AVE Trends in Intelligent Computing Systems

A Machine Learning Framework for Measuring Software Development Skill Evolution through Longitudinal Source Code Analysis

Prasanna Ranjith Christodoss[1], K. Srinarayani[2,*], Rahul Chauhan[3], M. Rehena Sulthana[4], Ramadas Kandath[5], Rahul Panakkal[6]

[1]Department of Computing, Mathematics and Physics, Messiah University, Mechanicsburg, Pennsylvania, United States of America.
[2]Department of Computer Science in AIML, SRM Institute of Science and Technology, Ramapuram, Chennai, Tamil Nadu, India.
[3]Department of Fitch Ratings, The Fitch Group, New York, United States of America.
[4]School of Information Technology and Engineering, Melbourne Institute of Technology, Melbourne, Victoria, Australia.
[5]Department of FSM Adoption and Services, SAP, Houston, Texas, United States of America.
[6]Department of Computer Science and Engineering, University of Illinois at Urbana, Champaign, Illinois, United States of America.
prchristodoss@messiah.edu[1], srinarak@srmist.edu.in[2], rahul.chauhan@thefitchgroup.com[3], rsulthana@academic.mit.edu.au[4], ramadas.kandath@sap.com[5], rahulp8@illinois.edu[6]

*Corresponding author

Abstract: Assessing software development proficiency remains a challenging task in both educational and industrial environments. Conventional evaluation approaches, including examinations, coding interviews, and manual code reviews, typically assess developers based on isolated code submissions and often fail to capture how programming skills evolve. Such snapshot-based assessments provide limited insight into learning trajectories, improvement patterns, and long-term coding behavior. This study proposes a machine learning-driven framework to measure software development skill evolution through longitudinal analysis of source code versions. The framework extracts a comprehensive set of software engineering metrics, including cyclomatic complexity, maintainability index, naming consistency, documentation quality, object-oriented design adherence, and error-handling effectiveness, from successive code submissions. These metrics are utilized to model developer proficiency using supervised machine learning classifiers, enabling categorization into Beginner, Intermediate, and Advanced skill levels. To capture temporal progression, two novel measures are introduced: the Growth Index, which quantifies the magnitude of improvement between historical and recent code versions, and the Drift Score, which evaluates the stability and consistency of that improvement over time. Experimental evaluation demonstrates that combining static code quality metrics with temporal progression indicators provides a more comprehensive assessment of developer capability than traditional single-submission approaches. The proposed framework offers an objective, scalable, and data-driven mechanism for tracking programming skill development and can support educational assessment, personalized learning systems, recruitment processes, and workforce development initiatives.

Keywords: Software Development Proficiency; Machine Learning; Source Code Analysis; Software Engineering Metrics; Developer Skill Evolution; Longitudinal Analysis; Growth Index; Drift Score; Proficiency Classification; Code Quality.

Cite as: P. R. Christodoss, K. Srinarayani, R. Chauhan, M. R. Sulthana, R. Kandath, and R. Panakkal, "A Machine Learning Framework for Measuring Software Development Skill Evolution Through Longitudinal Source Code Analysis," *AVE Trends in Intelligent Computing Systems*, vol. 3, no. 3, pp. 144–155, 2026.

Journal Homepage: https://www.avepubs.com/user/journals/details/ATICS

Received on: 21/06/2025, **Revised on:** 16/10/2025, **Accepted on:** 17/11/2025, **Published on:** 07/09/2026

Copyright © 2026 P. R. Christodoss *et al.,* licensed to AVE Trends Publishing Company. This is an open access article distributed under CC BY-NC-SA 4.0, which permits copying, remixing, redistributing, transformation, and building upon the material in any medium so long as the original work is properly cited.

DOI: https://doi.org/10.64091/ATICS.2026.000353

1. Introduction

The rapid growth of software development across academia, industry, and online learning platforms has increased the need for reliable methods to assess programming proficiency. Traditionally, software development skills have been evaluated through examinations, coding assignments, technical interviews, and manual code reviews. While these approaches provide a snapshot of a developer's current capabilities, they often fail to capture the dynamic nature of skill acquisition and improvement [1]. Programming expertise doesn't develop instantaneously; it emerges through continuous learning, repeated practice, code refinement, and exposure to increasingly complex development tasks. Consequently, assessing developers using isolated code submissions provides only a limited representation of their actual growth trajectory. The widespread adoption of version control systems and collaborative software development platforms has created unprecedented opportunities for studying how coding practices evolve. Developers routinely produce multiple versions of source code as they implement new features, fix defects, refactor existing components, and improve software quality [2]. These version histories contain valuable information regarding changes in coding style, structural design, maintainability, documentation practices, and problem-solving strategies. Despite the availability of such longitudinal data, most existing assessment approaches continue to evaluate software artifacts independently, without considering the progression of coding behavior across successive submissions. Previous studies have shown that software engineering metrics such as cyclomatic complexity, maintainability index, cohesion, coupling, and documentation quality can evaluate software quality and maintainability. Similarly, machine learning techniques have been successfully applied to tasks including defect prediction, code smell detection, software quality estimation, and developer profiling. However, existing research predominantly focuses on evaluating individual code artifacts rather than modeling the temporal evolution of developer skills [3].

