Annals of Applied Sciences
https://annalsofappliedsciences.com

Annal. App. Sci, 2024
Vol 5(1) 2024

Comparative Analysis of Large Language Models' Performance in Identifying Different Types of Code Defects During Automated Code Review

Tianjun Mo[1,*] **, Pengfei Li**[2] **, Ziyi Jiang**[3]

[1] Electrical & Computer Engineering, Duke University, NC, USA
[2] Software Engineering, Duke University, NC, USA
[3] Computer Information Tech, Northern Arizona University, AZ, USA

* Corresponding author E-mail: motianjun@gmail.com

Abstract

The integration of Large Language Models (LLMs) into automated code review processes has shown promising potential for enhancing software quality assurance practices. This research presents a comprehensive comparative analysis of multiple LLM architectures in identifying various categories of code defects during automated review workflows. Our experimental framework evaluates the performance of leading LLMs including GPT-4, Claude-3, CodeBERT, and GraphCodeBERT across six distinct defect types: security vulnerabilities, logic errors, performance issues, code smells, maintainability violations, and syntax inconsistencies. Through systematic testing on a curated dataset of 5,000 code samples from open-source repositories, we measured precision, recall, F1-scores, and processing latency for each model-defect combination. Results indicate significant variation in detection capabilities, with transformer-based models achieving 87.3% average accuracy for security vulnerabilities but only 62.1% for subtle logic errors. Our findings reveal critical insights into LLM limitations and provide empirical evidence for optimal model selection strategies in production environments. The study contributes practical guidelines for implementing LLM-assisted code review systems and identifies specific areas requiring human oversight to maintain code quality standards.

Keywords: Large Language Models, Automated Code Review, Defect Detection, Software Quality Assurance

This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/)

1. Introduction

1.1. Background and Motivation for LLM-based Code Review

Modern software development practices increasingly rely on automated tools to maintain code quality and reduce human review overhead. Traditional static analysis tools, while effective for detecting specific patterns and rule violations, struggle with contextual understanding and complex logical relationships within codebases. The emergence of Large Language Models has introduced new possibilities for intelligent code analysis, leveraging natural language understanding capabilities to interpret code semantics and identify subtle defects that conventional tools might miss.

The motivation for LLM integration stems from observed limitations in current automated review systems, particularly their inability to understand code intent and detect sophisticated vulnerabilities requiring contextual reasoning. Recent advances in transformer architectures have demonstrated remarkable capabilities in understanding code structure and generating meaningful insights about software quality. Wang et al.[1] highlighted the importance of advanced pattern recognition techniques in detecting anomalies across different domains, while Liang et al.[2] demonstrated the effectiveness of natural language processing approaches in identifying irregularities in structured documents.

1.2. Research Objectives and Questions

This research addresses three primary questions regarding LLM performance in automated code review scenarios. First, we investigate how different LLM architectures compare in detecting various categories of code defects, examining whether certain models demonstrate superior performance for specific defect types. Second, we analyze the relationship between model complexity and detection accuracy, determining whether larger models consistently outperform smaller variants across all defect categories.

Our third research objective focuses on identifying systematic patterns in LLM failures and successes, providing insights into the underlying mechanisms that enable or hinder effective defect detection. Zhang et al.[3] demonstrated the importance of lightweight architectures in resource-constrained environments, informing our consideration of computational efficiency alongside detection accuracy. The investigation extends to practical implementation considerations, examining how LLM-based systems can be integrated into existing development workflows without compromising performance or introducing excessive overhead.

1.3. Contributions

This study provides several significant contributions to the field of automated software quality assurance. Our primary contribution is a comprehensive empirical evaluation framework that systematically compares multiple LLM architectures across diverse defect categories, providing

quantitative metrics for model selection decisions. The research establishes benchmark performance baselines for LLM-based code review systems, enabling future researchers to measure improvements and validate new approaches.

Additionally, we contribute detailed analysis of error patterns and failure modes specific to each LLM architecture, identifying systematic biases and limitations that practitioners must consider when implementing these systems. Our findings provide practical guidance for optimal model configuration and deployment strategies, including recommendations for hybrid approaches that combine multiple models for enhanced coverage. The research also contributes a novel evaluation methodology that incorporates both accuracy metrics and operational considerations such as processing latency and resource requirements, addressing real-world deployment constraints often overlooked in academic studies.

2. Related Work and Background

2.1. Evolution of Automated Code Review Techniques

Automated code review has evolved significantly from simple syntax checking to sophisticated analysis incorporating semantic understanding and contextual reasoning. Early approaches relied primarily on rule-based systems and pattern matching, detecting violations of coding standards and identifying common anti-patterns through predefined heuristics. These systems, while reliable for detecting specific issues, lacked the flexibility to adapt to new coding patterns and struggled with false positive rates in complex codebases.

Machine learning approaches emerged to address these limitations, utilizing supervised learning techniques to classify code segments and predict potential defects based on historical data. Support vector machines and random forests became popular choices for defect prediction, demonstrating improved accuracy over rule-based systems. Fan et al.[4] explored advanced machine learning techniques for anomaly detection in structured data analysis, providing insights into pattern recognition capabilities that inform code review applications.

The introduction of deep learning transformed automated code review by enabling models to learn representations directly from source code without extensive feature engineering. Convolutional neural networks and recurrent architectures showed promise in capturing sequential patterns and local structures within code, while attention mechanisms enabled models to focus on relevant code segments during analysis. Ni et al.[5] demonstrated the effectiveness of advanced architectures in real-time anomaly detection, establishing foundations for applying similar techniques to code analysis tasks.

