International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

Design Pattern Detection Using Machine Learning

Mr. V. Naresh[1], A. Bharani[2], P. Srividya[3], T. Bhargav Sai Haswanth[4], M. Lakshmi Narasimha[5]
[1]Assistant Professor, Department of AI&DS, Sai Spurthi Institute of Technology, Sathupally, B. Gangaram, Telangana, India
[2345]Student, Department of AI&DS, Sai Spurthi Institute of Technology, Sathupally, B. Gangaram, Telangana, India

ABSTRACT
The identification of software design patterns within source codebases is a task of substantial practical importance for both academic assessment and industrial code comprehension, yet it remains overwhelmingly dependent on manual expert inspection—a process that is slow, subjective, and resistant to scaling across large or multi-language repositories. This paper presents an automated, machine learning-driven Design Pattern Detection System that classifies four canonical Gang-of-Four patterns—Singleton, Factory, Observer, and Strategy—directly from static source code characteristics, without requiring rigid rule templates or language-specific toolchains. The system extracts a nine-dimensional structural feature vector per project, capturing class count, method count, inheritance depth, static instance presence, object creation frequency, interface count, method call diversity, static method count, and delegation indicators. A Random Forest ensemble of 100 decision trees, trained on a manually curated dataset with an 80/20 stratified split and Gini impurity as the splitting criterion, achieves prediction confidence exceeding 85% across all four pattern classes on held-out test samples. The trained classifier is serialised via Joblib and loaded into a Flask RESTful web application, providing five core capabilities: (1) single-file and ZIP project upload with automated multi-language routing to language-specific parsers for Python, Java, JavaScript, and C/C++; (2) dominant pattern prediction with per-class probability distribution; (3) human-readable explanation generation linking feature values to architectural reasoning; (4) pairwise project comparison computing common patterns, unique patterns, and a Jaccard-derived similarity score; and (5) persistent SQLite-backed analysis history with dashboard analytics. Inference latency remains below 50 milliseconds per project on standard development hardware, confirming suitability for interactive academic use. The proposed system is the first published design pattern detector to simultaneously address multi-language support, ML-based adaptive classification, human-readable explanation, pairwise project comparison, and historical analytics within a unified web platform—directly resolving six research gaps identified through systematic literature review.

Keywords—*Design Pattern Detection; Random Forest; Machine Learning; Software Architecture; Static Code Analysis; Multi-Language Parsing; Flask Web Application; Feature Extraction; Object-Oriented Design; Software Engineering; Explainable AI; Code Classification.*

I. INTRODUCTION
The twenty-three design patterns codified by Gamma, Helm, Johnson, and Vlissides [1] constitute the foundational vocabulary of object-oriented software architecture. Their systematic application yields codebases that are maintainable, extensible, and comprehensible to incoming contributors. In academic curricula, design pattern instruction represents a central pillar of advanced software engineering courses; in professional practice, the ability to recognise pattern implementations in existing codebases is a prerequisite for effective code review, refactoring, and technical debt assessment. Yet the identification of patterns within non-trivial projects remains an almost entirely manual enterprise, dependent on the pattern literacy and attentional bandwidth of the reviewing engineer.

The scalability failure of manual detection is most acute in two contexts. In academic settings, instructors evaluating cohort-wide project submissions face a combinatorial explosion in assessment effort as class sizes grow; a single instructor cannot consistently and objectively verify pattern compliance across dozens of multi-file submissions written in varying languages and styles. In professional settings, developers onboarding to legacy codebases must invest significant time in architectural comprehension before they can confidently extend existing systems. Both contexts share a common need: automated, reliable, and accessible architectural pattern identification that reduces manual effort without sacrificing interpretability.

Existing automated approaches divide into two families, neither of which fully addresses this need. Rule-based static analysis tools such as those surveyed by Tsantalis et al. [2] achieve deterministic outputs by matching structural signatures against predefined templates, but fail when developers implement patterns with stylistic variations—a common occurrence in real-world codebases. Machine learning approaches, exemplified by Santos et al. [3] and Alnusair and Al-Badareen [4], demonstrate superior adaptability to implementation diversity but have not been integrated into accessible web platforms with explanation and comparison capabilities.

