Um *pipeline* híbrido para detecção inteligente e confiável de *bugs* e *code smells* em Python

Fernanda Alice Duarte
PPGI - Instituto de Computação
Universidade Federal do Amazonas
Manaus-AM, Brasil
fernanda.duarte@icomp.ufam.edu.br

Rosiane de Freitas
Instituto de Computação
Universidade Federal do Amazonas
Manaus-AM, Brasil
rosiane@icomp.ufam.edu.br

Lucas C. Cordeiro
Department of Computer Science
University of Manchester
Manchester, UK
lucas.cordeiro@manchester.ac.uk

Resumo

A detecção precoce de *bugs* e *code smells* exige equilíbrio entre precisão, cobertura e custo. Em Python, cuja natureza dinâmica dificulta a verificação formal, ferramentas como EVA e PyVeritas traduzem código para C, mas podem perder semântica. O ESBMC-Python supera essa limitação ao verificar diretamente o código via *bounded model checking*, porém com alto custo e necessidade de especificações. Modelos de linguagem (LLMs e SLMs) oferecem compreensão semântica, mas sofrem com falsos positivos e alucinações. Dado o contexto, neste trabalho é proposto um *pipeline* híbrido que integra triagem por LLM, validação estrutural via AST e verificação formal com ESBMC-Python para três categorias de *bugs* e três de *code smells*. Avaliamos cinco modelos (gpt-5.5, claude-sonnet-4-6, gemini-3.1-flash-lite, deepseek-r1:7b e qwen2.5-coder:7b) em 70 funções Python sintéticas, geradas com auxílio de LLM e revisadas manualmente, alcançando precisão de *bugs* entre 0,977 e 1,000 (0,081 em *code smells*) e taxa de redução de ruído entre 0,500 e 1,000, reduzindo o total de falsos positivos de 48 para 2 somando os cinco modelos, com ganhos mais expressivos nos modelos locais. A validação AST eliminou completamente alucinações estruturais, presentes apenas nos modelos locais. Os resultados indicam que a abordagem híbrida aumenta a precisão, oferecendo um caminho prático para integrar garantias formais à revisão de código Python. A linha de base, uso isolado da LLM sem confirmação formal, alcança precisão entre 0,627 e 0,977.

Palavras-chave

Code smells, Detecção de falhas, ESBMC, IA Generativa, Modelos de linguagem, Verificação formal

1 Introdução

A linguagem de programação Python é amplamente empregada em diferentes domínios [11], tornando importante avaliar a qualidade do código produzido, tanto na manutenibilidade quanto na presença de *bugs* que comprometem seu comportamento. Sua natureza dinâmica, porém, dificulta a verificação formal. O ESBMC-Python introduziu suporte a essa verificação via *bounded model checking* (BMC), que explora execuções até um limite finito de passos, detectando violações de asserções, erros aritméticos e acessos inválidos [2]. Paralelamente, modelos de linguagem (LLMs e SLMs) apoiam a revisão de código, mas produzem falsos positivos e justificativas incorretas [1]; modelos locais são mais suscetíveis a alucinações estruturais.

Diante desse cenário, a contribuição deste trabalho é um *pipeline* híbrido no qual um modelo de linguagem realiza a triagem inicial de funções Python, classificando potenciais *bugs* e *code smells*; uma etapa de validação estrutural verifica a consistência das hipóteses geradas; e o ESBMC-Python confirma formalmente apenas os *bugs* verificáveis. Este trabalho é guiado por duas questões de pesquisa: RQ1, o *pipeline* híbrido aumenta a confiabilidade dos achados da LLM (precisão) sem deixar de detectar os *bugs* reais já identificados por ela (revocação)? RQ2, como o *pipeline* se comporta na detecção de *code smells*, que não passam por confirmação formal? Os resultados confirmam RQ1, elevando a precisão para entre 0,977 e 1,000 (Seção 4.4); quanto a RQ2, a precisão varia amplamente entre modelos e categorias, sem o ganho de confiabilidade observado nos *bugs* (Seção 4.4).