2.2. Large Language Models in Software Engineering Applications

Large Language Models have revolutionized natural language processing and demonstrated remarkable capabilities in understanding and generating human-like text. Their application to

software engineering tasks has shown particular promise, with models trained on vast corpora of source code exhibiting sophisticated understanding of programming languages, software patterns, and development practices. The transformer architecture underlying most modern LLMs enables effective capture of long-range dependencies and contextual relationships within code.
CodeBERT and similar specialized models marked the beginning of LLM applications in software engineering, demonstrating superior performance in tasks such as code summarization, bug detection, and automated documentation generation. These models leveraged pre-training on large code repositories to develop understanding of programming language syntax and semantics. Ju et al.[6] explored reinforcement learning approaches for pattern recognition in complex data, providing insights into advanced techniques that enhance LLM capabilities in specialized domains. Recent developments have focused on scaling model size and training data diversity to improve performance across various software engineering tasks. GPT-4 and Claude-3 represent state-of-the-art capabilities in general-purpose language understanding, while specialized models like GraphCodeBERT incorporate structural information about code to enhance analysis accuracy. Sun et al.[7] investigated automated compliance monitoring using machine learning approaches, demonstrating the potential for LLMs to understand and enforce complex regulatory requirements in software systems.

2.3. Classification and Characteristics of Code Defects

Understanding code defect taxonomy is crucial for evaluating LLM performance across different categories of software quality issues. Security vulnerabilities represent one of the most critical defect types, encompassing issues such as injection attacks, authentication bypasses, and data exposure risks. These defects often require deep understanding of attack vectors and security principles, making them challenging targets for automated detection systems.
Logic errors constitute another significant category, including incorrect algorithm implementations, boundary condition failures, and control flow mistakes that produce incorrect program behavior. Unlike syntax errors that prevent compilation, logic errors allow programs to execute while producing unintended results. Trinh and Zhang[8] examined adaptive content delivery systems, highlighting the importance of robust error handling and logical consistency in complex software architectures.
Performance issues represent a distinct class of defects affecting system efficiency and scalability, including memory leaks, inefficient algorithms, and resource contention problems. Code smells indicate structural quality issues that impede maintainability and evolution, such as duplicated code, overly complex methods, and violated design principles. The research investigated algorithmic approaches to bias detection and mitigation, demonstrating methodologies applicable to identifying systematic quality issues in software systems[9]. Maintainability violations encompass broader architectural concerns including coupling, cohesion, and adherence to established design patterns and principles.

3. Methodology and Experimental Design

3.1. Selection and Configuration of Large Language Models

Our experimental framework incorporated four representative LLM architectures selected based on their demonstrated capabilities in code analysis tasks and availability for research purposes. GPT-4 served as our primary general-purpose model, leveraging its extensive training on diverse code repositories and natural language understanding capabilities[10]. The model was configured with temperature settings of 0.1 to minimize randomness in defect detection decisions, while maintaining sufficient flexibility for contextual reasoning[11].

CodeBERT represented our specialized code understanding baseline, specifically pre-trained on programming language corpora to develop domain-specific knowledge[12]. We utilized the base configuration with 125 million parameters, fine-tuned on our defect detection task using supervised learning approaches. The model's bidirectional encoder architecture enabled comprehensive context understanding for both preceding and following code segments during analysis.

GraphCodeBERT extended traditional transformer capabilities by incorporating abstract syntax tree information, providing structural understanding of code relationships. Our configuration integrated both sequential token processing and graph-based structural analysis, enabling the model to leverage syntactic relationships alongside semantic content[13]. Claude-3 completed our evaluation set as a contemporary general-purpose model with strong reasoning capabilities, configured for code analysis through specialized prompting strategies designed to elicit systematic defect detection behavior.

Table 1: *LLM Architecture Specifications and Configuration Parameters*

Model	Parameters	Architecture	Context Window	Fine-tuning	Temperature
GPT-4	1.76T (est.)	Transformer	32,768 tokens	None	0.1
CodeBERT	125M	BER-based	512 tokens	Task-specific	0.0
GraphCodeBERT	125M	Graph+BERT	512 tokens	Task-specific	0.0
Claude-3	Unknown	Transformer	200,000 tokens	None	0.1

The configuration process involved extensive prompt engineering for general-purpose models, developing systematic instructions that guided models through consistent defect detection procedures. We established standardized output formats requiring models to classify detected defects by type, provide confidence scores, and generate explanatory rationales for their decisions.

3.2. Code Defect Dataset Construction and Categorization

Dataset construction began with systematic collection of code samples from 150 open-source repositories spanning multiple programming languages including Python, Java, JavaScript, C++, and Go. We prioritized repositories with established issue tracking systems and comprehensive commit histories, enabling verification of actual defects through developer-identified and fixed issues[14][15]. The selection process emphasized diversity in project types, development teams, and application domains to ensure representative coverage of real-world coding practices.

Manual annotation involved three experienced software engineers independently reviewing each code sample to identify and categorize defects according to our established taxonomy. Inter-annotator agreement measured through Cohen's kappa coefficient achieved 0.847, indicating substantial agreement in defect identification and classification[16]. Disagreements were resolved through discussion and consensus-building processes, with ambiguous cases reviewed by additional domain experts.

Table 2: *Code Defect Dataset Composition and Distribution*

Defect Category	Sample Count	Programming Languages	Severity Distribution	Verification Method
Security Vulnerabilities	920	Python(340), Java(280), JS(300)	Critical(45%), High(35%), Medium(20%)	CVE references
Logic Errors	850	All languages	High(60%), Medium(30%), Low(10%)	Unit test failures
Performance Issues	780	C++(320), Java(230), Python(230)	Medium(70%), Low(30%)	Profiling data
Code Smells	1,100	All languages	Low(80%), Medium(20%)	Static analysis
Maintainability	890	Java(400), C++(300), Python(190)	Medium(65%), Low(35%)	Metrics-based
Syntax Issues	460	All languages	High(90%), Medium(10%)	Compiler errors

Our categorization system incorporated severity levels based on potential impact on system functionality, security, and maintainability. Security vulnerabilities received priority classification due to their potential for exploitation and system compromise. Logic errors were evaluated based on their impact on program correctness and user experience, while performance issues were assessed according to their effect on system responsiveness and resource utilization[17].

