Journal of Information Technology and Computer Science
Volume 11, Number 2, August 2026, pp. 255-266
Journal Homepage: www.jitecs.ub.ac.id

Detection of God Class Smells in PHP Systems Using Object-Oriented Metrics

Naveed Toofani[1], Bayu Priyambadha[2], Achmad Arwan[3]

[1,2,3] Universitas Brawijaya, Malang
[1]naveedtoofani@student.ub.ac.id, [2]bayu_priyambadha@ub.ac.id, [3]arwan@ub.ac.id
*Corresponding Author

Received 24 April 2026; accepted 12 June 2026

Abstract. Software maintainability becomes difficult when object-oriented systems contain God Class code smells, where a single class handles excessive responsibilities and controls large portions of system behavior. Most existing God Class detection approaches use metric thresholds developed for statically typed languages such as Java, which may not accurately represent the architectural characteristics of dynamically typed PHP systems. This problem can reduce software maintainability, increase system complexity, and complicate future development and testing activities. This study applies a quantitative empirical research approach to evaluate metric-based God Class detection in PHP systems. Twelve open-source PHP projects containing 7,866 classes were collected from GitHub and analyzed using the PDepend static analysis tool. The extracted object-oriented metrics included Weighted Methods per Class (WMC), Number of Public Methods (NPM), Depth of Inheritance Tree (DIT), and Lines of Code (LOC). A class was classified as a God Class when at least three out of four threshold values were exceeded. The detection results were validated using statistical analysis, K-Means clustering, machine learning consistency validation, PHPMD comparison, and expert manual validation. The results identified 421 God Classes and showed that WMC, NPM, and LOC are strong indicators of God Class behavior in PHP systems, while DIT has lower influence due to framework-based inheritance structures. The study demonstrates that metric-based detection can effectively identify maintainability problems in PHP applications and provides a foundation for future PHP-specific threshold development.

Keywords: God Class, PHP, Object-Oriented Metrics, Static Analysis, Code Smell Detection

1 Introduction

Software maintainability has become one of the major challenges in modern software engineering because software systems continue to increase in size, complexity, and functionality. Large-scale object-oriented systems require continuous modification and maintenance throughout their lifecycle, which increases the possibility of poor design quality and technical debt. Poor software design can reduce code readability, increase system complexity, and make future development more difficult. One important indicator of poor design quality is the existence of code smells, which are structural weaknesses in software that negatively affect maintainability without immediately causing system failure [8], [15], [16].

Among various code smells, God Class is considered one of the most critical design problems in object-oriented software systems because a single class handles

excessive responsibilities and controls large portions of system behavior [13], [18]. A God Class usually contains high complexity, excessive public methods, low cohesion, and strong coupling with other classes. This condition makes testing, debugging, and software evolution more difficult because modifications in one class may affect multiple system components simultaneously [4], [8]. In large software systems, God Classes can gradually increase maintenance cost and reduce software quality over time.

Previous studies commonly detect God Class using object-oriented metrics such as Weighted Methods per Class (WMC), Number of Public Methods (NPM), Depth of Inheritance Tree (DIT), and Lines of Code (LOC) [13], [18], [20]. These metrics are widely used because they help measure class complexity, class size, inheritance structure, and responsibility distribution. However, most existing threshold values were originally developed for statically typed programming languages such as Java and C#. Modern PHP systems use dynamically typed architecture and commonly implement frameworks based on Model-View-Controller (MVC), dependency injection, and service-oriented structures. As a result, large controller and service classes may naturally contain higher metric values without necessarily representing poor software design.

The use of inappropriate threshold values in PHP systems may produce inaccurate God Class detection results. Some classes may be incorrectly classified as God Classes because framework architecture affects inheritance structure and class size differently from statically typed systems. This condition can increase false positive detection and reduce the reliability of metric-based detection methods in PHP applications. Therefore, evaluating whether traditional object-oriented metrics can effectively identify God Classes in PHP systems becomes an important research problem.