2 Trabalhos Relacionados

O ESBMC-Python verifica diretamente programas Python anotados com tipos, sem tradução para outra linguagem [2]. Em contraste, o PyVeritas transpila o código para C via LLM e verifica com CBMC [6]; de forma semelhante, o EVA traduz para C via LLM antes de usar o ESBMC tradicional [8]. Este trabalho usa a LLM apenas para triagem, preservando o Python original para verificação direta: evita a perda de fidelidade da tradução, ao custo de ficar restrito ao subconjunto já suportado pelo ESBMC-Python.

Modelos de linguagem têm resultados promissores na análise de código, mas ainda produzem falsos positivos e justificativas incorretas [1]. O *Bounded Model Checking* confirma ou refuta propriedades com contraexemplos, mas enfrenta custo computacional. Tihanyi et al. [10] defendem abordagens híbridas que combinam a cobertura das LLMs com a confiabilidade da verificação formal.

Modelos de linguagem também identificam *code smells*: Santos et al. [7] apontam atributos estruturais compartilhados entre predição de defeitos e *code smells*; Souza et al. [9] observam desempenho variável por tipo de *smell*; Gheyi et al. [4] mostram que SLMs têm desempenho instável entre modelo e tarefa, instabilidade também observada aqui (Seção 4.4). Este trabalho combina triagem por LLM, validação por AST e confirmação formal sem tradução para outra linguagem, combinação não observada nos trabalhos aqui discutidos.

3 Pipeline Híbrido

O *pipeline* proposto recebe como entrada uma função Python e produz um relatório contendo *bugs* verificáveis confirmados formalmente e *code smells* identificados pelo modelo de linguagem. Conforme ilustrado na Figura 1, a arquitetura é composta por cinco etapas: *Preprocessing*, *Analyzer*, *AST Validation*, ESBMC-Python e *Final Report*. Após a etapa de análise pelo modelo de linguagem, os achados são separados em dois fluxos: hipóteses de *bugs* verificáveis são submetidas à validação estrutural por AST e, posteriormente,

ao ESBMC-Python para confirmação formal, enquanto *code smells* seguem diretamente para o relatório final.

Nesta avaliação preliminar, o *pipeline* foi restrito a seis categorias: três *bugs* verificáveis formalmente (*division by zero*, *out-of-bounds access* e *assertion violation*) e três *code smells* amplamente reconhecidos na literatura de refatoração [3] (*long method*, *many parameters* e *complex conditional*).

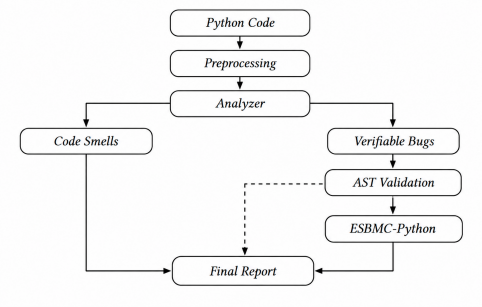

Figura 1: Pipeline híbrido para análise de funções Python, em cinco etapas: (1) *Preprocessing*, (2) *Analyzer*, (3) *AST Validation*, (4) ESBMC-Python, (5) *Final Report*. *Code smells* seguem direto ao *Final Report*; *bugs* verificáveis passam por *AST Validation* (descarta alucinações, linha tracejada) e depois por ESBMC-Python, que confirma, refuta ou retorna inconclusivo; os três desfechos chegam ao *Final Report*.

Preprocessing. O arquivo `.py` é convertido em Árvore Sintática Abstrata (AST) via módulo `ast` do Python. Cada função é representada internamente por uma estrutura que organiza parâmetros, anotações de tipo, operações, laços e condicionais, usada como entrada para a etapa *Analyzer*.