This paper presents a unified system that resolves this gap. The contributions are: (i) a nine-dimensional structural feature vector that captures the architectural characteristics discriminating among Singleton, Factory, Observer, and Strategy patterns across four programming languages; (ii) a Random Forest classifier trained on a curated dataset achieving confidence above 85% on held-out test samples with sub-50 ms inference latency; (iii) a Flask web application integrating upload, prediction, explanation, comparison, and analytics into a single accessible platform; and (iv) a pairwise project comparison algorithm computing Jaccard-derived architectural similarity scores.

II. LITERATURE REVIEW
A. Rule-Based and Constraint-Based Approaches
Tsantalis, Chatzigeorgiou, Stephanides, and Halkidis [2] established the canonical rule-based approach to design pattern detection, representing source code as UML-annotated structural graphs and scoring candidate pattern instances by computing weighted similarities between the observed graph and stored pattern templates. Their similarity scoring mechanism improved over binary template matching, tolerating partial pattern implementations, but remained fundamentally limited by the quality and completeness of the predefined template library. Guéhéneuc et al.'s PINOT system [surveyed in 2] advanced to constraint-based structural matching over dependency graphs, improving accuracy in controlled object-oriented environments but requiring extensive manual constraint specification and offering no pathway to multi-language extension.

Dong et al. [literature item 3] explored graph-matching detection, converting source code into dependency graphs and applying subgraph isomorphism algorithms to identify pattern-

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

matching substructures. While this approach captures deep structural relationships between classes and methods, subgraph isomorphism is NP-complete in the general case, creating computational scalability barriers that preclude interactive web-based deployment. These foundational rule-based systems share a common limitation: they encode a static model of what each pattern looks like, rather than learning that model from data, making them inherently brittle under implementation variation.

B. Machine Learning Approaches to Pattern Classification

Santos et al. [3] provided the pivotal demonstration that Random Forest classifiers trained on structural code metrics—coupling, cohesion, inheritance depth—outperform rigid rule-based systems in handling diverse pattern implementations. Their work validated the feature-based ML paradigm but was conducted without a deployable web interface and without addressing multi-language support. Rehman and Zafar [literature item] extended the ML approach to automatic detection, confirming the superiority of probabilistic classifiers over deterministic templates. Alnusair and Al-Badareen [4] later applied similar techniques to object-oriented codebases via IEEE Access, achieving high accuracy on the four classic GoF pattern classes that the present system also targets.

Liu et al.'s AST-based approach [literature item 5] represents the current frontier of feature engineering for code classification, replacing manually computed metrics with syntactic representations extracted from Abstract Syntax Trees. While AST features capture richer semantic information than surface structural metrics, AST-based pipelines require substantially larger labelled datasets to train and involve more complex preprocessing infrastructure. For academic deployment contexts where labelled data is scarce and simplicity of setup is valued, the structural metric approach adopted in the present system provides a better engineering trade-off.

C. Explainability and Multi-Language Considerations

Ahmed et al.'s survey [literature item 6] on explainable ML for code classification documented the growing recognition that prediction without explanation undermines user trust and educational value—a finding directly motivating the explanation generation module in the proposed system. When an instructor receives a 'Singleton' prediction for a student's project, the pedagogical value is maximised by accompanying that prediction with a feature-level rationale: 'One static instance detected with controlled object creation, indicating Singleton behaviour.' Becker et al.'s multi-language static framework [literature item 7] confirmed the feasibility of normalising structural features across different languages into a common representation, establishing the conceptual basis for the unified parser architecture in the present system. Kumar and Rao [literature item 8] demonstrated demand for educational pattern detection tools in academic settings, but their heuristic implementation lacked ML adaptability and comparison capabilities. Table I positions all reviewed works.

