44	0.0	0.0	0.0	0.0

Fonte: Elaborado pelo autor (2025).

A avaliação do modelo Gemini na identificação de "átomos de confusão" em códigos Java, utilizando um prompt genérico, revelou um desempenho geral significativamente limitado. Conforme detalhado na Tabela 3, a capacidade do modelo em encontrar corretamente as diversas categorias de átomos foi, na maioria dos casos, nula, resultando em valores de Recall e F-measure iguais a zero para categorias como "Change of Literal Encoding", "Logic as Control Flow", "Pre Increment Decrement", "Omitted Curly Braces", "Post Increment Decrement", "Arithmetic as Logic" e "Infix Operator Precendece".

Mesmo para os átomos onde houve alguma detecção, como "Conditional Operator" (F-measure de 0.094) e "Type Conversion" (F-measure de 0.049), os baixos valores de Recall (0.05 e 0.025, respectivamente) indicam que o modelo falha em identificar a vasta maioria das ocorrências reais. Embora a Precisão para esses dois casos específicos tenha sido mais alta (0.8 e 1.0, respectivamente, considerando os Falsos Positivos informados), sugerindo que as poucas identificações feitas tendem a estar corretas, a incapacidade de encontrar um volume representativo dos átomos compromete severamente a utilidade da ferramenta.

Portanto, respondendo à questão central, o Gemini, quando empregado com um prompt genérico, não demonstra ser eficaz na identificação da maioria dos "átomos de confusão" analisados.

4.2 Há diferença na eficácia do Gemini entre o uso de prompt genérico e específico?

há uma diferença significativa na eficácia do modelo Gemini ao se comparar o uso de prompts específicos com prompts genéricos.

Conforme os resultados apresentados na **Tabela 2**, quando o modelo foi orientado por um prompt específico — elaborado com base em princípios de engenharia de prompt e com foco claro nas categorias de átomos de confusão —, seu desempenho foi visivelmente superior. Foram observados casos com **recall considerável** (até 0,5 para *Type Conversion* e *Arithmetic as Logic*), além de uma **maior variedade de átomos corretamente identificados**, indicando uma compreensão mais adequada do modelo diante de instruções claras e direcionadas.

Em contraste, os resultados obtidos com o prompt genérico (Tabela 3) mostram uma queda acentuada nas métricas de desempenho, com a maioria dos átomos não sendo reconhecida, e com **valores nulos ou muito baixos** nas principais métricas de avaliação.

Assim, pode-se afirmar que a eficácia do Gemini na identificação de átomos de confusão **está diretamente relacionada à qualidade e especificidade do prompt fornecido**. O uso de prompts genéricos limita severamente a performance do modelo, ao passo que prompts bem estruturados e orientados potencializam significativamente sua capacidade de análise, assim, possivelmente, torna-se indispensável o uso de cadeias de pensamentos e few-shots.

4.2 A frequência de ocorrência de determinados átomos de confusão influencia as métricas de *precision* e *recall* na classificação realizada pelo Gemini?

A frequência com que diferentes classes ou categorias ocorrem em um conjunto de dados é um fator reconhecidamente capaz de influenciar o desempenho de modelos de classificação, impactando diretamente métricas como Precisão (Precision) e Revocação (Recall). Modelos podem, por exemplo, apresentar viés em direção a classes majoritárias ou ter dificuldade em aprender as características distintivas de classes minoritárias.

Para esta análise, em questão, recorreu-se primariamente aos resultados obtidos com o prompt específico (Tabela 2), uma vez que esta abordagem resultou

em um maior número de detecções em diversas categorias, permitindo uma observação mais detalhada de possíveis correlações.

Quanto a Tabela 2:

- O átomo "Logic as Control Flow", com a maior frequência esperada (QE=98), apresentou um Recall (0.0102) e Precisão (0.25) notadamente baixos. Isso indica que, mesmo sendo uma categoria abundante, o modelo demonstrou grande dificuldade em identificá-la corretamente.
- "Conditional Operator", o segundo átomo mais frequente (QE=80), obteve um Recall de 0.2125 e Precisão de 0.4473. Embora superiores aos de "Logic as Control Flow", esses valores ainda são modestos, sugerindo que a alta frequência, por si só, não garantiu uma detecção robusta.
- Em contraste, "Type Conversion" (QE=40) e "Arithmetic as Logic" (QE=4), com frequências média e baixa respectivamente, apresentaram o mesmo valor de Recall (0.5) com o prompt específico. Este foi o maior Recall observado entre os átomos que tiveram múltiplas ocorrências reais e alguma detecção, superando o Recall de átomos mais frequentes. No entanto, suas Precisões (0.1739 para Type Conversion e 0.2857 para Arithmetic as Logic) foram baixas, indicando um número considerável de falsos positivos em relação aos verdadeiros positivos.
- "Infix Operator Precendece" (QE=44), com frequência média, também apresentou baixos valores de Recall (0.1136) e Precisão (0.2631).
- Para os átomos com frequência muito baixa (e.g., "Omitted Curly Braces" com QE=1) que não foram detectados, a análise da influência da frequência no acerto é inviabilizada.

No contexto do prompt genérico (Tabela 3), a maioria dos átomos não foi detectada, tornando a análise da influência da frequência ainda mais restrita. Para os dois átomos que tiveram alguma detecção:

- "Conditional Operator" (QE=80) apresentou Recall extremamente baixo (0.05), mas uma Precisão elevada (0.8).

- "Type Conversion" (QE=40) também teve Recall muito baixo (0.025), mas Precisão perfeita (1.0). Nestes casos, a alta Precisão concomitante a um baixo Recall, em átomos com frequência considerável, pode sugerir que o modelo atuou de forma excessivamente conservadora, classificando um átomo como positivo apenas em cenários de alta confiança, mas falhando em generalizar para a maioria das ocorrências.

Logo, os dados apresentados **não revelaram uma correlação direta e inequívoca entre a frequência de ocorrência dos "átomos de confusão" e o desempenho do modelo Gemini**, medido por Precisão e Recall. Contudo, observou-se que átomos de alta frequência, como "Logic as Control Flow", puderam apresentar desempenho inferior (especialmente em Recall) quando comparados a átomos de frequência média ou baixa, como "Type Conversion" e "Arithmetic as Logic" (particularmente com o prompt específico).

Este panorama sugere que outros fatores, para além da simples frequência, exerceram um impacto mais determinante na capacidade de identificação do modelo. Entre esses fatores, destacam-se:

1. A complexidade intrínseca e a sutileza das características definidoras de cada "átomo de confusão": Alguns padrões podem ser inerentemente mais difíceis para o modelo aprender e detectar, independentemente de sua frequência.
2. A adequação e especificidade do prompt: Conforme discutido na seção anterior (4.2), a engenharia do prompt demonstrou ser um fator crucial, com o prompt específico resultando em uma maior capacidade de detecção geral.
3. A capacidade de generalização do modelo Gemini: A habilidade do modelo em generalizar a partir dos exemplos implícitos em sua base de treinamento para os padrões específicos dos átomos de confusão investigados.

Assim, é importante ressaltar que o número total de "átomos encontrados" foi relativamente baixo em diversas categorias, especialmente com o prompt genérico. Esta escassez de detecções positivas limita a possibilidade de se estabelecerem conclusões estatisticamente robustas acerca do impacto isolado da frequência de ocorrência nas métricas de desempenho.

Ademais, embora a frequência das classes seja um aspecto classicamente relevante na avaliação de modelos de classificação, no presente estudo sobre a identificação de "átomos de confusão" em códigos Java pelo modelo Gemini, ela não se manifestou como o principal modulador do desempenho em termos de Precisão e Recall. Fatores relacionados à natureza dos próprios átomos e, fundamentalmente, à formulação do prompt, aparetaram ter desempenhado um papel mais proeminente.

4.3 Discussão

Nesta seção, são discutidas as implicações práticas e teóricas dos resultados apresentados, bem como as potenciais ameaças à validade que podem ter influenciado as conclusões deste estudo.