Analyzer. Cada função pré-processada é enviada ao modelo de linguagem via *prompt* estruturado com *persona prompting* (especialista em segurança de código Python) e *Chain-of-Thought* (CoT), mitigando alucinações e organizando o raciocínio [9, 10]; o *prompt* completo está disponível no repositório de artefatos (Disponibilidade de Artefatos). O modelo produz uma saída em *schema* JSON com categoria, expressão, a flag `verifiable` e justificativa (exemplos reais no repositório de artefatos, Disponibilidade de Artefatos); hipóteses com `verifiable` falso não são submetidas ao ESBMC-Python, sendo registradas no relatório final como não verificadas, sob um *timeout* independente de 300 s (`--llm-timeout`) para acomodar a maior latência dos modelos locais via Ollama. *Bugs* verificáveis seguem para *AST Validation*; *code smells* vão direto ao *Final Report*.

AST Validation. A correspondência é textual: a expressão da LLM é normalizada via `ast.unparse` e comparada a um nó da AST de mesmo tipo (operação binária para divisão por zero, acesso indexado para *out-of-bounds*), exigindo os mesmos identificadores e literais. Quando a expressão se repete na função, a linha reportada pela LLM define qual ocorrência é usada. Sem correspondência exata, a busca é ampliada a qualquer nó executável (chamadas, operações binárias, acessos indexados, comparações, atributos). Para asserções, aceita-se correspondência textual com o código ou estrutural com a condição do `assert`. Sem correspondência em nenhuma fase, a hipótese é uma alucinação estrutural e é descartada [5].

ESBMC-Python. Apenas hipóteses que passaram pela *AST Validation* são submetidas ao ESBMC-Python, executado com `--function`, `--incremental-bmc` e `--assign-param-nondet`. Essas opções restringem a análise à função, realizam a verificação incremental e tratam seus parâmetros como entradas não determinísticas. O solucionador SMT utilizado é o Z3 (v4.8.12); o limite de *unwinding* não é fixado manualmente, sendo determinado incrementalmente pelo próprio `--incremental-bmc`. O *timeout* do *pipeline* é de 30 s por execução; se excedido, o resultado é inconclusivo, embora nenhuma execução tenha atingido esse limite no experimento reportado. A hipótese é confirmada apenas quando a violação corresponde à categoria prevista.

Final Report. O relatório final consolida os *code smells* produzidos pelo modelo de linguagem, os *bugs* verificáveis confirmados pelo ESBMC-Python e o resultado da validação estrutural. Nesta avaliação, esse relatório é comparado ao rótulo de *ground truth* do *dataset* para o cálculo das métricas, como mostrado na próxima seção.

4 Avaliação Preliminar

Os experimentos foram executados com ESBMC-Python 8.3.0, Python 3.9+ e Ollama 0.30.5 (modelos locais), em GPU NVIDIA RTX 3060 (6GB VRAM), WSL2.

4.1 Ameaças à Validade

Distinguem-se três ameaças à validade. Interna: execução única, sem testes estatísticos; não mitigada. Construto: *dataset* sintético (N=55) e taxonomia de seis categorias dada ao modelo facilitam a tarefa; defeitos selecionados via Flow A enviesam a amostra para *bugs* já prováveis pelo ESBMC-Python, mitigado parcialmente por rótulos atribuídos manualmente e independentes do verificador. Externa: sem comparação com *baselines* tradicionais (Pylint, Bandit, Radon) nem com EVA ou PyVeritas, por restrição de tempo; fica para trabalho futuro.