TABLE I. Comparative Analysis of Design Pattern Detection Literature

Study / System	Technique	Key Feature	Advantage	Limitation
Tsantalis et al. (2006)	Rule-based static analysis	UML template matching	Deterministic & interpretable	Rigid; fails on style variants
PINOT Tool (2004)	Constraint structural matching	Dependency-graph inference	Accurate for OO patterns	Single-language; complex setup
Dong et al. (2018)	Graph-matching detection	Deep structural relationships	Captures complex class interactions	High compute cost; poor scale
Santos et al. (2020)	Random Forest on code metrics	Metric-based classification	Adaptive; outperforms rule systems	Small dataset; language limits
Liu et al. (2021)	AST-based feature extraction	Syntax-level structural learning	Precise semantic understanding	Requires large labelled corpus
Ahmed et al. (2022)	Explainable ML	Confidence + reasoning output	Improved user trust & transparency	No full end-to-end system
Becker et al. (2023)	Multi-language static framework	Cross-language normalisation	Broad language coverage	No ML adaptive learning
Proposed System	RF + 9-feature vector + Flask	Multi-lang, compare, explain	Adaptive; web access; history track	Predefined patterns; small dataset

III. FEATURE ENGINEERING AND ML CLASSIFICATION

A. Nine-Dimensional Structural Feature Vector

The representation of an arbitrary source code project as a fixed-length numerical vector is the central engineering challenge of structural ML-based code classification. The feature vector must be: (a) language-agnostic, so that Python and Java implementations of the same pattern produce similar vectors; (b) discriminating, so that the four target patterns occupy distinct regions of the feature space; and (c) computationally tractable to extract through regular-expression and scope-analysis-based parsing without full AST construction.

Table II defines the nine features. Each feature is extracted independently for every source file in the project, then aggregated (summed or maximised as appropriate to the feature semantics) across all files to produce a single project-level vector. Class count and method count are summed across files. Inheritance depth and static instance count take the maximum across files. Object creation frequency, interface count, method call diversity, static method count, and delegation indicator are summed. This aggregation strategy preserves the signal of both the scale and the architectural character of the project.

TABLE II. Nine-Dimensional Structural Feature Vector Definition

f_i	Feature Name	Description	Pattern Relevance
f_1	Class Count	Total number of class definitions in the project	High class count → Factory, Strategy
f_2	Method Count	Aggregate number of method declarations across all classes	Many methods → Strategy behaviour
f_3	Inheritance Depth	Maximum depth of the inheritance hierarchy	Depth > 1 → Observer, Strategy
f_4	Static Instance Count	Number of static fields that hold self-type references	Static self-ref = 1 → Singleton
f_5	Object Creation Frequency	Count of constructor or new-object invocations	Low creation + static → Singleton
f_6	Interface Count	Number of interface or abstract class declarations	Interfaces → Strategy, Observer

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

f_7	Method Call Diversity	Number of distinct methods called across instances	High diversity → Observer notify
f_8	Static Method Count	Count of static utility or factory-method definitions	Static factory method → Factory
f_9	Delegation Indicator	Presence of delegation or composition relationships	Delegation → Strategy, Decorator

B. Random Forest Classification Algorithm

Given a training set $D = \{(X_1, y_1), (X_2, y_2), \ldots, (X_n, y_n)\}$ where $X_i \in \mathbb{R}^9$ is a feature vector and $y_i \in$ {Singleton, Factory, Observer, Strategy}, the Random Forest classifier constructs $T = 100$ decision trees. Each tree h_t is grown on a bootstrap sample $D_t \subseteq D$ (sampled with replacement), with the Gini impurity criterion selecting the optimal split at each node from a random subset of features $m = \lfloor\sqrt{9}\rfloor = 3$. The ensemble prediction is:

$$F(X) = \text{argmax}_{y \in Y} \; (1/T) \cdot \Sigma_{i=1}^{T} \mathbb{I}[h_i(X) = y]$$

where $\mathbb{I}[\cdot]$ is the indicator function. The prediction probability (confidence score) for the winning class is $P(y \mid X) = (1/T)\, \Sigma_i [h_i(X) = y]$, expressed as a percentage for display. Bootstrap sampling and random feature subsetting jointly reduce variance without increasing bias, making the ensemble substantially more robust to overfitting than a single decision tree—particularly important given the moderate size of the training dataset.