4.3.1 Implicações dos Resultados

Os resultados obtidos, embora indiquem um desempenho insatisfatório do Gemini na tarefa, oferecem percepções valiosas para diferentes públicos:

- **Para Desenvolvedores:** A principal implicação é que não se deve confiar em LLMs de prateleira, como o Gemini, com prompts genéricos para a detecção de padrões de código sutis como os átomos de confusão. A eficácia da ferramenta está diretamente ligada à habilidade de construir prompts específicos e bem-estruturados, fornecendo contexto, exemplos e uma persona clara para o modelo. Mesmo assim, os resultados devem ser revisados com cautela.
- **Para Pesquisadores:** O estudo demonstra que, apesar da capacidade avançada dos LLMs, a detecção de "confusão" no código-fonte, que depende de um profundo entendimento semântico, continua sendo um desafio. A falta de correlação direta entre a frequência de um átomo e a performance do modelo sugere que futuras pesquisas devem focar menos na distribuição estatística e mais na complexidade intrínseca de cada padrão de código.
- **Para Criadores de Ferramentas de IA:** Os resultados servem como um benchmark inicial, indicando que modelos de propósito geral ainda não estão otimizados para tarefas de nicho na engenharia de software. Há uma clara oportunidade para o desenvolvimento de modelos especializados ou o ajuste

fino (*fine-tuning*) de modelos existentes com datasets de qualidade de código para criar ferramentas de análise mais precisas e confiáveis**.**

4.3.2 Ameaças à Validade

É fundamental reconhecer as limitações deste estudo que podem influenciar a interpretação dos resultados:

- **Ameaças à Validade Externa (Generalização):**
    - **Seleção do Modelo:** O estudo foi limitado ao uso do Gemini. Os resultados podem não ser generalizáveis para outros LLMs, como o GPT-4 ou Claude, que podem apresentar desempenhos distintos.
    - **Linguagem e Dataset:** A análise foi restrita à linguagem Java e a um único dataset. Os "átomos de confusão" podem se manifestar de formas diferentes em outras linguagens, e os resultados podem não se aplicar a outros tipos de projetos de software.
- **Ameaças à Validade Interna (Fatores do Estudo):**
    - **Disponibilidade dos Dados:** Como explicitamente mencionado, 170 dos 489 trechos de código do dataset original não estavam mais disponíveis. Essa perda de dados, que representa mais de 34% do total, é a principal ameaça interna, pois a distribuição de átomos nos códigos perdidos é desconhecida e poderia alterar os resultados de precisão e revocação.
    - **Desbalanceamento das Classes:** A frequência dos átomos no dataset é altamente desbalanceada, com "Logic as Control Flow" correspondendo a 30.7% dos casos, enquanto "Omitted Curly Braces" representa apenas 0,3%. Isso pode ter enviesado o modelo, dificultando o aprendizado das características de classes minoritárias.
- **Ameaças à Validade de Confiabilidade (Replicação):**
    - **Natureza do LLM:** Os LLMs são modelos em constante evolução. A versão do Gemini utilizada neste trabalho pode ser atualizada, e execuções futuras com os mesmos prompts podem gerar resultados diferentes. Além disso, o uso de temperatura=1 introduz uma

variabilidade inerente nas respostas, embora a extração com Regex busque padronizar os resultados.

5 TRABALHOS RELACIONADOS

O presente trabalho fundamenta-se em investigações prévias relacionadas à compreensão e interpretação de código-fonte, com ênfase na identificação de padrões que tendem a gerar confusão durante a leitura e análise do código. Especificamente, este estudo baseia-se na pesquisa conduzida por Chris Langhout e Maurício Aniche, da Universidade de Delft(**LANGHOUT**), a qual, por sua vez, tem como referência o estudo seminal realizado por Gopstein et al. Esta pesquisa inicial investigou os chamados *átomos de confusão* no contexto da linguagem C, evidenciando como tais padrões podem contribuir significativamente para o aumento na incidência de erros de programação.

Inspirado por essas descobertas, o presente trabalho propõe a replicação do estudo no contexto da linguagem Java, com o intuito de verificar a aplicabilidade dos mesmos princípios e analisar de que forma esses padrões afetam a precisão e a legibilidade do código, especialmente entre programadores iniciantes. Dessa forma, busca-se aprofundar o entendimento acerca do impacto dos átomos de confusão na prática da programação e oferecer subsídios relevantes para aprimorar a clareza e a compreensão do código-fonte.

6 CONCLUSÃO

O presente trabalho teve como objetivo central investigar a eficácia do modelo de linguagem Gemini na detecção de "átomos de confusão" em código-fonte Java, uma tarefa sutil que impacta diretamente a qualidade e manutenibilidade de software. Para tal, foi conduzido um estudo empírico utilizando um dataset público e duas

abordagens de interação distintas: um prompt genérico e um prompt específico, construído com base em técnicas de engenharia de prompt.