4.2 Configuração Experimental

O *dataset* utilizado (disponível no repositório de artefatos, Disponibilidade de Artefatos) é composto por 70 funções Python: 45 contêm *bugs* executáveis (15 por categoria, totalizando 48 instâncias de defeito, já que *out-of-bounds* pode ocorrer mais de uma vez por função), 15 contêm *code smells* (5 por categoria) e 10 são funções limpas, usadas como controle negativo. Todas as 70 funções possuem anotações de tipo completas (parâmetros e retorno), requisito do ESBMC-Python. Todas (bugs, *code smells* e funções limpas) foram geradas com auxílio de um modelo de linguagem (Claude Sonnet 4.6, mesmo modelo avaliado como claude-sonnet-4-6 na Seção 4.4) e revisadas manualmente pela autora. Os *bugs* foram construídos iterativamente com o ESBMC-Python, executado sobre cada função (Flow A) até confirmar a violação esperada; o rótulo de categoria foi então atribuído manualmente; por construção, o *dataset* contém apenas defeitos alcançáveis pelo ESBMC-Python (ameaça de construto, ver Ameaças à Validade). As 48 instâncias de defeito eram todas alcançáveis e foram identificadas pelo verificador (P=R=F1=1,000), resultado consistente com a construção do *dataset* descrita acima; o Flow B também identifica *code smells*,

fora do escopo do ESBMC-Python. Os *code smells* foram construídos para violar limiares de complexidade de Fowler et al. [3], verificados manualmente: *many_parameters* segue limiar fixo (≥5 parâmetros); *long_method* e *complex_conditional* foram julgados qualitativamente, sem limiar numérico.

Foram comparados três fluxos de execução: **Flow A**, execução direta do ESBMC-Python, utilizada apenas para validar o *dataset*; **Flow B**, *pipeline* híbrido proposto, composto pela triagem do modelo de linguagem, validação estrutural por AST e confirmação formal pelo ESBMC-Python; e **Flow C**, uso isolado do modelo de linguagem, sem confirmação formal. O Flow A não é tratado como abordagem concorrente, sendo empregado apenas como teste de sanidade para verificar a alcançabilidade dos defeitos anotados. Embora o mesmo verificador seja utilizado no Flow A e no Flow B, essa circularidade é discutida como ameaça de construto em Ameaças à Validade. Os Flows B e C reutilizam exatamente a mesma saída do modelo de linguagem para cada função.

O estudo utiliza cinco modelos, acessados em julho de 2026: três via API (gpt-5.5, claude-sonnet-4-6 e gemini-3.1-flash-lite) e dois modelos locais executados via Ollama (deepseek-r1:7b e qwen2.5-coder:7b). A seleção buscou representar diferentes famílias de modelos: proprietários (OpenAI, Anthropic, Google) e abertos executáveis localmente [1]. O qwen2.5-coder:7b foi incluído por especialização em código, e o deepseek-r1:7b por ser família recente avaliada em qualidade de código [6, 9]. SLM refere-se aqui aos dois modelos locais (~7B parâmetros). A temperatura foi configurada como 0 para deepseek-r1:7b, qwen2.5-coder:7b e gemini-3.1-flash-lite, para aumentar a reprodutibilidade [6]. Para gpt-5.5 e claude-sonnet-4-6, esse parâmetro não foi aceito pela API, permanecendo no valor padrão (não divulgado) de cada provedor. Apenas gpt-5.5 e claude-sonnet-4-6 utilizaram APIs pagas.

4.3 Métricas

Precisão (P), Revocação (R) e F1 são calculados por achado (*finding-level*): P avalia a confiabilidade dos achados, R a cobertura dos defeitos e F1 sintetiza o equilíbrio entre ambas. Cada instância esperada no *ground truth* é comparada aos achados por função e categoria, sem reutilizar instâncias já casadas (múltiplas ocorrências da mesma categoria, como dois *out-of-bounds*, contam como TPs separados). Achado sem correspondência conta como Falso Positivo (FP); instância não identificada conta como Falso Negativo (FN). Funções limpas só podem contribuir com FP.

Acurácia (Acc) e o Coeficiente de Correlação de Matthews (MCC) são calculados por função (bug vs. limpa): TP se o modelo confirmou algum achado, TN se a função limpa não teve achado confirmado. Acc isolada pode ser enganosa em *datasets* desbalanceados [1], por isso é sempre acompanhada do MCC (45 funções com bug, 10 limpas).