As a result, important aspects of learning progression, consistency of improvement, and long-term development trends remain insufficiently explored. Another limitation of conventional assessment methods is their reliance on subjective evaluation. Manual code reviews often depend on reviewer expertise, personal judgment, and varying evaluation criteria, which may introduce inconsistencies across assessments [5]. Furthermore, large-scale educational environments and software organizations generate substantial volumes of source code, making continuous manual monitoring impractical. Automated and objective approaches are therefore necessary to ensure scalability, consistency, and reproducibility in software skill assessment. To address these challenges, this study proposes a machine learning-based framework to measure the evolution of software development skills by analyzing multiple versions of source code. The framework extracts a diverse set of software engineering metrics that characterize code complexity, maintainability, documentation practices, naming consistency, adherence to object-oriented design, and error-handling effectiveness [9]. These features are then used to train machine learning models that classify developers into predefined proficiency levels. Beyond proficiency classification, the proposed framework introduces two temporal indicators: the Growth Index (GI), which quantifies the magnitude of improvement observed across code versions, and the Drift Score (DS), which measures the consistency and stability of that improvement throughout the development timeline. The proposed approach contributes to the growing field of software analytics by shifting the focus from static code assessment to longitudinal developer evaluation. By integrating software engineering metrics, pattern recognition techniques, and machine learning models within a unified framework, the study enables objective measurement of programming skill progression while preserving scalability across large developer populations. The major contributions of this work are summarized as follows [12]:

- A longitudinal framework for assessing software development skill evolution using successive source code versions.
- A multi-metric feature extraction methodology incorporating complexity, maintainability, documentation quality, naming consistency, object-oriented design practices, and error-handling characteristics.
- A machine learning-based proficiency classification model for categorizing developers into beginner, intermediate, and advanced skill levels.
- The introduction of the Growth Index (GI) to quantify the magnitude of skill improvement across software development timelines.
- The introduction of the Drift Score (DS) to evaluate the consistency and stability of developer progression.
- A scalable and objective assessment methodology applicable to educational institutions, coding platforms, recruitment systems, and professional software development environments.

2. Literature Review

McCracken et al. [1] conducted a large-scale multi-national study to evaluate the programming skills of first-year computer science students across multiple institutions. Their research revealed significant variations in programming competency among students despite similar educational backgrounds. The study highlighted the challenges associated with objectively measuring programming proficiency and emphasized the need for standardized assessment frameworks. Their findings underscored the

need for systematic approaches to evaluating software development skills. Beaver and Schiavone [2] investigated the relationship between development team skill and software product quality. Their analysis showed that higher developer expertise directly contributes to improved software quality and lower defect rates. The study emphasized the importance of accurately measuring developer skill levels in software engineering environments. Their findings suggest that reliable skill assessment mechanisms can play a critical role in improving project outcomes. Rojpaisarnkit et al. [3] proposed a methodology for determining code proficiency levels using Python programming textbooks and code analysis techniques. The study employed machine learning methods to classify programming proficiency based on source code characteristics. Experimental results showed that code structure and quality indicators can effectively distinguish different levels of programming expertise. Their work demonstrated the feasibility of automatically classifying proficiency using source code analysis.

Nuñez-Varela et al. [4] presented a systematic mapping study on source code metrics and their applications in software engineering. The research reviewed numerous software metrics used for assessing software quality, maintainability, and complexity. The authors highlighted the widespread adoption of metrics such as cyclomatic complexity and maintainability index for evaluating source code characteristics. Their findings provide a strong foundation for metric-based software skill assessment frameworks. Kan [5] provided a comprehensive overview of software quality engineering metrics and predictive models used in software evaluation. The book discussed various quantitative measures for assessing software quality attributes, including reliability, maintainability, and complexity. The author emphasized the importance of objective software measurement in quality assurance processes. This work serves as a foundational reference for applying software metrics to analyze developer proficiency. Jay et al. [6] examined the relationship between Cyclomatic Complexity and Lines of Code through empirical analysis. Their findings revealed a stable correlation between software size and structural complexity across different software projects. The study demonstrated that complexity metrics can provide meaningful insights into software design and development practices. These findings support the use of complexity-based indicators in software skill evaluation systems. Reddy and Ojha [7] investigated feature selection techniques for predicting software maintainability using machine learning models. The study evaluated multiple maintainability prediction approaches and identified the software metrics most influential in predicting maintainability outcomes.

Experimental results demonstrated that appropriate feature selection significantly improves prediction accuracy. Their work highlights the importance of maintainability metrics in assessing software quality. Le et al. [8] presented a comprehensive survey of deep learning approaches for source code modeling and generation. The review examined various neural network architectures used for source code understanding, classification, and prediction tasks. The authors discussed the potential of machine learning techniques for extracting meaningful patterns from source code repositories. Their findings demonstrated the growing role of artificial intelligence in software engineering analytics. Folleco et al. [9] studied the effectiveness of Random Forest classifiers for software quality modeling under noisy data conditions. The research analyzed how imperfections in software datasets affect classification performance. Experimental findings showed that Random Forest models maintain strong predictive performance even with class noise. Their results support the suitability of Random Forest algorithms for software engineering prediction tasks. Malhotra and Chug [10] explored applying machine learning algorithms to software maintainability prediction. The study compared multiple classification techniques using software metrics as predictive features. The results showed that machine learning models can effectively estimate software maintainability characteristics. Their research reinforced the value of data-driven approaches for evaluating software quality.