Several recent studies have investigated code smell detection using machine learning, metric analysis, and empirical validation techniques. Santos et al. [8] examined the relationship between code smells and software maintainability, while Gao et al. [18] analyzed object-oriented metrics for God Class detection. Alenezi et al. [1] and Yadav et al. [11] reviewed machine learning approaches for code smell detection, and Rio and Brito e Abreu [6] investigated PHP code smells in web applications. Although these studies provide important findings, most previous research mainly focuses on Java-based systems, while empirical evaluation of God Class detection in PHP systems remains limited.

Based on the existing literature, there is still limited research that evaluates whether traditional Java-based object-oriented metric thresholds can reliably detect God Classes in dynamically typed PHP systems. In addition, few studies combine statistical analysis, clustering analysis, machine learning consistency validation, tool-based validation, and expert manual validation to evaluate detection reliability in PHP applications. This creates a research gap regarding the effectiveness of metric-based God Class detection in modern PHP environments.

To address this problem, this study evaluates God Class detection in PHP systems using object-oriented metrics extracted through static analysis. The study applies threshold-based detection using WMC, NPM, DIT, and LOC metrics across twelve large open-source PHP projects. The detection results are validated using statistical analysis, clustering analysis, machine learning consistency validation, PHPMD comparison, and expert manual validation to examine the reliability of the proposed detection approach.

Therefore, the objective of this study is to evaluate the effectiveness of metric-based God Class detection in PHP systems and identify which object-oriented metrics serve as reliable indicators of God Class behavior in dynamically typed software environments. The findings of this study are expected to support future research in developing PHP-specific threshold standards for maintainability analysis and code smell detection.

Research Contributions

1. Evaluates the effectiveness of object-oriented metrics for detecting God Classes in PHP systems.
2. Applies a multi-metric threshold rule using WMC, NPM, DIT, and LOC metrics.
3. Combines statistical analysis, clustering analysis, machine learning consistency validation, PHPMD comparison, and expert manual validation.
4. Identifies WMC, NPM, and LOC as stronger indicators of God Class behavior in PHP systems.
5. Provides empirical evidence supporting future PHP-specific threshold development.

2 Method

2.1 Research Type

This research applies a quantitative empirical approach to evaluate the effectiveness of metric-based God Class detection in PHP systems. The analysis focuses on object-oriented metrics extracted from real-world open-source PHP projects to determine whether traditional metric thresholds can reliably identify God Classes in dynamically typed software environments. Several validation techniques, including statistical analysis, clustering analysis, machine learning consistency validation, tool-based validation, and expert manual validation, were used to evaluate detection reliability.

2.2 Research Procedure

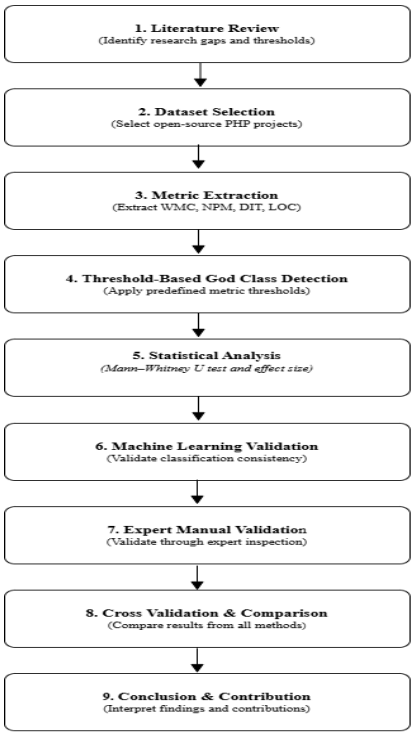

Figure 1. Research Procedure for God Class Detection

The research procedure consists of several stages, as illustrated in Figure 1. The first stage involves selecting open-source PHP projects from GitHub that represent different categories of real-world applications, including content management systems, e-commerce platforms, customer support systems, and framework libraries. The second stage involves extracting object-oriented metrics using the PDepend static analysis tool. The extracted metrics include Weighted Methods per Class (WMC), Number of Public Methods (NPM), Depth of Inheritance Tree (DIT), and Lines of Code (LOC).