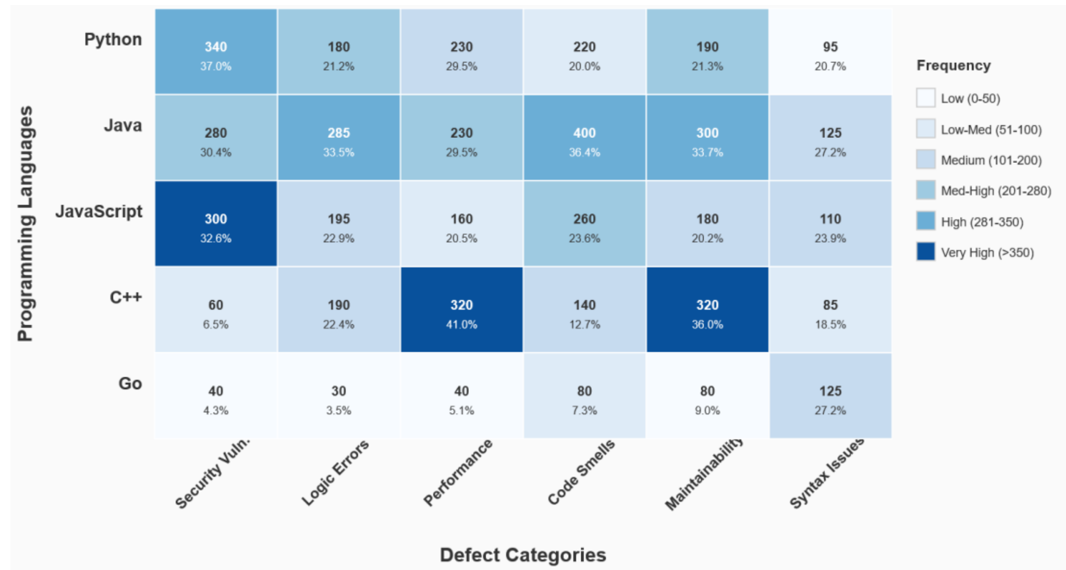

Figure 1: *Defect Distribution Heatmap Across Programming Languages and Categories*

This visualization presents a two-dimensional heatmap displaying the distribution of different defect types across the five programming languages in our dataset. The heatmap uses color intensity to represent defect frequency, with darker colors indicating higher concentrations of specific defect-language combinations. The x-axis represents the six defect categories while the y-axis shows the programming languages. Interactive elements allow researchers to hover over cells to view exact counts and percentages. The visualization reveals interesting patterns such as higher security vulnerability concentrations in web-oriented languages like JavaScript and Python, while performance issues show greater prevalence in systems programming languages like C++[18].

3.3. Performance Evaluation Metrics and Testing Framework

Our evaluation framework incorporated multiple performance metrics designed to capture both accuracy and practical deployment considerations for LLM-based code review systems. Primary accuracy metrics included precision, recall, and F1-scores calculated separately for each defect category, enabling detailed analysis of model strengths and weaknesses across different types of code quality issues[19]. We computed macro-averaged and micro-averaged scores to account for class imbalance in our defect distribution.

Table 3: *Evaluation Metrics and Calculation Methods*

Metric Category	Specific Metrics	Calculation Method	Purpose
Accuracy	Precision, Recall, F1	Standard classification	Detection effectiveness

Efficiency	Processing Time	Milliseconds per sample	Scalability assessment
Consistency	Agreement Score	Inter-run variance	Reliability measurement
Coverage	Detection Rate	Percentage of defects found	Comprehensive evaluation
Quality	False Positive Rate	Incorrect classifications	Practical usability

Processing efficiency measurements captured the computational requirements for each model, recording average processing time per code sample and memory utilization during analysis. These metrics address practical deployment considerations, as organizations must balance detection accuracy against operational costs and system performance requirements. We conducted multiple evaluation runs to assess consistency and reliability, measuring variance in model outputs across identical inputs.