Douce et al. [11] reviewed automatic assessment methods for programming education based on test-driven evaluation techniques. The study analyzed the strengths and limitations of automated programming assessment systems used in academic environments. The authors found that automated assessment significantly reduces evaluation effort while improving consistency. Their work highlighted the growing demand for objective, scalable methods for evaluating programming skills. Costa et al. [12] evaluated educational data mining techniques for predicting academic performance in introductory programming courses. The study applied machine learning algorithms to identify students at risk of poor performance early in learning. Experimental results demonstrated that learning-related features can effectively predict programming success. Their findings illustrated the potential of data-driven methods for monitoring programming skill development. Lu et al. [13] investigated the application of learning analytics in collaborative programming courses delivered through Massive Open Online Courses (MOOCs). The study analyzed student engagement patterns and their relationship with learning outcomes. Results showed that learning analytics can provide valuable insights into student progress and programming performance. Their work demonstrated the usefulness of longitudinal learning data for evaluating how programming skills evolve. Linstead et al. [14] proposed large-scale techniques for mining internet-scale software repositories.

The research explored methods for extracting knowledge from extensive collections of software artifacts and developer activities. Their findings demonstrated that repository mining can reveal meaningful patterns regarding software development behavior and expertise. This work established the potential of repository-based analytics for developer assessment. Yurdugül and Aşkar [15] conducted a longitudinal study examining the relationship between programming ability, problem-solving skills, and gender. The research tracked student development over an extended period to identify learning patterns and progression

trends. The results revealed that programming competence evolves gradually through continuous practice and experience. Their study emphasized the importance of longitudinal approaches when evaluating programming skill development. Conboy et al. [16] examined the role of temporal complexity in analytics and organizational decision-making. Their findings demonstrated that incorporating temporal dimensions into analytical models can improve understanding of evolving behaviors and performance trends. The study highlighted the significance of time-aware analytics in evaluating dynamic systems. These findings support incorporating measures of temporal progression into software skill assessment frameworks. Qureshi and Fang [17] proposed a growth mixture modeling approach to analyze socialization and participation patterns in open-source software projects. Their research showed that developer growth trajectories vary significantly across individuals and change over time. The authors emphasized the importance of modeling progression patterns rather than relying solely on static observations. Their findings provide theoretical support for measuring the evolution of software development skills through longitudinal analysis.

3. Methodology

3.1. Overview of the Proposed Framework

The proposed framework aims to measure the evolution of software development skills through longitudinal analysis of source code versions generated by individual developers. Unlike traditional assessment approaches that evaluate isolated code submissions, the proposed methodology examines multiple chronological versions of source code to identify patterns of improvement, consistency, and proficiency development.

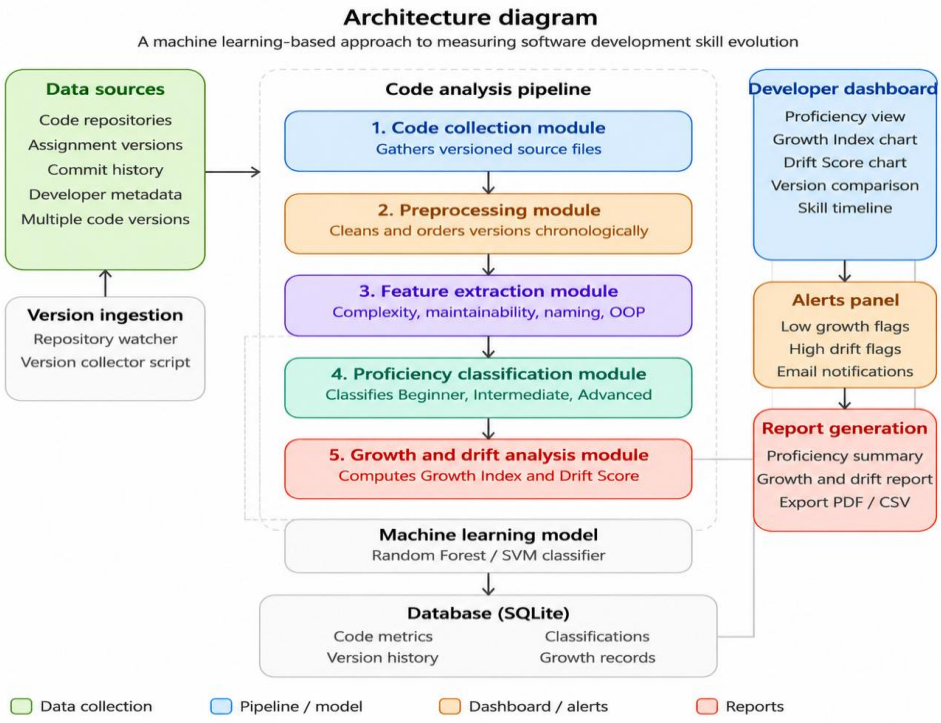


Figure 1: Proposed framework architecture for software development skill evolution analysis

The framework combines software engineering metric extraction, machine learning-based proficiency classification, and temporal progression analysis to generate a comprehensive evaluation of developer growth. The overall workflow consists of five major phases: (i) source code collection and preprocessing, (ii) feature extraction, (iii) proficiency classification, (iv) temporal growth analysis, and (v) performance reporting. Figure 1 illustrates the complete architecture of the proposed framework.