The third stage applies a threshold-based detection rule to identify God Classes. A class is classified as a God Class when at least three out of four threshold values are exceeded. The fourth stage performs validation using statistical analysis, clustering analysis, machine learning consistency validation, PHPMD comparison, and expert manual validation. Finally, the detection results are analyzed to evaluate the effectiveness of object-oriented metrics for identifying God Classes in PHP systems.

2.3 Research Subjects and Dataset

The research subjects consist of twelve open-source PHP projects collected from GitHub, namely Bagisto, Invoice Ninja, Laravel Boilerplate, Laravel Framework, MantisBT, OpenCart, osTicket, phpMyAdmin, PrestaShop, Slim Framework, Sylius, and WordPress. These projects were selected because they represent different scales and architectural characteristics of modern PHP applications, including small, medium, and large systems.

A total of 7,866 classes were extracted from the selected projects. The dataset includes projects based on various architectural styles such as MVC frameworks, service-oriented systems, and modular web applications. The reliability and generalizability of the evaluation process were improved by the diversity of project sizes and architectural characteristics.

2.4 Data Collection Technique

Data collection was performed using static analysis through the PDepend tool [25]. The tool was used to extract object-oriented metrics from the selected PHP projects automatically. The extracted metrics include WMC, NPM, DIT, and LOC. These metrics were selected because previous studies identified them as important indicators for detecting God Class behavior in object-oriented systems [13], [18], [20].

Before analysis, Min-Max normalization was applied to reduce differences in metric scales and improve the reliability of clustering and machine learning validation processes.

2.5 Object-Oriented Metrics

The study uses four object-oriented metrics to evaluate class characteristics. WMC measures class complexity based on the number and complexity of methods. NPM measures the number of public methods and represents interface size and responsibility distribution. DIT measures inheritance depth and indicates the structural position of a class within the inheritance hierarchy. LOC measures the physical size of a class and helps identify excessive responsibility concentration.

2.6 God Class Detection Rule

The study applies threshold-based detection using traditional object-oriented metric thresholds derived from previous God Class detection studies. The thresholds used are WMC > 47, NPM > 10, DIT > 3, and LOC > 500. A class is classified as a God Class when at least three out of four threshold values are exceeded. This multi-metric rule helps reduce false positive detection by avoiding classification based on a single metric value.

2.7 Data Analysis Techniques

Several data analysis techniques were used to evaluate detection effectiveness. Descriptive statistics were used to summarize metric distributions across God Classes and Non-God Classes. The Mann–Whitney U test was applied to examine statistical differences between the two groups because object-oriented metric data commonly show non-normal distributions.

K-Means clustering was used to determine whether detected God Classes naturally formed high-complexity groups based on metric similarity. In addition, Decision Tree and Random Forest models were used for machine learning consistency validation to evaluate whether the selected metrics could establish stable classification boundaries.

2.8 Validation Methods

The study applies multiple validation methods to improve detection reliability. PHPMD comparison was used to compare the proposed detection results with an established PHP static analysis tool. Expert manual validation was also conducted by three software engineering experts to evaluate responsibility concentration, cohesion, excessive public methods, and maintainability characteristics in selected classes.

The combination of statistical analysis, clustering analysis, machine learning validation, tool-based validation, and expert review helps provide stronger empirical evidence regarding the effectiveness of metric-based God Class detection in PHP systems.

3 Results and Discussion

3.1 God Class Detection Results

The number of detected God Classes varied across the twelve PHP projects. WordPress, PrestaShop, and Invoice Ninja contained more God Classes because of their larger architectural complexity.

The analysis identified 421 God Classes among 7,866 evaluated classes. These classes represented 5.35% of the complete dataset. The Slim Framework and Laravel Boilerplate projects showed no detected God Classes while PrestaShop and WordPress both demonstrated much higher God Class detection rates. PrestaShop showed the highest percentage of detected God Classes at 18.55%, followed closely by WordPress with 18.42% and Invoice Ninja with 17.65%. These systems contain large modules and complex service structures. As a result, multiple classes handle different responsibilities.

The projects of Sylius OpenCart and Bagisto demonstrate lower percentage values because they distribute tasks more evenly throughout their systems or they implement better modular development methods. The results demonstrate that metric-based

threshold detection successfully identifies maintainability issues in PHP systems while showing how framework design affects the development of God Classes.