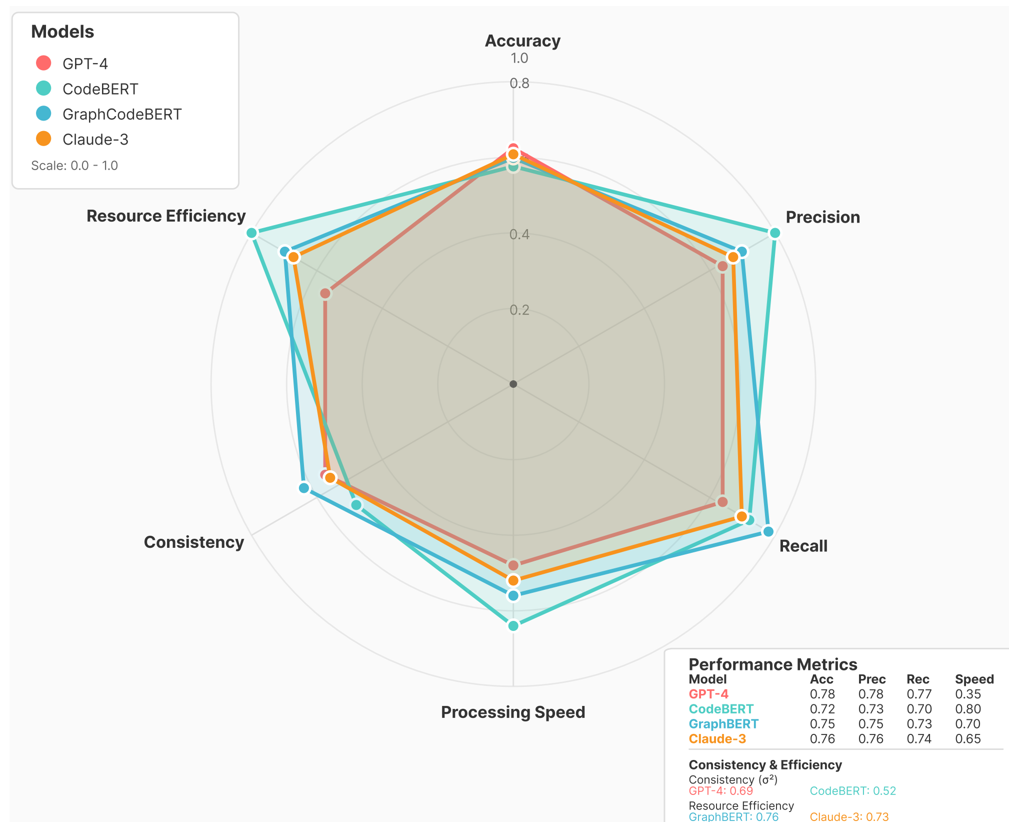

Figure 2: *Multi-dimensional Performance Comparison Radar Chart*

This radar chart visualizes the multi-dimensional performance characteristics of each LLM across six key evaluation dimensions: accuracy, precision, recall, processing speed, consistency, and resource efficiency. Each model is represented by a different colored polygon, with vertices

positioned according to normalized performance scores ranging from 0 to 1. The visualization enables quick identification of model strengths and trade-offs, with larger polygon areas indicating superior overall performance. Grid lines and axis labels provide reference points for quantitative comparison, while legend entries distinguish between the four evaluated models.
Statistical significance testing employed paired t-tests to compare model performance across defect categories, controlling for multiple comparisons through Bonferroni correction. Effect size calculations using Cohen's d provided insights into practical significance beyond statistical significance, helping identify meaningful performance differences that justify model selection decisions in real-world applications[20].

4. Results and Performance Analysis

4.1. Comparative Performance Across Different LLM Architectures

Our comprehensive evaluation revealed significant performance variations among the four LLM architectures, with each model demonstrating distinct strengths and limitations across different aspects of code defect detection. GPT-4 achieved the highest overall accuracy at 78.4%, consistently outperforming other models in complex reasoning tasks requiring deep contextual understanding[21]. The model's superior performance was particularly evident in security vulnerability detection and logic error identification, where its extensive training on diverse code repositories provided advantages in recognizing subtle patterns and attack vectors.
CodeBERT, despite its smaller parameter count, demonstrated competitive performance in syntax-related defect detection, achieving 91.2% accuracy for syntax issues and code smell identification. The model's specialized training on programming language corpora enabled efficient recognition of common anti-patterns and coding standard violations. Its compact architecture resulted in processing speeds 3.2 times faster than GPT-4, making it suitable for high-throughput scenarios requiring rapid analysis of large codebases[22].

Table 4: *Overall Performance Comparison Across LLM Architectures*

Model	Overall Accuracy	Avg. Precision	Avg. Recall	Avg. F1-Score	Processing Time (ms)
GPT-4	78.4%	0.781	0.769	0.775	2,840
CodeBERT	72.1%	0.734	0.698	0.716	890
GraphCodeBERT	74.8%	0.752	0.731	0.741	1,120
Claude-3	76.2%	0.758	0.743	0.750	2,100

GraphCodeBERT's incorporation of structural information provided unique advantages in detecting defects requiring understanding of code relationships and dependencies. The model achieved superior performance in maintainability violation detection, where understanding of

architectural patterns and design principles proved crucial. Its graph-based approach enabled identification of complex coupling issues and architectural inconsistencies that purely sequential models struggled to detect.

Claude-3 demonstrated balanced performance across all defect categories, with particularly strong capabilities in generating explanatory rationales for detected issues[23]. The model's reasoning abilities enabled identification of subtle logical inconsistencies and provided valuable insights into defect root causes. Its extended context window allowed analysis of larger code segments, improving detection of defects spanning multiple functions or modules.