3.2. Source Code Collection and Preprocessing

The first stage involves collecting multiple source code versions for each developer. These versions may originate from academic programming assignments, coding platform submissions, software repositories, or version-controlled development environments. Each code version is linked to a developer identifier and a timestamp to preserve the temporal sequence of

submissions. Before analysis, the collected source code undergoes preprocessing to ensure consistency and reliability. Non-source files, duplicate submissions, incomplete programs, and corrupted code artifacts are removed. The remaining files are organized chronologically for each developer, enabling accurate analysis of skill progression over time. Standard preprocessing procedures also include code normalization, whitespace handling, and metadata validation.

3.3. Software Metric Extraction

To characterize software quality and programming practices, a set of software engineering metrics is extracted from each source code version. These metrics are selected because they capture multiple dimensions of code quality and developer competence.

Cyclomatic Complexity: Cyclomatic Complexity (CC) measures the structural complexity of a program by quantifying the number of independent execution paths within the source code. Lower complexity values generally indicate simpler and more maintainable code structures:

$$CC = E - N + 2P \quad (1)$$

Where:

- (E) Represents the Number of Edges in the Control Flow Graph,
- (N) Represents the Number of Nodes,
- (P) Denotes the Number of Connected Components.

3.4. Maintainability Index

The Maintainability Index (MI) evaluates how easily software can be modified and maintained. Higher MI values indicate better software maintainability:

$$MI = 171 - 5.2 \ln(V) - 0.23(CC) - 16.2 \ln (LOC) \quad (2)$$

Where:

- (V) is Halstead Volume,
- (CC) is Cyclomatic Complexity,
- (LOC) is Lines of Code.

Naming consistency: Naming consistency evaluates adherence to established naming conventions for variables, functions, classes, and methods. Consistent naming improves readability and software maintainability.

Documentation Quality: Documentation quality measures the presence and adequacy of comments, docstrings, and explanatory annotations within source code. Well-documented software facilitates maintenance and knowledge transfer.

Object-Oriented Design Adherence: This metric evaluates how well developers apply object-oriented programming principles, including encapsulation, inheritance, abstraction, and modular class design.

Error-Handling Quality: Error-handling quality assesses the robustness of exception management mechanisms, input validation procedures, and fault-tolerance strategies implemented in the source code. The extracted metrics collectively form a feature vector representing each code version.

Feature Normalization: Since the extracted metrics possess different numerical ranges and scales, feature normalization is performed before model training. Min-Max normalization is employed to transform all features into a common range between 0 and 1:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \quad (3)$$

Where:

- (X) is the Original Feature Value
- (X_{Min}) and (X_{Max}) denote the Minimum and Maximum Values Observed Within the Dataset.

Normalization prevents features with larger numerical ranges from dominating the learning process.

3.5. Machine Learning-Based Proficiency Classification

Following feature extraction and normalization, supervised machine learning algorithms are employed to classify developers into predefined proficiency categories:

- Beginner
- Intermediate
- Advanced

Random Forest and Support Vector Machine (SVM) classifiers are selected due to their proven effectiveness in software engineering prediction tasks. During training, the classification model learns relationships between software engineering metrics and proficiency labels. During inference, unseen code versions are assigned to the most probable proficiency category based on their extracted feature patterns.

3.6. Growth Index Computation

To quantify skill improvement over time, a Growth Index (GI) is introduced. The Growth Index measures the overall improvement observed between earlier and later code versions by considering changes in quality-related software metrics. Let (S_i) represent the aggregated quality score of version (i), and (S_f) represent the aggregated quality score of the final version. The Growth Index is computed as:

$$GI = \frac{S_f - S_i}{S_i} \quad (4)$$

Where higher values indicate greater improvement across the development timeline; the Growth Index enables quantitative measurement of developer progression beyond simple proficiency classification.

3.7. Drift Score Computation

While the Growth Index measures the magnitude of improvement, it does not capture the stability of that improvement. Therefore, a Drift Score (DS) is introduced to measure consistency throughout the development process. The Drift Score is computed as the standard deviation of consecutive growth values across multiple code versions:

$$DS = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(g_i-\bar{g})^2} \quad (5)$$

Where:

- (G_I) Represents the Growth Observed Between Consecutive Versions,
- (\Bar{G}) Denotes the Mean Growth Value,
- (N) is the Total Number of Growth Intervals.