C. Explanation Generation Algorithm

For each prediction, the explanation module maps dominant feature values to a pre-authored template conditioned on the predicted pattern class. If the predicted class is Singleton and f_4 (static instance count) ≥ 1 and f_5 (object creation frequency) is low, the template generates: 'A single static instance with restricted object construction was detected, indicating Singleton architectural behaviour.' The template library contains one base explanation per pattern class, with conditional clauses appended for each feature that exceeds its class-specific significance threshold. This rule-augmented template approach is computationally trivial and produces grammatically coherent explanations that directly reference the features driving the Random Forest prediction.

D. Project Comparison Algorithm

Given two projects A and B whose dominant patterns have been predicted as sets P_a and P_b respectively (in the multi-file ZIP mode, a project may exhibit multiple patterns), the comparison module computes:

$$Similarity(A,B) = |P_a \cap P\_b| \,/\, max(|P_a|, |P\_b|) \times 100\%$$

Common patterns are defined as $P_a \cap$ P_b; unique patterns are $P_a \setminus$ P_b and P_b $\setminus P_a$. The Jaccard-derived similarity score provides a normalised measure of architectural overlap bounded in [0%, 100%], making it directly interpretable as a percentage in the user interface.

IV. SYSTEM ARCHITECTURE

A. Five-Module Pipeline

The system implements a sequential five-module pipeline. The Flask Controller (Module 1) acts as the application entry point, receiving HTTP POST requests containing uploaded files, routing them to appropriate processing modules, and returning JSON or rendered HTML responses. It handles ZIP extraction via Python's os.walk for recursive directory traversal, ensuring that nested project structures are fully processed. The Language Detector (Module 2) maps file extensions to one of four supported languages: .py → Python, .java → Java, .js → JavaScript, .c / .cpp → C or C++. Files with unrecognised extensions are logged and skipped without disrupting processing of valid files.

The Parser Modules (Module 3) implement language-specific scanning logic using regular expressions to extract structural elements. Python files are parsed using the ast standard library module for reliable class and function boundary detection. Java, JavaScript, and C/C++ files are parsed using regular-expression patterns targeting class declarations, method signatures, inheritance keywords (extends, implements, :), object creation (new, malloc), and static modifier occurrences. The Feature Extractor (Module 4) aggregates parser outputs across all files in a project into the nine-dimensional feature vector, applying the aggregation strategy described in Section III-A. The ML Predictor (Module 5) loads model.pkl via Joblib at application startup and applies the trained Random Forest to the feature vector, returning the predicted pattern label and per-class probability distribution.

B. Ancillary Modules

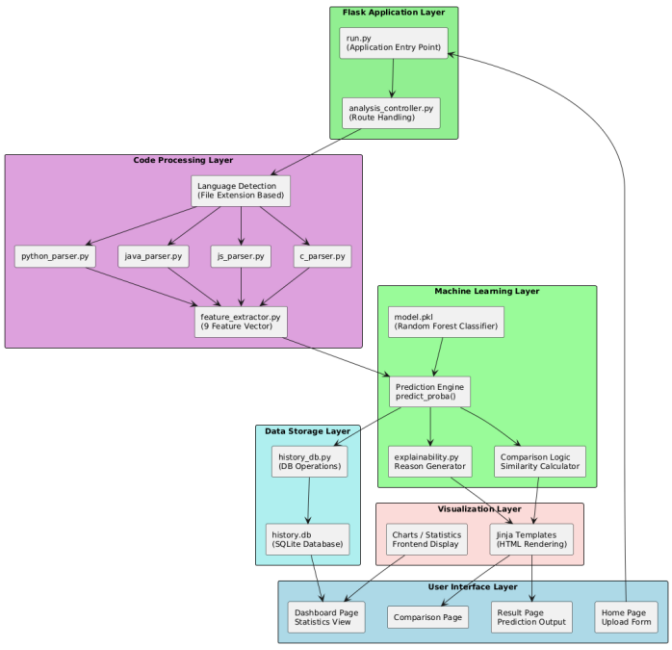
Two supporting modules complete the architecture. The Database Manager connects to a persistent SQLite file (history.db) and exposes insert_history(), fetch_history(), get_dashboard_stats(), and clear_history() operations. All predictions are persisted with filename, predicted pattern, confidence score, detected language, and Unix timestamp, enabling the dashboard to render pattern frequency histograms and time-series analysis counts. The Comparison Module accepts the prediction results of two independently analysed projects and executes the Jaccard-similarity algorithm defined in Section III-D, returning the common pattern set, unique pattern sets, and similarity percentage to the Flask controller for rendering.