Table 1 Shows that WordPress and PrestaShop contained the highest percentage of detected God Classes.

Project	Total Classes	God Classes	Percentage (%)
Bagisto	974	13	1.33
Laravel Framework	1073	73	6.80
Invoice Ninja	85	15	17.65
Laravel Boilerplate	100	0	0.00
MantisBT	99	3	3.03
OpenCart	1465	15	1.02
osTicket	1264	67	5.30
phpMyAdmin	1150	59	5.13
PrestaShop	275	51	18.55
Slim Framework	55	0	0.00
Sylius	653	1	0.15
WordPress	673	124	18.42

3.2 Statistical Differences

Classes show structural and complexity differences from Non-God Classes. The method identifies large classes through threshold value detection but fails to demonstrate their existence as design issues. Framework design causes some classes to exceed metric values while maintaining proper maintainability. The statistical analysis indicates that God Classes exhibit higher complexity, larger size, and stronger responsibility concentration

Table 2 Indicates that WMC, NPM, and LOC showed significant differences between God Classes and Non-God Classes.

Metric	God Mean	Non-God Mean	p-value	Effect Size
WMC	196.12	11.43	<0.001	Large
NPM	34.92	3.59	<0.001	Large
LOC	1359.41	97.90	<0.001	Large
DIT	0.86	1.32	<0.01	Small

Object-oriented metric data requires the Mann–Whitney U test because it does not conform to normal distribution patterns. Software systems contain most classes with low values which creates a distribution where few classes possess extremely high values. The t-test and other parametric tests require normal distribution which makes them unsuitable for this situation. The Mann–Whitney U test is a non-parametric method that works better for comparing two independent groups with non-normal data. This makes it suitable for comparing God Classes and Non-God Classes.

The results indicate that God Classes exhibit higher WMC, NPM, and LOC values. These metrics demonstrated statistically significant differences between God Classes and Non-God Classes. The results demonstrate that God Classes exhibit higher complexity, larger size, and more public methods than standard classes. DIT

demonstrates statistical significance although its effect size remains weaker which makes it a less reliable measurement. The architectural framework of PHP systems determines inheritance depth instead of design deficiencies. The WMC and NPM and LOC metrics provide better detection capabilities for identifying God Classes within PHP systems.

3.3 Clustering Findings

The researchers applied clustering analysis to test whether the identified God Classes created an independent high-complexity group based on their metric measurements. Clustering groups classes based on similarities in WMC, NPM, DIT, and LOC values. The approach tests whether the detection rule based on thresholds successfully matches actual structural attributes of the dataset. The detection method shows greater accuracy when most God Classes belong to the high-complexity group.

The K-Means algorithm divided the 7,866 classes into three clusters which represented different levels of complexity that ranged from low to high. Cluster 0 contained 1,857 classes with 90 God Classes (4.8%), while Cluster 1 contained 3,303 classes with 109 God Classes (3.3%). The second cluster included 2,706 classes which had 222 God Classes (8.2%) making it the most populated cluster for discovered God Classes. The results demonstrate that the majority of God Classes were assigned to the high-complexity group.

The higher percentage of God Classes in Cluster 2 confirms that classes with large size, high complexity, and many public methods tend to form a distinct structural group. The clustering results support the effectiveness of the threshold-based detection method. Most detected God Classes belonged to the high-complexity cluster. The two clusters contained mainly standard classes which possessed minimal structural danger and demonstrated excellent duty sharing abilities.

The research results demonstrate that WMC NPM and LOC serve as effective metrics for detecting God Classes within PHP systems because these metrics directly impact the identification of high-complexity code clusters. DIT showed less impact because inheritance depth is often affected by framework architecture rather than poor design quality. The proposed detection method receives strong validation through clustering analysis which establishes that it functions reliably in large PHP systems.

Table 3 Shows that most God Classes were grouped within the high-complexity cluster.