Lower Drift Score values indicate stable and consistent improvement, whereas higher values suggest irregular or inconsistent development patterns.

3.8. Developer Skill Evolution Assessment

The final assessment combines proficiency classification, Growth Index, and Drift Score to generate a comprehensive representation of software development skill evolution. The classification model provides the current proficiency level, the Growth Index quantifies the magnitude of improvement, and the Drift Score evaluates progression stability. Together, these measures provide a multidimensional assessment framework that identifies not only current developer proficiency but also the trajectory and consistency of skill development over time.

4. Experimental Setup

4.1. Dataset preparation

To evaluate the proposed framework, researchers constructed a dataset of multiple source code versions generated by individual developers. Each developer contributed a sequence of code submissions representing different stages of their programming

development. The collected code samples were organized chronologically to preserve temporal relationships required for skill evolution analysis. Each source code version was subjected to static analysis to extract software engineering metrics. The extracted features included Cyclomatic Complexity, Maintainability Index, Naming Consistency Score, Documentation Quality Score, Object-Oriented Design Adherence Score, and Error-Handling Quality Score. These metrics collectively formed the feature space used for machine learning-based proficiency prediction and temporal progression analysis. Before model training, incomplete submissions, duplicate files, and invalid source code instances were removed to ensure data quality. Feature normalization was subsequently applied to eliminate scale differences among extracted metrics.

4.2. Experimental Environment

All experiments were conducted using Python-based machine learning and software analysis libraries. The implementation environment is shown in Table 1.

Table 1: Experimental environment and system configuration

Component	Specification
Programming Language	Python 3.11
Machine Learning Library	Scikit-Learn
Data Processing	Pandas, NumPy
Static Code Analysis	Radon, Pylint
Visualization	Matplotlib, Seaborn
Development Environment	Jupyter Notebook / VS Code
Operating System	Windows 11

The experiments were executed on a workstation equipped with an Intel Core i5 processor, 8 GB RAM, and SSD storage.

4.3. Feature Vector Construction

For each source code version, the extracted software engineering metrics were aggregated into a numerical feature vector:

$$F = [CC, MI, NC, DQ, OOP, EH] \quad (6)$$

Where:

- (CC) = Cyclomatic Complexity
- (MI) = Maintainability Index
- (NC) = Naming Consistency
- (DQ) = Documentation Quality
- (OOP) = Object-Oriented Design Adherence
- (EH) = Error-Handling Quality

The resulting feature vectors served as inputs to the machine learning classifiers.

4.4. Machine Learning Models

Two supervised machine learning algorithms were selected for proficiency classification:

- **Random Forest (RF):** It was selected because of its robustness against overfitting and its ability to model complex non-linear relationships among software metrics. The classifier constructs multiple decision trees and determines the final prediction through majority voting.
- **Support Vector Machine (SVM):** It was employed as a comparative baseline classifier. SVM identifies optimal decision boundaries that maximize separation among proficiency categories within the feature space. The comparison aimed to identify the most effective model for predicting developer proficiency.

4.5. Training and Validation Strategy

The dataset was divided into training and testing subsets using an 80:20 split. The training set was used to train the model, while the testing set was reserved exclusively for performance evaluation. To improve reliability and reduce sampling bias, k-

fold cross-validation was performed during training. The dataset was partitioned into k equal subsets, with each subset used once as validation data and the remaining subsets used for training. The final performance was computed as the average across all folds.

4.6. Evaluation Metrics

Model performance was evaluated using standard classification metrics widely adopted in machine learning research.

Accuracy: Accuracy measures the proportion of correctly classified instances among all evaluated samples:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (7)$$

Precision: Precision evaluates the proportion of correctly predicted positive instances among all positive predictions:

$$\text{Precision} = \frac{TP}{TP + FP} \quad (8)$$

Recall: Recall measures the proportion of actual positive instances correctly identified by the classifier:

$$\text{Recall} = \frac{TP}{TP + FN} \quad (9)$$

F1-Score: The F1-Score represents the harmonic mean of Precision and Recall:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (10)$$

Where TP, TN, FP, and FN denote True Positives, True Negatives, False Positives, and False Negatives, respectively.

4.7. Temporal Skill Evolution Evaluation

Beyond proficiency classification, the proposed framework evaluates software development skill progression using the Growth Index (GI) and Drift Score (DS). The Growth Index quantifies the magnitude of improvement between earlier and later code versions, while the Drift Score measures the consistency of improvement across the development timeline. Developers with high Growth Index values and low Drift Scores are interpreted as demonstrating steady, sustained skill development. Conversely, low Growth Index values or high Drift Scores indicate limited or inconsistent progression.

4.8. Comparative Evaluation

A comparative analysis was conducted between the evaluated machine learning models using the previously defined performance metrics. Classification accuracy, precision, recall, and F1-score were compared to determine the most effective classifier for software development proficiency prediction. In addition, researchers examined temporal analysis results from Growth Index and Drift Score computations to assess the proposed framework's ability to capture both the magnitude and stability of software skill evolution across multiple source code versions.