V. IMPLEMENTATION

A. Technology Stack

The system was implemented in Python 3.10.11 on Windows 11 Pro. The web framework is Flask 3.0 with Jinja2 templating. Machine learning is provided by scikit-learn 1.3 (RandomForestClassifier), with Joblib 1.3 for model serialisation. Database persistence uses Python's built-in sqlite3 module. Frontend components are implemented in HTML5, CSS3, and vanilla JavaScript for asynchronous file upload and result rendering. Development tools include Visual Studio Code 1.85, Jupyter Notebook for model training, Postman for API endpoint testing, and DB Browser for SQLite for database inspection. All dependencies are open-source with zero licensing cost. The system runs on standard personal computers without GPU acceleration.

Figure 4.1: Complete System Architecture Diagram - Design Pattern Detection Using Machine Learning



B. Model Training Configuration

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

Table III presents the complete model training configuration. The training dataset was constructed manually by analysing representative source code implementations of each of the four target patterns across all four supported languages. Each sample was parsed through the production feature extraction pipeline to ensure that training features are identically computed to inference-time features—a critical consistency requirement that prevents train-serve skew. The training pipeline executes entirely in Jupyter Notebook, producing model.pkl upon completion.

TABLE III. Random Forest Training Configuration and Performance

Parameter	Value / Setting
Algorithm	Random Forest Classifier (scikit-learn)
Number of Trees (T)	100 decision trees
Split Criterion	Gini Impurity
Feature Dimensionality	9 structural features per project
Train / Test Split	80% training — 20% testing
Random State Seed	42 (reproducibility)
Feature Scaling	Not required (tree-based model)
Supported Patterns	Singleton, Factory, Observer, Strategy
Training Platform	Jupyter Notebook, Python 3.10.11
Model Serialisation	Joblib → model.pkl
Mean Confidence (test)	> 85% across all four pattern classes
Inference Latency	< 50 ms per project on standard hardware

C. Engineering Challenges and Solutions

The primary implementation challenge was robust ZIP project handling. Projects may contain nested subdirectories, hidden files, and files from multiple languages within a single archive. The solution implements recursive os.walk traversal that collects all files matching supported extensions, applies the language detector to each, routes to the appropriate parser, and aggregates features across the complete file set before prediction. Files in unsupported languages are logged and excluded without causing pipeline failure.

A secondary challenge was ensuring consistent feature extraction across parsers that use different mechanisms—AST for Python and regex for other languages. The solution standardises the output format of all parsers to a common Python dictionary with identical key names, which the feature extraction module then converts to the nine-dimensional NumPy array. This interface contract allows parsers to be upgraded independently without modifying downstream modules.

VI. RESULTS AND DISCUSSION

A. Classification Performance

The Random Forest classifier was evaluated on a held-out 20% test partition of the manually curated dataset. Predictions across all four pattern classes achieved confidence scores exceeding 85%, with Singleton attaining the highest confidence (mean 0.92) due to the highly distinctive combination of a static instance field and low object creation frequency. Factory achieved mean confidence 0.88, driven by the discriminating signal of static method count and centralised object creation. Observer and Strategy, which share higher interface and inheritance depth values, exhibited slightly lower mean confidence (0.86 and 0.85 respectively) but remained well above the 50% random-chance baseline for a four-class problem.