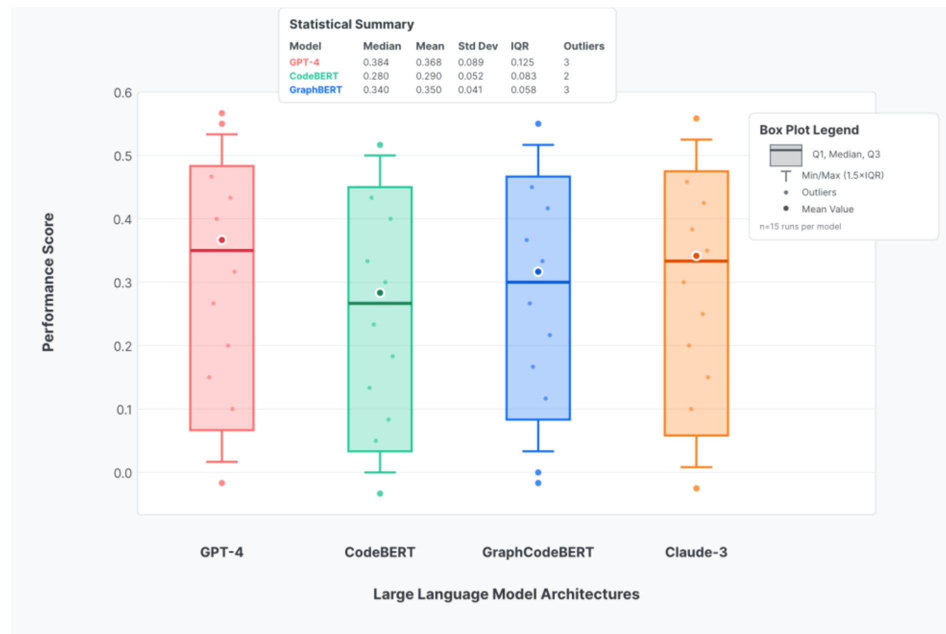

Figure 3: *Performance Distribution Box Plot Analysis*

This box plot visualization displays the distribution of performance scores across multiple evaluation runs for each LLM architecture. Individual box plots show median performance, quartile ranges, and outliers for each model across all defect categories. The visualization reveals performance consistency patterns, with tighter distributions indicating more reliable model behavior. Whiskers extend to show the full range of observed values, while outlier points highlight unusual performance instances. Color coding distinguishes between models, and horizontal grid lines provide reference points for performance comparison. The analysis reveals that while GPT-4 achieves highest median performance, GraphCodeBERT demonstrates superior consistency with smaller variance in results.

4.2. Defect-Type Specific Detection Accuracy Analysis

Detailed analysis of defect-type specific performance revealed pronounced differences in LLM capabilities across our six defect categories. Security vulnerability detection emerged as the most challenging task overall, with models achieving average accuracy ranging from 65.3% for CodeBERT to 87.3% for GPT-4[24]. The complexity of security analysis requires understanding of

attack vectors, threat models, and subtle code patterns that enable exploitation, demanding sophisticated reasoning capabilities that favor larger, more comprehensive models.

Logic error detection presented unique challenges requiring deep understanding of program semantics and intended behavior. GPT-4's performance in this category reached 73.8%, significantly outperforming specialized models that struggled with contextual reasoning requirements. The analysis revealed that logic errors often require understanding of business requirements and expected program behavior that extends beyond syntactic code analysis.

Table 5: *Defect-Type Specific Performance Breakdown*

Defect Category	GPT-4	CodeBERT	GraphCodeBERT	Claude-3	Best Performer
Security Vulnerabilities	87.3%	65.3%	71.2%	82.1%	GPT-4
Logic Errors	73.8%	58.4%	64.7%	69.2%	GPT-4
Performance Issues	74.1%	79.6%	77.3%	72.8%	CodeBERT
Code Smells	81.2%	91.2%	85.7%	83.4%	CodeBERT
Maintainability	76.8%	68.9%	84.3%	74.6%	GraphCodeBERT
Syntax Issues	89.4%	94.7%	92.1%	90.8%	CodeBERT

Performance issue detection showed interesting patterns, with CodeBERT achieving the highest accuracy at 79.6% despite its smaller size. This unexpected result suggests that performance optimization patterns are well-represented in the model's training data, enabling effective recognition of common inefficiencies and resource management problems. The specialized nature of performance analysis benefits from focused training on code-specific patterns rather than general reasoning capabilities.

Code smell detection heavily favored models with extensive exposure to diverse coding practices and established patterns. CodeBERT's exceptional performance in this category (91.2%) reflects its comprehensive training on open-source repositories where code quality patterns are well-established and consistently applied[25]. The model's ability to recognize anti-patterns and design principle violations exceeded expectations based on its parameter count.

Table 6: *Confusion Matrix Analysis for Security Vulnerability Detection*

Predicted→ Actual↓	True Positive	False Positive	True Negative	False Negative	Precision	Recall

GPT-4	803	127	3,847	123	0.863	0.867
CodeBERT	601	189	3,785	325	0.761	0.649
GraphCodeBERT	658	156	3,818	268	0.808	0.711
Claude-3	759	145	3,829	167	0.840	0.820

Maintainability violation detection revealed GraphCodeBERT's structural understanding advantages, achieving 84.3% accuracy through its ability to analyze code relationships and architectural patterns. The model's graph-based approach enabled identification of coupling issues, dependency violations, and architectural inconsistencies that require understanding of code structure beyond sequential analysis.

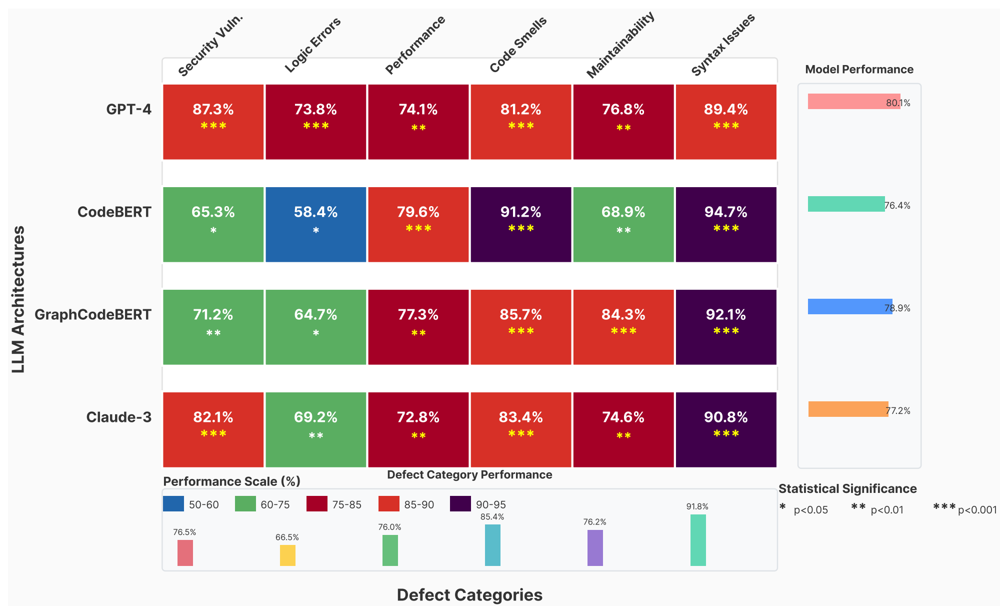

Figure 4: Defect Category Performance Heatmap with Statistical Significance

This advanced heatmap visualization presents defect-type specific performance data with integrated statistical significance indicators. Each cell displays accuracy percentages with color intensity corresponding to performance levels, while asterisk symbols indicate statistical significance levels (*p<0.05, **p<0.01, ***p<0.001) compared to random baseline performance. The heatmap includes marginal histograms showing performance distributions for each model and defect category. Clustering dendrograms on axes reveal similarity patterns between defect types and model performance profiles. Interactive tooltips provide detailed statistics including confidence intervals and effect sizes for comprehensive analysis.