5. Results and Discussions

5.1. Classification Performance Analysis

Researchers evaluated the proposed framework's performance using Random Forest (RF) and Support Vector Machine (SVM) classifiers. Researchers trained both models using the extracted software engineering metrics and evaluated them on the test dataset using Accuracy, Precision, Recall, and F1-Score. Table 2 presents the overall classification performance of the evaluated models.

Table 2: Performance comparison of machine learning models

Model	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
Random Forest	93.4	92.8	93.1	92.9
SVM	87.6	86.9	87.2	86.8

The results indicate that the Random Forest classifier achieved the highest overall performance, attaining an accuracy of 93.4% and an F1-score of 92.9%. The superior performance can be attributed to its ensemble learning mechanism, which effectively

captures complex nonlinear relationships among software engineering metrics. In contrast, although SVM performed competitively, its accuracy was lower, suggesting it was less effective at modeling the heterogeneous characteristics of developer proficiency levels. The findings show that software engineering metrics contain sufficient discriminative information to distinguish between Beginner, Intermediate, and Advanced developers. Furthermore, the results confirm that machine learning models can successfully learn patterns associated with software quality and coding maturity.

5.2. Proficiency Level Classification Results

To further analyze classifier behavior, researchers evaluated proficiency prediction performance across individual developer categories (Table 3).

Table 3: Class-wise classification performance of random forest

Proficiency Level	Precision (%)	Recall (%)	F1-Score (%)
Beginner	94.2	95.1	94.6
Intermediate	91.8	90.7	91.2
Advanced	92.4	93.5	92.9

The Beginner category achieved the highest classification performance, mainly because novice coding patterns typically show stronger distinguishing characteristics, such as inconsistent naming conventions, limited documentation, and higher structural complexity. The Intermediate category produced slightly lower scores because it overlaps with improving beginners and emerging advanced developers. Nevertheless, all categories achieved F1-scores exceeding 90%, indicating balanced classification performance across proficiency levels.

5.3. Growth Index Evaluation

The Growth Index (GI) was introduced to quantify the magnitude of developer improvement across successive code versions.

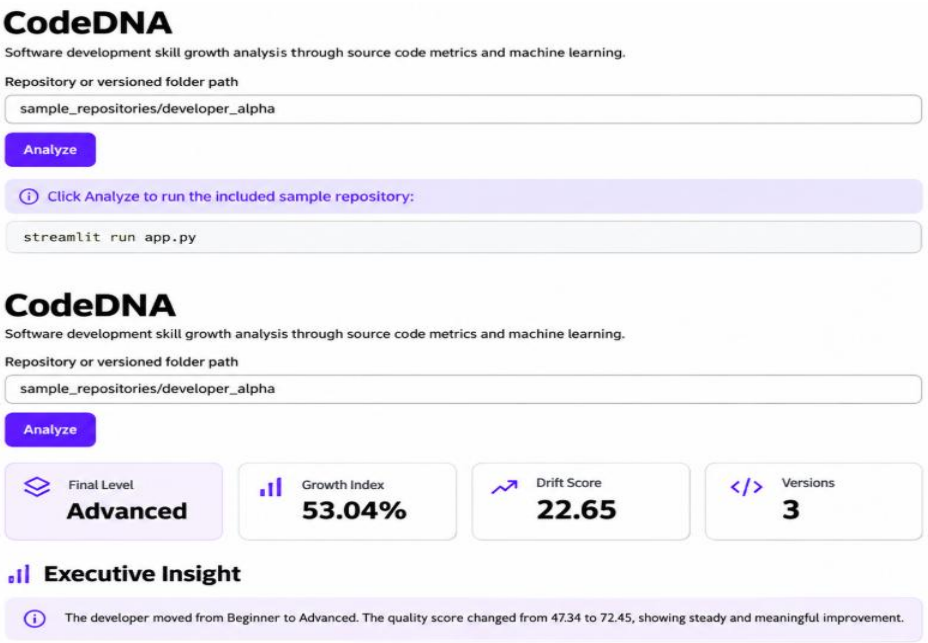


Figure 2: system interface showing developer proficiency classification, growth index, and drift score output (codedna tool)