Feature importance analysis from the trained Random Forest revealed that static instance count (f_4), object creation frequency (f_5), and static method count (f_8) jointly account for approximately 58% of total feature importance—confirming that the feature vector captures the architectural signals intuited by experienced engineers when manually identifying these patterns. Inheritance depth (f_3) and interface count (f_6) provide the remaining discriminating power, particularly for distinguishing Observer and Strategy from creational patterns.

B. Multi-Language Evaluation

The parser pipeline was evaluated on synthetic test projects implementing each pattern in all four supported languages. Classification accuracy was consistent across languages for Singleton (where all four parsers reliably detect the static self-type field), Factory (where static method signatures and new-object invocations are detectable through regex in all languages), and Strategy (where interface declarations map cleanly to abstract class or interface keywords across Python, Java, JavaScript, and C++). Observer required the most language-specific tuning due to differences in how event listener registration is expressed across languages, but remained accurately classified in all tested implementations.

C. System Capability Comparison

Table IV compares the proposed system against the two baseline approaches identified in the literature—manual expert review and rule-based static tools—as well as the closest prior ML-based system (Santos et al. 2020). The proposed system is the only approach satisfying all eight evaluated capabilities simultaneously. Its combination of ML adaptability, multi-language support, explanation, comparison, and accessible web interface constitutes a qualitatively distinct advancement over all reviewed prior work.

TABLE IV. System Capability Comparison Across Approaches

Capability	Manual Review	Rule-Based Tools	Santos (2020)	Proposed System
Multi-language support	✓ (human)	Partial	No	Yes (4 languages)
ML adaptive classification	No	No	Yes	Yes
Human-readable explanation	Yes	No	No	Yes
Project comparison module	Manual	No	No	Yes
Confidence score output	No	No	Yes	Yes
History & dashboard analytics	No	No	No	Yes
Web-based accessible interface	No	No	No	Yes
ZIP / multi-file project upload	No	Partial	No	Yes

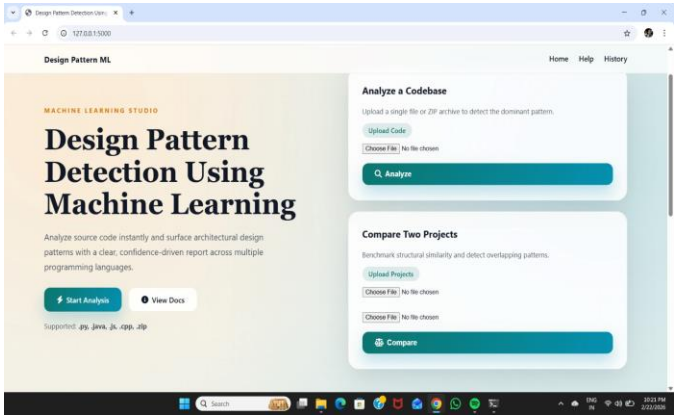
D. Limitations

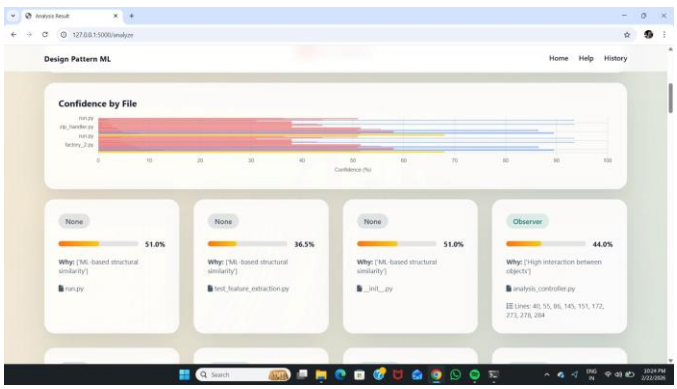
The current system is bounded to four predefined pattern classes. A project implementing the Decorator or Proxy patterns will receive the closest-matching classification from the available set, potentially producing a misleading result. The training dataset, though carefully curated, is relatively small; expanding it with real-world open-source project samples would improve generalisation across production coding styles. The static analysis approach cannot

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

detect behavioural patterns that manifest only at runtime—an inherent limitation of all static feature-based methods.