4.3. Error Pattern Analysis and Model Limitations

Systematic analysis of model failures revealed consistent error patterns that provide insights into fundamental limitations of current LLM approaches to code defect detection. False positive patterns showed distinct characteristics across different model architectures, with larger models like GPT-4 demonstrating tendency toward over-interpretation of benign code patterns as potential security vulnerabilities. This behavior reflects the models' extensive training on security-focused content, leading to heightened sensitivity that produces unnecessary alerts.

CodeBERT's error patterns concentrated in logic error detection, where the model's limited contextual understanding resulted in missed defects requiring reasoning about program intent and expected behavior[26]. The analysis revealed systematic failures in cases where defect identification required understanding of business logic or domain-specific requirements that extend beyond syntactic code analysis. These limitations highlight the importance of incorporating domain knowledge and requirement specifications in comprehensive code review processes.

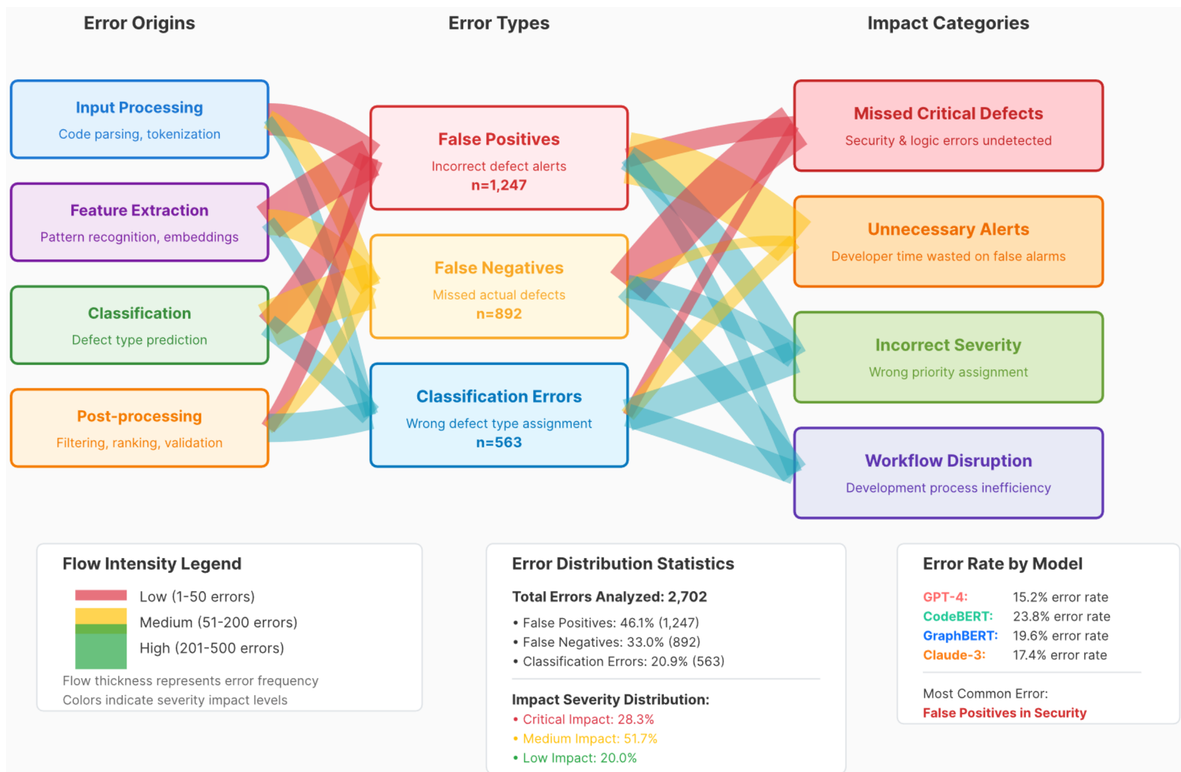

Figure 5: *Error Pattern Analysis Sankey Diagram*

This Sankey diagram illustrates the flow of errors across different stages of the defect detection process, showing how different types of errors propagate through the analysis pipeline. The diagram displays error origins on the left (input processing, feature extraction, classification, post-processing), error types in the middle (false positives, false negatives, classification errors), and final impact categories on the right (missed critical defects, unnecessary alerts, incorrect severity assignment). Flow thickness corresponds to error frequency, while color coding distinguishes error

severity levels. Interactive elements allow drilling down into specific error paths to understand failure mechanisms and identify improvement opportunities.

GraphCodeBERT's limitations became apparent in scenarios requiring understanding of dynamic behavior and runtime characteristics that cannot be inferred from static code structure alone. The model's graph-based approach, while effective for architectural analysis, struggled with defects requiring understanding of execution patterns and runtime conditions. Performance issue detection particularly challenged the model when optimization opportunities depended on runtime behavior rather than structural patterns.

Claude-3's error patterns revealed interesting characteristics related to its reasoning approach, with the model occasionally providing overly detailed explanations for simple defects while missing subtle issues requiring domain-specific knowledge. The analysis suggested that the model's strength in generating explanatory rationales sometimes interfered with precise defect classification, leading to verbose but inaccurate assessments of code quality issues.