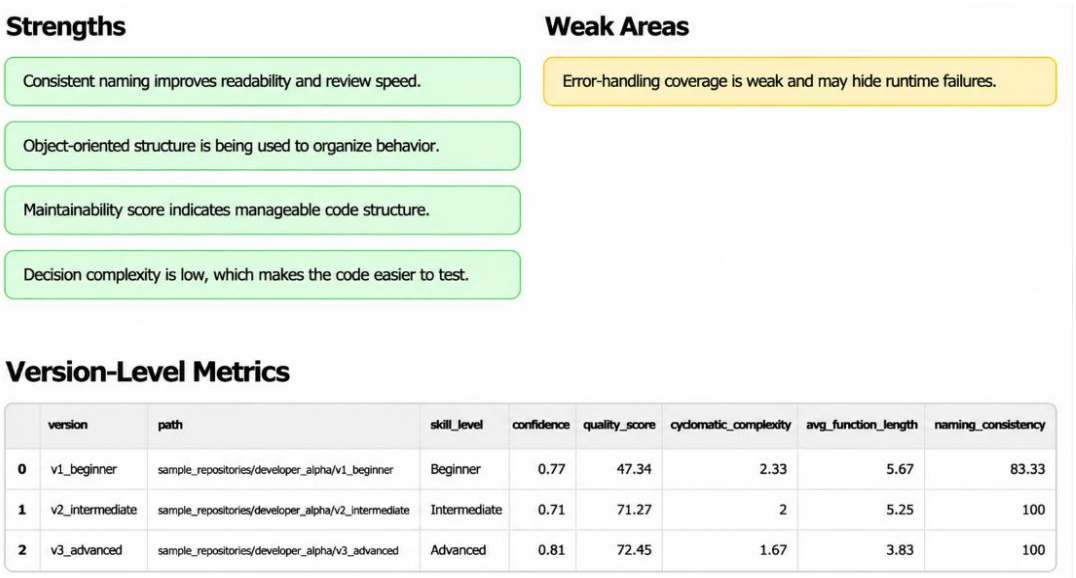
Analysis of developer trajectories revealed clear differences in progression patterns among participants. Developers who consistently improved maintainability, documentation quality, and error-handling practices achieved significantly higher Growth Index values. In contrast, developers whose coding patterns remained largely unchanged across versions produced comparatively lower scores. Figure 2 illustrates representative Growth Index trends for selected developers. The results show that the proposed Growth Index captures measurable changes in coding quality that conventional single-submission assessment methods would miss. The metric provides a quantitative measure of learning progression and helps identify developers who show sustained improvement throughout the evaluation period.

5.4. Drift Score Analysis

While the Growth Index measures the magnitude of improvement, it does not account for the consistency of progression. Therefore, the Drift Score (DS) was employed to evaluate stability across successive code versions. Developers with low Drift Scores showed gradual, consistent improvements in software engineering practices. Conversely, developers with high Drift Scores showed fluctuating performance, with periods of improvement followed by temporary regressions. The results indicate that Growth Index and Drift Score provide complementary information. For example, two developers may achieve similar Growth Index values while exhibiting substantially different Drift Scores. In such cases, one developer may show steady learning, whereas the other may improve irregularly through sporadic performance gains. This observation highlights the importance of incorporating temporal stability measures when evaluating how software development skills evolve.

5.5. Comparative Analysis with Traditional Assessment Approaches

Traditional software skill assessment methods generally evaluate a single code submission and therefore provide limited insight into long-term development patterns. In contrast, the proposed framework performs longitudinal analysis across multiple code versions and captures both current proficiency and developmental trajectory. Integrating software engineering metrics, machine learning classification, Growth Index computation, and Drift Score analysis enables a more comprehensive understanding of developer capability (Figure 3).



	version	path	skill_level	confidence	quality_score	cyclomatic_complexity	avg_function_length	naming_consistency
0	v1_beginner	sample_repositories/developer_alpha/v1_beginner	Beginner	0.77	47.34	2.33	5.67	83.33
1	v2_intermediate	sample_repositories/developer_alpha/v2_intermediate	Intermediate	0.71	71.27	2	5.25	100
2	v3_advanced	sample_repositories/developer_alpha/v3_advanced	Advanced	0.81	72.45	1.67	3.83	100

Figure 3: Version-level metrics table and quality trend chart showing skill progression from beginner to advanced level

The framework not only identifies a developer's current skill level but also determines whether they are improving, stagnating, or showing inconsistent progress. These capabilities are particularly valuable in educational environments, where instructors monitor student development over time, and in recruitment settings, where long-term growth potential may matter as much as current technical proficiency. The experimental findings demonstrate that software engineering metrics combined with machine learning techniques provide an effective mechanism for objective software skill assessment. Among the evaluated models, Random Forest achieved the highest classification performance because it can model complex interactions among multiple quality-related features. More importantly, the proposed Growth Index and Drift Score extend conventional proficiency assessment by incorporating temporal information into the evaluation process. Existing approaches predominantly focus on static code quality measurements, whereas the proposed framework captures both the magnitude and consistency of developer improvement. This enables a richer understanding of how software development skills evolve and provides actionable insights for educators, recruiters, and software organizations. Overall, the results validate the proposed framework as a scalable, data-driven solution for measuring software development skill evolution through longitudinal source code analysis.

6. Conclusion

This work presented a framework for assessing the evolution of software development skills using machine learning and longitudinal source code analysis. Unlike traditional assessment methods that evaluate developers based on separate coding tasks or submissions, the proposed framework analyzes successive code versions to provide a more holistic view of

programming proficiency and its evolution over time. This framework combines software engineering metrics such as cyclomatic complexity, maintainability index, naming consistency, documentation quality, adherence to object-oriented design, and error-handling efficiency, enabling a structured assessment of code quality and developer abilities. This objective, consistent assessment is further enhanced by classifying developers into Beginner, Intermediate and Advanced levels. We consider the introduction of the Growth Index and Drift Score an important contribution, as they measure skill improvement and consistency of progress across successive code submissions. The integration of static code quality metrics with temporal progression indicators provides a broader view of learning paths of developers, highlighting progress and variations in coding behaviour. This approach can benefit educational institutions, software companies, recruitment teams, and professional training programs by enabling ongoing skill evaluation, tailored instruction recommendations, and evidence-based competency assessment. To summarise, the framework shows how longitudinal source code analysis and machine learning can shift software development skill assessment from a snapshot-based process to a continuous, data-driven evaluation method. Future work can validate the framework on larger, more diverse developer datasets, explore advanced deep learning techniques, and study its applicability to different programming languages and software development environments. These improvements would further improve its accuracy, adaptability, and practical value for long-term programming skill development.