Cluster	Total Classes	God Classes	Percentage (%)
Cluster 0	1857	90	4.8
Cluster 1	3303	109	3.3
Cluster 2	2706	222	8.2
Total	7866	421	—

3.4 Machine Learning Consistency Validation

The researchers used machine learning validation to test whether their selected object-oriented metrics could create stable decision boundaries which would separate God Classes from Non-God Classes. The primary objective of this stage was to validate the internal consistency of the threshold-based detection rule instead of developing a new prediction system. The proposed rule demonstrates logical strength and structural value because WMC NPM DIT and LOC successfully distinguish between the two groups. The process demonstrates that researchers established God Classes through purposeful threshold selection instead of using random classification methods. Machine learning was used as a validation technique rather than the primary detection method.

The researchers selected Decision Tree and Random Forest because both methods are widely used in software engineering classification tasks and provide strong interpretability [1], [9], [11], [14]. The system used WMC NPM DIT and LOC as input features while the threshold-based detection rule generated the target labels. The testing team selected only those classes which had complete metric values to achieve steady training and accurate evaluation results.

The results showed perfect performance, where accuracy, precision, recall, and F1-score all reached 1.00 for both models. This should not be interpreted as better predictive performance compared to other machine learning methods. The perfect results demonstrate that the threshold rule creates a precise classification boundary because the models used labels derived from that same threshold rule. The system shows strong internal logical consistency, but it does not demonstrate better predictive performance than competing methods.

Future research should compare this rule-based method with expert-labeled datasets and other machine learning techniques such as Support Vector Machine, XGBoost, and Neural Networks. The use of manually verified God Class labels for cross-project validation will establish more reliable evidence about predictive performance. The current machine learning results function as consistency validation which demonstrates the proposed PHP system detection method maintains reliable performance.

Table 4 demonstrates strong consistency between the threshold-based labels and the machine learning models.

Model	Accuracy	Precision	Recall	F1-Score
Decision Tree	1.00	1.00	1.00	1.00
Random Forest	1.00	1.00	1.00	1.00

3.5 Tool-Based Validation Using PHPMD

The researchers verified their metric-based detection system through testing which included PHPMD results, a popular static analysis tool that evaluates PHP code quality [6], [20]. The tool PHPMD detects design flaws which occur when a system contains too many public methods and the class structure becomes complicated.

PHPMD identified 57 classes showing strong indications of God Class attributes through their excessive class complexity and their numerous public methods. The proposed threshold-based method found 56 classes which matched 421 God Classes that were identified during the detection process.

Table 5 indicates strong agreement between the proposed detection method and PHPMD.

Description	Count
God Classes detected by proposed approach	421
Classes flagged by PHPMD	57
Overlapping validated classes	56
PHPMD coverage of proposed approach (%)	98.2%

The detection method achieved 98.2% validation coverage because the proposed detection rule matched results from established static analysis tool. The single mismatch occurred because PHPMD applies individual rule thresholds, while the proposed method uses a multi-metric rule requiring at least three threshold violations.

The two methods produce different results because their class identification criteria function differently. The proposed method uses multiple metrics to evaluate structural evidence, whereas PHPMD detects strong warnings based on single class indicators.

The metric-based detection system proved reliable through these results, while the proposed method showed strong compatibility with actual industry tools that assess software maintainability in PHP systems.

3.6 Manual Validation with Expert Confirmation

The assessment of class existence as a God Class requires validation through manual checks because automated metric evaluation lacks sufficient capability to make this determination [4], [21], [23]. The threshold values of some classes exceed their limits because framework design, service responsibilities and MVC architectural patterns determine their operational structure. The expert review process provides a solution to assess selected classes through their actual duties and their ability to work together and their capacity to be maintained. The detection method needs this step to achieve better credibility while it decreases the occurrence of false positive results. The detected classes received verification which confirmed their status as actual design issues. Three PHP software engineering experts who possess object-oriented design and code review and software maintainability analysis expertise performed the manual validation. The experts selected for the evaluation process possessed practical experience in assessing class structure and responsibility distribution and maintainability testing of actual PHP systems. The experts conducted independent reviews of separate class groups to eliminate bias and enhance the validation process. The assessment used a structured checklist to examine responsibility concentration together with mixed concerns and excessive public methods and Single Responsibility Principle (SRP) breaches. The evaluation process established a more organized method which executed consistently throughout all evaluation sessions.