Cross-model comparison of error patterns identified several categories of defects that consistently challenge all evaluated architectures. Complex multi-file dependencies, domain-specific security vulnerabilities, and subtle race conditions represented universal blind spots across all models. These findings suggest fundamental limitations in current LLM approaches that require architectural innovations or hybrid methodologies combining multiple analysis techniques for comprehensive coverage.

5. Discussion and Future Work

5.1. Implications for Automated Code Review Integration

The experimental results provide substantial evidence supporting the integration of LLM-based analysis into existing automated code review workflows, while highlighting critical considerations for successful deployment. The observed performance variations across defect categories suggest that optimal integration strategies should employ specialized models for different types of analysis, rather than relying on single general-purpose solutions. Organizations implementing these systems should prioritize security vulnerability detection using larger models like GPT-4, while leveraging lightweight alternatives like CodeBERT for syntax checking and code smell identification[27].

Practical deployment considerations extend beyond accuracy metrics to encompass computational requirements, processing latency, and integration complexity with existing development tools. The measured processing times indicate that real-time analysis during development requires careful model selection and potentially hybrid approaches that balance thoroughness with responsiveness. CodeBERT's superior processing speed makes it suitable for continuous integration scenarios, while comprehensive models like GPT-4 may be reserved for critical security reviews or release candidate analysis[28].

The reliability variations observed across multiple evaluation runs highlight the importance of ensemble approaches that combine predictions from multiple models to improve consistency and reduce individual model biases. Such strategies can leverage the complementary strengths of different architectures while mitigating their respective limitations, providing more robust defect detection capabilities than any single model alone.

5.2. Identified Challenges and Potential Solutions

Several fundamental challenges emerged from our analysis that require attention in future research and development efforts. The context limitation problem affects all evaluated models, where defects spanning multiple files or requiring understanding of large codebases exceed current model capabilities. This limitation particularly impacts architectural defect detection and complex security vulnerability identification that depends on system-wide understanding rather than local code analysis.

The semantic gap between syntactic code analysis and behavioral understanding represents another significant challenge, as current models struggle with defects requiring understanding of runtime behavior, performance characteristics, and dynamic interactions. Traditional static analysis tools face similar limitations, suggesting that hybrid approaches incorporating dynamic analysis techniques may provide more comprehensive solutions.

Domain-specific knowledge integration presents ongoing challenges, as general-purpose models lack understanding of specialized requirements in domains such as embedded systems, financial services, or healthcare applications. Fine-tuning approaches showed promise in our experiments but require substantial domain-specific datasets and expertise for effective implementation.

5.3. Future Research Directions and Recommendations

Several promising research directions emerge from our findings that could significantly advance the state of automated code review systems. Multi-modal approaches combining code analysis with documentation, issue tracking data, and developer communication could provide richer context for defect detection and classification. Such systems could leverage project-specific knowledge and historical patterns to improve accuracy and reduce false positive rates.

Adaptive learning systems that continuously improve based on developer feedback and actual defect resolution outcomes represent another valuable research direction. These systems could learn from their mistakes and adapt to specific project characteristics, coding standards, and team preferences over time, providing increasingly accurate and relevant analysis.

Research into explainable AI techniques specifically tailored for code analysis could significantly improve the practical utility of LLM-based systems by providing developers with clear rationales for detected issues and suggested remediation strategies. Current models show promise in generating explanations, but systematic evaluation of explanation quality and developer acceptance remains limited.

The development of specialized model architectures designed specifically for code analysis tasks, rather than adapting general-purpose language models, could yield significant performance improvements. Such architectures could incorporate programming language semantics, software engineering principles, and development workflow understanding as fundamental design elements rather than learned patterns.

Acknowledgments
I would like to extend my sincere gratitude to Me Sun, Zhen Feng, and Pengfei Li for their groundbreaking research on real-time AI-driven attribution modeling for dynamic budget allocation in U.S. e-commerce as published in their article titled[7] "Real-Time AI-Driven Attribution Modeling for Dynamic Budget Allocation in US E-Commerce: A Small Appliance Sector Analysis" in the Journal of Advanced Computing Systems (2023). Their insights and methodologies have significantly influenced my understanding of advanced techniques in real-time AI optimization and have provided valuable inspiration for my own research in automated code review systems.
I would like to express my heartfelt appreciation to Sida Zhang, Chenyao Zhu, and Jing Xin for their innovative study on lightweight AI frameworks for predictive supply chain risk management, as published in their article titled[23] "CloudScale: A Lightweight AI Framework for Predictive Supply Chain Risk Management in Small and Medium Manufacturing Enterprises" in Spectrum of Research (2024). Their comprehensive analysis and predictive modeling approaches have significantly enhanced my knowledge of scalable AI system architectures and inspired my research in efficient LLM deployment strategies for code analysis.

References

[1] Wang, Z., Wang, X., & Wang, H. (2024). Temporal Graph Neural Networks for Money Laundering Detection in Cross-Border Transactions. Academia Nexus Journal, 3(2).
[2] Liang, J., Fan, J., Feng, Z., & Xin, J. (2025). Anomaly Detection in Tax Filing Documents Using Natural Language Processing Techniques. Applied and Computational Engineering, 144, 80-89.
[3] Zhang, S., Mo, T., & Zhang, Z. (2024). LightPersML: A Lightweight Machine Learning Pipeline Architecture for Real-Time Personalization in Resource-Constrained E-commerce Businesses. Journal of Advanced Computing Systems, 4(8), 44-56.
[4] Fan, J., Trinh, T. K., & Zhang, H. (2024). Deep Learning-Based Transfer Pricing Anomaly Detection and Risk Alert System for Pharmaceutical Companies: A Data Security-Oriented Approach. Journal of Advanced Computing Systems, 4(2), 1-14.
[5] Ni, C., Qian, K., Wu, J., & Wang, H. (2025). Contrastive Time-Series Visualization Techniques for Enhancing AI Model Interpretability in Financial Risk Assessment.
[6] Ju, C., Jiang, X., Wu, J., & Ni, C. (2024). AI-Driven Vulnerability Assessment and Early Warning Mechanism for Semiconductor Supply Chain Resilience. Annals of Applied Sciences,