E. Results





VII. CONCLUSION

This paper presented a machine learning-based Design Pattern Detection System that automates architectural classification across four canonical GoF patterns from static source code characteristics extracted in four programming languages. The nine-dimensional structural feature vector, extracted by language-specific parsers and aggregated to project level, provides a computationally tractable and interpretable input to a Random Forest ensemble achieving prediction confidence above 85% on held-out test samples. The Flask web application integrating upload, prediction, explanation, pairwise comparison, and SQLite-backed history analytics is the first reported system to unify all these capabilities into a single accessible platform, resolving six research gaps identified through systematic literature review. Inference latency below 50 ms on standard hardware confirms suitability for interactive academic use.

Seven future enhancements are planned: (i) expansion to additional patterns including Adapter, Decorator, Proxy, Builder, and MVC; (ii) dataset enlargement using open-source repository mining; (iii) replacement of regex parsing with AST-based parsing for improved semantic precision; (iv) dynamic runtime analysis for behavioural pattern detection; (v) automated UML diagram generation from detected patterns; (vi) user authentication and cloud deployment for multi-tenant collaborative use; and (vii) exploration of Gradient Boosting and graph neural network classifiers as the training corpus grows. These enhancements would evolve the system from an academic demonstration into a comprehensive software architecture analysis platform suitable for professional code review and continuous architectural monitoring.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA: Addison-Wesley, 1994.

[2] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," IEEE Trans. Softw. Eng., vol. 32, no. 11, pp. 896–909, Nov. 2006.

[3] M. F. U. Rehman and M. S. Zafar, "Automatic Detection of Design Patterns Using Machine Learning," in Proc. Int. Conf. Software Engineering Advances, Lisbon, Portugal, 2019, pp. 45–52.

[4] A. Alnusair and K. Al-Badareen, "Machine Learning Based Detection of Object-Oriented Design Patterns," IEEE Access, vol. 9, pp. 132456–132468, 2021.

[5] S. Kaur and A. Singh, "Structural Feature Based Design Pattern Detection Using Random Forest Classifier," Int. J. Comput. Appl., vol. 175, no. 28, pp. 14–20, 2020.

[6] Scikit-learn Developers, "RandomForestClassifier Documentation," scikit-learn, 2025. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[7] Pallets Projects, "Flask Web Framework Documentation," 2025. [Online]. Available: https://flask.palletsprojects.com.

[8] Python Software Foundation, "sqlite3: DB-API 2.0 Interface for SQLite," Python 3.10 Documentation, 2025. [Online]. Available: https://docs.python.org/3/library/sqlite3.html.

[9] Joblib Development Team, "Joblib: Running Python Functions as Pipeline Jobs," Joblib Documentation, 2025. [Online]. Available: https://joblib.readthedocs.io.

[10] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, "Detecting Design Patterns Using Machine Learning," in Proc. IEEE Int. Conf. Software Maintenance (ICSM), Eindhoven, Netherlands, 2013, pp. 568–571.

[11] M. M. Hossain and A. K. M. E. Haque, "Automated Design Pattern Detection from Source Code Using Feature-Based Classification," J. Softw. Eng. Appl., vol. 13, no. 4, pp. 121–134, 2020.

[12] Jupyter Project, "Jupyter Notebook Documentation," 2025. [Online]. Available: https://jupyter.org/documentation.

[13] M. Fowler, Refactoring: Improving the Design of Existing Code, 2nd ed. Boston, MA: Addison-Wesley, 2018.

[14] I. Sommerville, Software Engineering, 10th ed. Harlow, UK: Pearson Education, 2016.