The team examined 24 selected classes which they chose from various PHP projects for their manual evaluation work. The class selection contained eight distinct God Classes and eight distinct Non-God Classes and eight classes which showed uncertain dependence on framework systems. The research team received improved study results because their larger sample group not only included clear cases but also complex scenarios which included classes that used framework design elements to affect metric outcomes. The experts reviewed each class independently and then compared their evaluations with the metric-based detection results. The detection rule testing process helped determine whether the proposed detection rule accurately measured actual software design quality.

Table 6 shows high agreement between expert evaluation and the proposed detection method.

Category	Total Classes	Agreement	Agreement (%)
Clear God Classes	8	8	100%
Non-God Classes	8	8	100%
Borderline Classes	8	6-7	75–87.5%
Total	24	22-23	91.6–95.8%

The results demonstrated a strong connection between the metric-based detection method and expert assessment. All experts confirmed the existence of God Classes and Non-God Classes but they showed partial disagreement about borderline framework-driven classes because large classes retained good cohesion through

framework responsibilities. The results demonstrate that large class size does not determine design quality because architects should use their architectural knowledge to detect problems. The manual validation process gains higher reliability through multiple experts and a larger number of reviewed classes which also strengthens the God Class detection method for PHP systems.

4 Conclusion

The research tested how well object-oriented metrics detect God Classes in PHP systems by applying these metrics to 12 large open-source projects which contained 7866 classes. The analysis discovered 421 God Classes which accounted for 5.35 percent of all entries in the database. The statistical analysis revealed that WMC NPM and LOC served as the primary indicators which proved God Class behavior because these metrics displayed substantial differences between God Classes and Non-God Classes. DIT showed weaker influence and was less reliable as a standalone metric because inheritance depth in PHP frameworks often depends on architectural design rather than poor software quality.

The proposed detection method acquired reliability through four different validation methods which included clustering analysis and machine learning consistency validation and PHPMD comparison and expert manual validation. The PHPMD comparison showed 98.2% overlap which indicates strong consistency with an established industry tool. The researchers conducted the study using machine learning to verify rule consistency, while traditional machine learning methods used predictive classification for their operations. Previous studies which employed Support Vector Machine Random Forest and ensemble learning models mainly investigated Java systems while this research delivers PHP-specific empirical evaluation through multiple validation approaches.

The findings indicate that framework architecture influences metric interpretation in PHP systems. The WordPress, PrestaShop and Laravel systems contain extensive service classes and controller components which exceed LOC and NPM limits because their MVC framework design and operational functions. The only time a large class should be viewed as a God Class is when the class exhibits both low cohesion and mixed responsibilities and unmanageable complexity. Large class size alone should not be considered a design problem. This approach helps people identify which framework-based classes should be classified as standard classes and which classes should be treated as God Classes.

The research should aim to establish PHP-specific threshold standards that apply to various frameworks and architectural systems. The three optimization techniques which include Genetic Algorithm and Particle Swarm Optimization (PSO) and Differential Evolution enable the development of precise threshold measurements through expert-annotated datasets [22], [24]. The cross-project validation process uses manually verified God Class labels to improve detection accuracy through comparison with XGBoost and Support Vector Machine and Neural Networks. The research will identify framework differences between WordPress and Laravel and Symfony and other PHP systems to develop detection methods that meet current PHP development needs.

References

1. M. Alenezi, A. Agrawal, and K. Chaturvedi, “A systematic literature review on machine learning techniques for code smell detection,” IEEE Access, vol. 9, pp. 154594–154612, 2021.V.