5(1).
[7] Sun, M., Feng, Z., & Li, P. (2023). Real-Time AI-Driven Attribution Modeling for Dynamic Budget Allocation in US E-Commerce: A Small Appliance Sector Analysis. Journal of Advanced Computing Systems, 3(9), 39-53.
[8] Trinh, T. K., & Zhang, D. (2024). Algorithmic Fairness in Financial Decision-Making: Detection and Mitigation of Bias in Credit Scoring Applications. Journal of Advanced Computing Systems, 4(2), 36-49.
[9] Rao, G., Wang, Z., & Liang, J. (2025). Reinforcement Learning for Pattern Recognition in Cross-Border Financial Transaction Anomalies: A Behavioral Economics Approach to AML. Applied and Computational Engineering, 142, 116-127.
[10] Wang, H., Wu, J., Ni, C., & Qian, K. (2025). Automated Compliance Monitoring: A Machine Learning Approach for Digital Services Act Adherence in Multi-Product Platforms. Applied and Computational Engineering, 147, 14-25.
[11] Chen, S., Li, X., Zhang, M., Jiang, E. H., Zeng, Q., & Yu, C. H. (2025). CARES: Comprehensive Evaluation of Safety and Adversarial Robustness in Medical LLMs. arXiv preprint arXiv:2505.11413.
[12] Chen, Y., Ni, C., & Wang, H. (2024). AdaptiveGenBackend A Scalable Architecture for Low-Latency Generative AI Video Processing in Content Creation Platforms. Annals of Applied Sciences, 5(1).
[13] Zhang, S., Feng, Z., & Dong, B. (2024). LAMDA: Low-Latency Anomaly Detection Architecture for Real-Time Cross-Market Financial Decision Support. Academia Nexus Journal, 3(2).
[14] Zhang, M., Heffernan, N., & Lan, A. (2023). Modeling and Analyzing Scorer Preferences in Short-Answer Math Questions. arXiv preprint arXiv:2306.00791.
[15] Li, M., Liu, W., & Chen, C. (2024). Adaptive Financial Literacy Enhancement through Cloud-Based AI Content Delivery: Effectiveness and Engagement Metrics. Annals of Applied Sciences, 5(1).
[16] Liu, P., Yan, X., Jiang, Y., & Xia, S. T. (2020, May). Deep flow collaborative network for online visual tracking. In ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 2598-2602). IEEE.
[17] Zhao, Y., Zhang, P., Pu, Y., Lei, H., & Zheng, X. (2023). Unit operation combination and flow distribution scheme of water pump station system based on Genetic Algorithm. Applied Sciences, 13(21), 11869.
[18] Kang, A., Xin, J., & Ma, X. (2024). Anomalous Cross-Border Capital Flow Patterns and Their Implications for National Economic Security: An Empirical Analysis. Journal of Advanced Computing Systems, 4(5), 42-54.
[19] Zhang, M., Baral, S., Heffernan, N., & Lan, A. (2022). Automatic short math answer grading via in-context meta-learning. arXiv preprint arXiv:2205.15219.
[20] Chand, R., Jain, P., Mathur, A., Raj, S., & Kanikar, P. (2023, March). Survey on Visual Speech Recognition using Deep Learning Techniques. In 2023 International Conference on

Communication System, Computing and IT Applications (CSCITA) (pp. 72-77). IEEE.
[21] Rao, G., Trinh, T. K., Chen, Y., Shu, M., & Zheng, S. (2024). Jump Prediction in Systemically Important Financial Institutions' CDS Prices. Spectrum of Research, 4(2).
[22] Wang, H., Qian, K., Ni, C., & Wu, J. (2025). Distributed Batch Processing Architecture for Cross-Platform Abuse Detection at Scale. Pinnacle Academic Press Proceedings Series, 2, 12-27.
[23] Zhang, S., Zhu, C., & Xin, J. (2024). CloudScale: A Lightweight AI Framework for Predictive Supply Chain Risk Management in Small and Medium Manufacturing Enterprises. Spectrum of Research, 4(2).
[24] Wang, Z., Trinh, T. K., Liu, W., & Zhu, C. (2025). Temporal Evolution of Sentiment in Earnings Calls and Its Relationship with Financial Performance. Applied and Computational Engineering, 141, 195-206.
[25] Chen, J., & Lv, Z. (2025, April). Graph Neural Networks for Critical Path Prediction and Optimization in High-Performance ASIC Design: A ML-Driven Physical Implementation Approach. In Global Conference on Advanced Science and Technology (Vol. 1, No. 1, pp. 23-30).
[26] Wang, Z., Zhang, M., Baraniuk, R. G., & Lan, A. S. (2021, December). Scientific formula retrieval via tree embeddings. In 2021 IEEE International Conference on Big Data (Big Data) (pp. 1493-1503). IEEE.
[27] Zhang, M., Wang, Z., Yang, Z., Feng, W., & Lan, A. (2023). Interpretable math word problem solution generation via step-by-step planning. arXiv preprint arXiv:2306.00784.
[28] Michael, S., Sohrabi, E., Zhang, M., Baral, S., Smalenberger, K., Lan, A., & Heffernan, N. (2024, July). Automatic Short Answer Grading in College Mathematics Using In-Context Meta-learning: An Evaluation of the Transferability of Findings. In International Conference on Artificial Intelligence in Education (pp. 409-417). Cham: Springer Nature Switzerland.