Acknowledgement: The authors gratefully acknowledge the academic support, institutional resources, and valuable contributions provided by Messiah University, SRM Institute of Science and Technology at Ramapuram, The Fitch Group, Melbourne Institute of Technology, SAP, and the University of Illinois at Urbana.

Data Availability Statement: The data supporting this study are available from the corresponding author upon reasonable request.

Funding Statement: This research and the preparation of the manuscript were carried out without any external funding or financial support.

Conflicts of Interest Statement: The authors declare that they have no conflicts of interest. This work constitutes an original contribution, and all sources and references have been appropriately cited and acknowledged.

Ethics and Consent Statement: The study was conducted in accordance with established ethical guidelines. Informed consent was obtained from all participants, and appropriate measures were taken to protect their privacy and maintain confidentiality.

References

1. M. O. McCracken, V. Almstrum, D. Diaz, M. J. Guzdial, D. Hagan, Y. B. D. Kolikant, C. E. Laxer, L. A. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," *in Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, New York, United States of America, 2001.
2. J. M. Beaver and G. A. Schiavone, "The effects of development team skill on software product quality," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–5, 2006.
3. R. Rojpaisarnkit, G. Robles, J. M. Gonzalez-Barahona, K. Matsumoto, and R. G. Kula, "Determining code proficiency levels from Python textbooks," *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 7, pp. 1–21, 2026.
4. A. S. Núñez-Varela, H. G. Pérez-González, F. E. Martínez-Pérez, and C. Soubervielle-Montalvo, "Source code metrics: A systematic mapping study," *Journal of Systems and Software*, vol. 128, no. 6, pp. 164–197, 2017.
5. S. H. Kan, "Metrics and Models in Software Quality Engineering," *2nd Ed*, *Addison-Wesley Professional*, Boston, Massachusetts, United States of America, 2003.
6. G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft, and C. Ward, "Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship," *Journal of Software Engineering and Applications*, vol. 2, no. 3, pp. 137–143, 2009.
7. B. R. Reddy and A. Ojha, "Performance of maintainability index prediction models: A feature selection-based study," *Evolving Systems*, vol. 10, no. 2, pp. 179–204, 2019.
8. T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Computing Surveys*, vol. 53, no. 3, pp. 1–38, 2020.
9. A. Folleco, T. M. Khoshgoftaar, J. Van Hulse, and L. Bullard, "Software quality modeling: The impact of class noise on the random forest classifier," *in IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, Hong Kong, China, 2008.
10. R. Malhotra and A. Chug, "Software maintainability prediction using machine learning algorithms," *Software Engineering: An International Journal (SeiJ)*, vol. 2, no. 2, pp. 19–36, 2012.

11. C. Douce, D. Livingstone, and J. Orwell, “Automatic test-based assessment of programming: A review,” *Journal on Educational Resources in Computing*, vol. 5, no. 3, pp. 1–13, 2005.
12. E. B. Costa, B. Fonseca, M. A. Santana, F. F. de Araújo, and J. Rego, “Evaluating the effectiveness of educational data mining techniques for early prediction of students' academic failure in introductory programming courses,” *Computers in Human Behavior*, vol. 73, no. 8, pp. 247–256, 2017.
13. O. H. T. Lu, J. C. H. Huang, A. Y. Q. Huang, and S. J. H. Yang, “Applying learning analytics for improving students engagement and learning outcomes in an MOOCs enabled collaborative programming course,” *Interactive Learning Environments*, vol. 25, no. 2, pp. 220–234, 2017.
14. E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, “Mining Internet-scale software repositories,” in *Proceedings of the 21st International Conference on Neural Information Processing Systems*, Vancouver, British Columbia, Canada, 2007.
15. H. Yurdugül and P. Aşkar, “Learning programming, problem solving and gender: A longitudinal study,” *Procedia - Social and Behavioral Sciences*, vol. 83, no. 7, pp. 605–610, 2013.
16. K. Conboy, D. Dennehy, and M. O'Connor, “‘Big time’: An examination of temporal complexity and business value in analytics,” *Information & Management*, vol. 57, no. 1, p. 103077, 2020.
17. I. Qureshi and Y. Fang, “Socialization in open-source software projects: A growth mixture modeling approach,” *Organizational Research Methods*, vol. 14, no. 1, pp. 208–238, 2011.

Publisher’s Note: *The publisher remains impartial concerning jurisdictional claims in published maps and institutional affiliations. Responsibility for the content rests entirely with the authors and does not necessarily reflect the publisher’s perspectives.*