As 15 funções com apenas *code smells* são excluídas desse cálculo. Acc, MCC, FCR e NRR aplicam-se apenas a *bugs* formais; para *code smells* reporta-se F1 por categoria (Tabela 2), com P/R agregados por modelo: gpt-5.5 0,600/0,800; claude-sonnet-4-6 0,789/1,000; gemini-3.1-flash-lite 0,800/0,800; deepseek-r1:7b 0,462/0,400; qwen2.5-coder:7b 0,081/0,933.

FCR (*Formal Confirmation Rate*) é a razão entre hipóteses confirmadas pelo ESBMC-Python (verdadeiros positivos) e o total de hipóteses submetidas ao verificador após a AST Validation, incluindo casos não confirmados e inconclusivos.

NRR (*Noise Reduction Rate*) mede a redução percentual de falsos positivos do Flow C para o Flow B: NRR = 1,000 indica eliminação total do ruído da LLM; quando não há falsos positivos no Flow C, a razão é indefinida e reportada como N/A.

Tabela 1: Flow C (somente LLM) vs. Flow B (*pipeline* híbrido). R não caiu em nenhum modelo, já que o *pipeline* só confirma hipóteses já geradas pela LLM. N=45 funções com *bug* (10 limpas). FCR = confirmadas/tentativas formais; NRR = redução percentual de falsos positivos do Flow C ao B. Execução única por modelo, sem repetição.

Modelo	R	P_C	$F1_C$	P_B	$F1_B$	Acc_B	MCC_B	FCR/NRR
gpt-5.5	1,000	0,889	0,941	1,000	1,000	1,000	1,000	0,889 / 1,000
claude-sonnet-4-6	1,000	0,960	0,980	0,980	0,990	1,000	1,000	0,960 / 0,500
gemini-3.1-flash-lite	0,896	0,977	0,935	1,000	0,945	0,909	0,770	0,977 / 1,000
qwen2.5-coder:7b	0,875	0,627	0,730	0,977	0,923	0,927	0,807	0,857 / 0,960
deepseek-r1:7b	0,625	0,682	0,652	1,000	0,769	0,709	0,498	0,909 / 1,000

4.4 Resultados e Discussão

A Tabela 1 compara Flow C e Flow B para detecção de bugs. Modelos de API já apresentam F1 alto no Flow C, enquanto os locais ficam mais baixos (F1 entre 0,652 e 0,730) devido a mais falsos positivos, consistente com a literatura [1].

O *pipeline* beneficiou todos os modelos proporcionalmente ao ruído do Flow C (NRR entre 0,500 e 1,000, Tabela 1), reduzindo o total de falsos positivos de 48 para 2 somando os cinco modelos, e eliminando quase todo o ruído dos modelos locais e da maioria dos modelos de API. Por modelo, os falsos positivos caíram de 6 para 0 (gpt-5.5), de 2 para 1 (claude-sonnet-4-6), de 1 para 0 (gemini-3.1-flash-lite), de 25 para 1 (qwen2.5-coder:7b) e de 14 para 0 (deepseek-r1:7b).

Isso é consistente, para Python, com o paradigma híbrido discutido por Tihanyi et al. [10].

O ganho por função acompanha o ruído inicial: o MCC do qwen2.5-coder:7b evolui de −0,132 para 0,807, o maior salto observado, seguido pelo deepseek-r1:7b (0,346 para 0,498); mesmo gpt-5.5 e claude-sonnet-4-6, já precisos no Flow C, chegam a MCC = 1,000 no Flow B.

A distância de F1 entre o melhor modelo local (qwen2.5-coder:7b) e o pior de API (gemini-3.1-flash-lite) cai de 0,205 para 0,022 do Flow C ao B. O deepseek-r1:7b permanece o mais fraco do grupo.