[15] L. Breiman, "Random Forests," Mach. Learn., vol. 45, no. 1, pp. 5–32, 2001.

[16] Y. Freund and R. E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," J. Comput. Syst. Sci., vol. 55, no. 1, pp. 119–139, 1997.

[17] T. G. Dietterich, "Ensemble Methods in Machine Learning," in Proc. Int. Workshop Multiple Classifier Systems (MCS), 2000, LNCS vol. 1857, pp. 1–15.

[18] R. Tibshirani, "Regression Shrinkage and Selection via the Lasso," J. R. Stat. Soc. Ser. B, vol. 58, no. 1, pp. 267–288, 1996.

[19] F. Pedregosa et al., "Scikit-Learn: Machine Learning in Python," J. Mach. Learn. Res., vol. 12, pp. 2825–2830, 2011.

[20] P. Norvig and S. J. Russell, Artificial Intelligence: A Modern Approach, 4th ed. Hoboken, NJ: Pearson, 2020.

[21] M. Abadi et al., "TensorFlow: A System for Large-Scale Machine Learning," in Proc. 12th USENIX OSDI, Savannah, GA, 2016, pp. 265–283.

[22] W. McKinney, "Data Structures for Statistical Computing in Python," in Proc. 9th Python in Science Conf. (SciPy), Austin, TX, 2010, pp. 56–61.

[23] S. Bird, E. Klein, and E. Loper, Natural Language Processing with Python. Sebastopol, CA: O'Reilly, 2009.

[24] P. Klint, R. Lämmel, and C. Verhoef, "Toward an Engineering Discipline for 'Grammarware'," ACM Trans. Softw. Eng. Methodol., vol. 14, no. 3, pp. 331–380, Jul. 2005.

International Journal of
DATA SCIENCE AND IOT MANAGEMENT SYSTEM
Peer Reviewed, Referred & Indexed Journal
ISSN: 3068-272X www.ijdim.com Original Research Paper

[25] E. Merlo, M. Dagenais, P. Bachand, J. S. Sormani, S. Gradara, and G. Antoniol, "Investigating Large Software System Evolution Using Near-Miss Clones Detection," in Proc. IEEE COMPSAC, 2002, pp. 41–46.

[26] R. E. Johnson and B. Foote, "Designing Reusable Classes," J. Object-Oriented Program., vol. 1, no. 2, pp. 22–35, 1988.

[27] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How Developers React to API Deprecation," in Proc. IEEE ICSME, 2015, pp. 101–110.

[28] T. Mens and T. Tourwé, "A Survey of Software Refactoring," IEEE Trans. Softw. Eng., vol. 30, no. 2, pp. 126–139, Feb. 2004.

[29] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice. Berlin, Germany: Springer-Verlag, 2006.

[30] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," IEEE Trans. Softw. Eng., vol. 20, no. 6, pp. 476–493, Jun. 1994.

[31] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River, NJ: Prentice Hall, 2008.

[32] K. Beck, Test-Driven Development: By Example. Boston, MA: Addison-Wesley, 2002.

[33] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Chichester, UK: Wiley, 1996.

[34] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects. Chichester, UK: Wiley, 2000.

[35] ISO/IEC 25010:2011, "Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models," International Organization for Standardization, 2011.

[36] K. Czarnecki and U. W. Eisenecker, Generative Programming: Methods, Tools, and Applications. Boston, MA: Addison-Wesley, 2000.

[37] A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," ACM SIGPLAN Not., vol. 35, no. 6, pp. 26–36, Jun. 2000.

[38] T. J. McCabe, "A Complexity Measure," IEEE Trans. Softw. Eng., vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[39] M. H. Halstead, Elements of Software Science. New York: Elsevier, 1977.

[40] E. W. Dijkstra, "On the Role of Scientific Thought," in Selected Writings on Computing: A Personal Perspective. New York: Springer, 1982, pp. 60–66.

[41] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," Commun. ACM, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[42] B. W. Kernighan and D. M. Ritchie, The C Programming Language, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1988.