Os resultados indicam que o desempenho geral do Gemini para esta tarefa, mesmo com uma abordagem sofisticada, mostrou-se insatisfatório para aplicação prática imediata, com uma acurácia geral abaixo de 50%. Contudo, a pesquisa revelou uma diferença de eficácia significativa e inequívoca entre os prompts. O prompt específico, claro e direcionado, obteve um desempenho visivelmente superior ao do prompt genérico , que se mostrou praticamente ineficaz na identificação da maioria das categorias de átomos. Esta constatação reforça a premissa de que a qualidade e a especificidade do prompt são fatores determinantes para o sucesso na utilização de LLMs em tarefas complexas de análise de código.

Adicionalmente, a análise não encontrou uma correlação direta entre a frequência de um átomo de confusão no dataset e a capacidade do modelo em detectá-lo. Átomos de alta frequência, como "Logic as Control Flow", apresentaram desempenho inferior a átomos de frequência média ou baixa, como "Type Conversion". Isso sugere que a complexidade intrínseca de cada padrão exerce um impacto mais proeminente no desempenho do modelo do que a simples frequência de ocorrência.

Como limitações, este estudo se concentrou em um único modelo (Gemini) e em um dataset que, conforme reportado, apresentou indisponibilidade em parte de seus dados, o que pode ter impactado as métricas de classes com menor representatividade.

Para trabalhos futuros, sugere-se a expansão desta pesquisa em múltiplas frentes: i) a replicação do estudo com outros modelos de linguagem de ponta, como o GPT-4 e o Claude; ii) a aplicação de técnicas de prompt mais avançadas, como a Cadeia de Pensamentos (*Chain-of-Thought*) e o fornecimento de múltiplos exemplos (*few-shot learning*); e iii) a investigação do impacto do ajuste fino (*fine-tuning*) de modelos especificamente para a tarefa de detecção de átomos de confusão.

Conclui-se, portanto, que a detecção de padrões sutis como os átomos de confusão permanece um desafio para os LLMs atuais em uma abordagem de zero-shot. Embora promissora, a aplicação dessas ferramentas no dia a dia do desenvolvedor para essa finalidade específica ainda depende de avanços

significativos, seja na sofisticação da interação com o modelo ou no treinamento especializado dos mesmos.

REFERÊNCIAS BIBLIOGRÁFICAS

ALSHAHWAN, Nadia et al. Automated unit test improvement using large language models at Meta. *In*: ACM SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), 32., 2024, Porto de Galinhas. **Proceedings**... [S.l.]: ACM, 2024.

ATOMS OF CONFUSION DATASET IN JAVA PROGRAMS. Zenodo, 2022. Disponível em: https://zenodo.org/record/7065842. Acesso em: 8 maio 2025.

AVIDAN, E.; FEITELSON, D. G. Effects of variable names on comprehension: an empirical study. *In*: IEEE/ACM INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), 25., 2017, Buenos Aires. **Anais**... [S.l.]: IEEE, 2017. p. 55-65. Disponível em: https://doi.org/10.1109/ICPC.2017.27. Acesso em: 8 maio 2025.

GOPSTEIN, D. et al. Understanding Misunderstandings in Source Code. *In*: JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, 11., 2017, Paderborn. **Proceedings**... [S.l.]: ACM, 2017.

LANGHOUT, Chris; ANICHE, Maurício. **Atoms of confusion in Java**. 2021. Preprint. Disponível em: https://arxiv.org/abs/2103.05424. Acesso em: 8 maio 2025.

MARTIN, Robert C. **Arquitetura Limpa: o guia do artesão para estrutura e design de software**. Rio de Janeiro: Alta Books, 2018.

PROMPTING GUIDE. **Prompt Engineering Guide**. 2023. Disponível em: https://www.promptingguide.ai/pt. Acesso em: 8 maio 2025.

RANE, Nitin Liladhar; CHOUDHARY, Saurabh P.; RANE, Jayesh. Gemini versus ChatGPT: applications, performance, architecture, capabilities, and implementation. **Journal of Applied Artificial Intelligence**, Mumbai, v. 5, n. 1, p. 69-93, 2024. Disponível em: https://doi.org/10.48185/jaai.v5i1.1052. Acesso em: 8 maio 2025.

SILVA, Luciana Lourdes et al. **Detecting Code Smells using ChatGPT: Initial Insights**. 2024. Preprint. Disponível em: https://doi.org/10.13140/RG.2.2.36634.04802. Acesso em: 8 maio 2025.