A *AST Validation* complementa a verificação formal ao filtrar alucinações estruturais (razão entre achados rejeitados e total de achados verificáveis propostos). Os três modelos de API não apresentaram alucinações, enquanto deepseek-r1:7b atingiu 25,00% (11 de 44 achados verificáveis propostos) e qwen2.5-coder:7b atingiu 10,91% (6 de 55), casos em que o modelo reporta expressões inexistentes no código, como `assert x > 0` para uma função que continha `assert value < limit`. Cada função é analisada em uma chamada independente, sem histórico entre requisições, descartando contaminação de contexto entre arquivos como causa. A AST elimina essas inconsistências sintáticas, enquanto o ESBMC-Python descarta hipóteses semanticamente incorretas sobre expressões válidas.

Tabela 2: Detecção de *code smells* por categoria (F1); Flow B iguala Flow C, sem confirmação formal. GPT=gpt-5.5, Claude=claude-sonnet-4-6, Gemini=gemini-3.1-flash-lite, Qwen=qwen2.5-coder:7b, DeepSeek=deepseek-r1:7b.

Categoria	GPT	Claude	Gemini	Qwen	DeepSeek
complex_conditional	0,667	0,909	0,750	0,140	0,000
long_method	0,571	1,000	0,889	0,170	0,500
many_parameters	0,769	0,769	0,769	0,139	0,667

Entre as categorias de bugs, os modelos locais seguem menos consistentes mesmo após a validação formal: o deepseek-r1:7b cai para F1 = 0,696 em *division by zero* contra F1 de até 0,889 nas outras categorias (queda ≥0,193); o qwen2.5-coder:7b cai para F1 = 0,800 em *out-of-bounds* (queda ≥0,168).

Os modelos de API não apresentam essa oscilação, com $F1_B$ agregados entre 0,945 e 1,000 (Tabela 1). Essas categorias exigem raciocínio sobre restrições de índices e valores de denominador, reduzindo a eficácia de modelos que dependem de padrões sintáticos [1]. O *pipeline* não recupera esses casos: o ESBMC-Python confirma ou descarta hipóteses já geradas pela LLM, mas não gera novas, de modo que erros de raciocínio semântico do modelo permanecem como FN mesmo após a validação formal.

Para *code smells*, observa-se comportamento distinto dos bugs executáveis. Como cada categoria contém apenas cinco exemplos, diferenças de um único acerto produzem variações expressivas nas métricas, limitando a generalização. *Long method* apresentou a maior variação entre modelos de API (F1 entre 0,571 e 1,000), enquanto os locais permaneceram consistentemente mais fracos (F1 ≤ 0,667), de acordo com Souza et al. [9], que relatam variabilidade elevada entre categorias de *code smells*.

O caso mais extremo de ruído foi observado em qwen2.5-coder:7b para detecção de *code smells*. Embora o modelo identifique 14 dos 15 casos presentes no *dataset* (recall = 0,933), ele produz 158 falsos positivos, resultando em precisão de apenas 0,081 (14/172).

Como o ESBMC-Python não verifica propriedades de manutenibilidade, esses achados seguem diretamente para o relatório final após a validação estrutural. Esse resultado evidencia que, embora a abordagem híbrida seja altamente eficaz para *bugs* executáveis, ainda é necessária uma estratégia complementar para reduzir o ruído na detecção de *code smells*. Essa observação também reforça os resultados de Santos et al. [7], que mostram que, apesar das similaridades estruturais entre *bugs* e *code smells*, ambas as tarefas apresentam desafios distintos. Essa diferença combina dois fatores: modelos locais são menos calibrados entre tarefas [4], e *code smells* não passam por filtragem posterior (*AST Validation* ou ESBMC-Python), chegando ao relatório final sem correção. Em suma, o *pipeline* é orientado ao aumento de precisão, não de revocação.