2. N. A. A. Khleel and K. Nehéz, “Improving accuracy of code smells detection using machine learning with data balancing techniques,” The Journal of Supercomputing, vol. 80, pp. 21048–21093, 2024.
3. Z. Li, P. Liang, and P. Avgeriou, “Architectural technical debt identification based on machine learning,” Journal of Systems and Software, vol. 165, Art. no. 110566, 2020.
4. J. Pantiuchina, A. Zaidman, and G. Bavota, “Improving code review effectiveness through code smell awareness,” Empirical Software Engineering, vol. 25, no. 5, pp. 3996–4033, 2020.
5. R. S. Rao, S. Dewangan, A. Mishra, and M. Gupta, “Dealing with class imbalance in code smell severity detection using PCA-based feature selection,” Scientific Reports, vol. 13, Art. no. 16245, 2023.
6. A. Rio and F. Brito e Abreu, “PHP code smells in web applications: Evolution, survival, and anomalies,” Journal of Systems and Software, vol. 200, Art. no. 111644, 2023.
7. R. Sandouka and H. Aljamaan, “Cross-project code smell detection using machine learning techniques,” Applied Sciences, vol. 12, no. 18, Art. no. 9345, 2022.
8. J. A. Santos, R. Oliveira, and E. Figueiredo, “On the relationship between code smells and software maintainability,” Information and Software Technology, vol. 137, Art. no. 106598, 2021.
9. A. Tahir, G. Rasool, and M. A. Alqarni, “Ensemble learning approach for software code smell detection,” IEEE Access, vol. 10, pp. 51234–51247, 2022.
10. R. Verdecchia, F. Ricca, and M. Torchiano, “An empirical study on the distribution and evolution of code smells in open-source systems,” Software Quality Journal, vol. 29, pp. 1387–1412, 2021.
11. P. S. Yadav, R. S. Rao, A. Mishra, and M. Gupta, “Machine learning-based methods for code smell detection: A survey,” Applied Sciences, vol. 14, no. 14, Art. no. 6149, 2024.
12. Y. Zhang, S. Wang, and D. Lo, “Deep learning-based code smell detection,” Information and Software Technology, vol. 144, Art. no. 106765, 2022.
13. A. Kaur, P. Singh, and H. Singh, “Software metrics-based prediction of God Class code smell using classification models,” Journal of King Saud University – Computer and Information Sciences, vol. 33, no. 9, pp. 1036–1045, 2021.
14. M. Y. Mhawish and M. Gupta, “Predicting code smells using machine learning techniques and software metrics,” Journal of Computer Science and Technology, vol. 35, no. 6, pp. 1234–1252, 2020.
15. F. Palomba, G. Bavota, M. Di Penta, and R. Oliveto, “Exploring the impact of code smells on software change-proneness,” IEEE Transactions on Software Engineering, vol. 46, no. 5, pp. 493–509, 2020.
16. T. Sharma and D. Spinellis, “A survey on software smells,” Journal of Systems and Software, vol. 138, pp. 158–173, 2020.
17. M. Ribeiro, E. Figueiredo, and A. Garcia, “Investigating the persistence of code smells in evolving software systems,” Empirical Software Engineering, vol. 26, pp. 1–38, 2021.
18. J. Gao, H. Li, and T. Chen, “An empirical analysis of object-oriented metrics for detecting God Class design smell,” Software Quality Journal, vol. 30, pp. 845–872, 2022.
19. M. Aljamaan and R. Sandouka, “A systematic review on software code smells,” International Journal of Electrical and Computer Engineering, vol. 15, no. 3, pp. 3010–3027, 2025.

20. F. Fontana, V. Ferme, and M. Zanoni, “Automatic detection of bad smells in code: An experimental assessment,” Journal of Object Technology, vol. 15, no. 1, pp. 1–38, 2021.
21. M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, “A systematic literature review on the code smells datasets and validation mechanisms,” arXiv preprint arXiv:2306.01377, 2023.
22. B. Aljohani, “A novel framework for code smell detection via dynamic analysis,” PeerJ Computer Science, vol. 12, Art. no. e3642, 2026
23. A. Siddiqui, A. Hussain, and A. Ali, “A comparative analysis of code smell detection,” Journal of Computer and Information Sciences, vol. 5, no. 4, pp. 44–58, 2025.
24. R. Maini and P. Gupta, “An optimization-based code smell refactoring sequencing approach,” Journal of Computational and Cognitive Engineering vol. 3, no. 2, pp. 101–116, 2024
25. M.W.Schmid, “PDepend: PHP Depend,” Available: https://pdepend.org. Accessed: Jan. 2026
26. M. Rahmaty, “Machine learning with big data to solve real-world problems”.