5 Considerações Finais

Neste trabalho foi apresentado um *pipeline* híbrido que combina LLMs/SLMs, validação estrutural por AST e verificação formal com ESBMC-Python para detectar *bugs* executáveis e *code smells* em funções Python. As três etapas se mostraram complementares [10]: a AST filtra alucinações estruturais [5], e o ESBMC-Python reduz mais o ruído da LLM justamente nos modelos locais (SLMs), onde a triagem isolada era menos confiável. Em resposta a RQ1, o *pipeline* elevou a precisão para entre 0,977 e 1,000 nos cinco modelos, sem perda de revocação; já RQ2 permanece limitada, pois *code smells* não passam por validação estrutural nem confirmação formal, mantendo a instabilidade já observada na triagem isolada da LLM.

Como trabalhos futuros, pretende-se ampliar o *dataset* com projetos reais e maiores, avaliar *bugs* e *code smells* em estudos separados, comparar modelos pagos e gratuitos, e investigar estratégias para melhorar a revocação (*harnesses* de teste ou votação entre modelos) [8, 10].

Disponibilidade de Artefatos

Código, *prompt*, *dataset* e métricas brutas por modelo e *flow* estão disponíveis para reprodução em: https://github.com/feralice/llm-esbmc-pipeline/tree/ise2026-artifacts/ise2026_artifacts.

Foi utilizada IA generativa como parte do *pipeline* proposto, portanto, como parte do ferramental da pesquisa em desenvolvimento. Mas, também, foi utilizada para refinamento dos textos gerados.

Referências

[1] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability Detection with Code Language Models: How Far Are We?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1729–1741. doi:10.1109/ICSE55347.2025.00038

[2] Bruno Farias, Rafael Menezes, Eddie B. de Lima Filho, Youcheng Sun, and Lucas C. Cordeiro. 2024. ESBMC-Python: A Bounded Model Checker for Python Programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1836–1840. doi:10.1145/3650212.3685304

[3] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Reading, MA.

[4] Rohit Gheyi, Márcio Ribeiro, and Jonhnanthan Oliveira. 2025. Evaluating the Effectiveness of Small Language Models in Detecting Refactoring Bugs. *arXiv preprint arXiv:2502.18454* (2025).

[5] Dipin Khati, Daniel Rodriguez-Cardenas, Paul Pantzer, and Denys Poshyvanyk. 2026. Detecting and Correcting Hallucinations in LLM-Generated Code via Deterministic AST Analysis. In *2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering (FORGE '26)*. 72–76. doi:10.1145/3793655.3793725 April 12–13, 2026, Rio de Janeiro, Brazil. arXiv:2601.19106.

[6] Pedro Orvalho and Marta Kwiatkowska. 2025. PyVeritas: On Verifying Python via LLM-Based Transpilation and Bounded Model Checking for C. arXiv:2508.08171 Preprint.

[7] Geanderson Santos, Amanda Santana, Gustavo Vale, and Eduardo Figueiredo. 2023. Yet Another Model! A Study on Model's Similarities for Defect and Code Smells. In *Fundamental Approaches to Software Engineering (FASE 2023) (Lecture Notes in Computer Science, Vol. 13991)*. Springer, 282–305. doi:10.1007/978-3-031-30826-0_16

[8] Shivkumar Shivaji, Natalia Lobakhina, Klaus Havelund, Alessandro Pinto, and Lucas Cordeiro. 2026. LLM-Assisted Translation and Bounded Model Checking of Python Code. In *3rd Artificial Intelligence ISoLA*. Accepted.

[9] Saymon Souza, Amanda Santana, Eduardo Figueiredo, Igor Muzetti, João Eduardo Montandon, and Lionel Briand. 2026. Beyond Strict Rules: Assessing the Effectiveness of Large Language Models for Code Smell Detection. *arXiv preprint arXiv:2601.09873* (2026).

[10] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Bilel Cherif, Richard Dubniczky, Ridhi Jain, and Lucas Cordeiro. 2026. *Vulnerability Detection: From Formal Verification to Large Language Models and Hybrid Approaches: A Comprehensive Overview.* Springer, 33–47. doi:10.1007/978-3-031-99447-0_3

[11] TIOBE Software. 2026. TIOBE Index. https://www.tiobe.com/tiobe-index/. Accessed: 2026-08-10.