INTERNATIONAL JOURNAL OF LATEST TECHNOLOGY IN ENGINEERING, MANAGEMENT & APPLIED SCIENCE (IJLTEMAS)

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

Modeling Transfer Learning for Efficient Code Reuse in Large-Scale Software Development

[1]paul Thomas Muge, Bassi Jeremiah Yusuf, [3] Jacob Lekchi Moltu

[1] Department of Electrical Electronic Engineering, Federal Polytechnic N'yak Shendam

[2] Department of Computer Engineering, Federal Polytechnic N'yak Shendam

[3] Department of Mechanical Engineering, Federal Polytechnic N'yak Shendam

DOI : https://doi.org/10.51583/IJLTEMAS.2026.150100008

Received: 05 January 2026; Accepted: 10 January 2026; Published: 22 January 2026

ABSTRACT

The important issues of code reuse in contemporary software development are covered in this study. Reusing code is an essential technique that raises software quality, lowers development costs, and increases productivity. However, because of problems like code repetition, a lack of knowledge of reusable components, maintenance difficulties, and scalability constraints, traditional approaches like libraries, APIs, and design patterns frequently fail in large-scale software development. Especially in big, distributed, and dynamic systems, these difficulties result in inefficiencies, longer development times, and lower software quality. In order to overcome these obstacles, this study suggests using transfer learning, a machine learning method that makes use of pre-trained models to enhance the recognition, modification, and incorporation of reusable code elements. The use of transfer learning in code reuse is a promising way to overcome the drawbacks of conventional approaches, and it has demonstrated notable success in domains such as computer vision and natural language processing. Through the use of models that have already been trained on sizable code corpora (such as open-source repositories like GitHub), transfer learning can help developers find and reuse high-quality code fragments across projects and programming languages more effectively, cutting down on duplication of efforts and enhancing code maintainability. The main aim of this study is to model how to use transfer learning for efficient code reuse in large-scale software development**.** Using extensive code datasets from private codebases or open-source repositories (like GitHub and GitLab), the study employed a quantitative methodology with a population size of 225. The findings show that CodeBERT is both robust and adaptable, offering high value for software engineering automation and developer assistance tools. The results demonstrate that the model whether trained from scratch or through transfer learning achieved perfect performance metrics in classifying and evaluating code snippets. This high accuracy indicates a strong capacity to enhance software development efficiency by enabling faster and more reliable code assessment. The equal performance of the transfer learning approach further shows that pretrained knowledge from large-scale open-source code can be effectively adapted to new tasks without compromising code quality or consistency. Overall, the findings confirm that the transfer learning model is not only stable and effective but also capable of delivering performance comparable to a fully trained model while requiring significantly less training data and computational resources. Use CodeBERT for automated code assessment, early bug detection, and identifying risky code patterns. Embed the model into Continuous Integration/Continuous Deployment (CI/CD) systems to enable automatic code review and error detection. Fine-tune with domain-specific datasets to maintain consistency with organizational coding standards. Conduct workshops to demonstrate efficiency gains from automated code evaluation.

Key words: Transfer learning model, Code representation, Software development efficiency, Code quality and consistency

INTRODUCTION

As a key technique in software engineering, code reuse seeks to decrease redundancy, boost efficiency, and improve software quality. Reusing existing code helps developers avoid "reinventing the wheel," which lowers

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

costs and speeds up development cycles (Frakes & Kang, 2005). Code reusability, according to Singh (2023), is the skill of developing software components that may be used in a variety of projects and situations. It is the epitome of efficiency, allowing developers to apply the knowledge gained from previous undertakings to new problems. Encapsulating functionality within reusable modules streamlines the development process, lowers costs, and improves software quality overall. Code reuse is even more important in large-scale software projects, when size and complexity present major obstacles.

However, given the dynamic and changing nature of contemporary software systems, conventional approaches to code reuse such as libraries, APIs, and design patterns frequently prove inadequate, as expressed by Bass et al., (2013). Transfer learning, according to Murel and Kavlakoglu (2024), is a machine learning technique that uses knowledge from one task or dataset to enhance model performance on a different dataset or related task. To put it another way, transfer learning enhances generalization in a different context by applying knowledge acquired in one context. There are numerous uses for transfer learning, ranging from deep learning model training to data science regression problem solving. Indeed, considering the vast quantity of data required to build deep neural networks, it is especially alluring for the latter.

One of the most promising strategies to combat the ongoing software crisis is software reuse. With advantages including shortened development times, better software quality, and cheaper expenses, it is a fundamental component of software engineering. Code reuse in large-scale software development, however, poses serious difficulties that may reduce its efficacy. However, there are a lot of issues with large-scale software development. Among them are

(i) code duplication and redundancy: Code redundancy frequently arises in large-scale projects when developers purposefully duplicate code to meet deadlines or are ignorant of existing reusable components. Inconsistencies throughout the codebase and higher maintenance costs result from this. According to research by Kapser and Godfrey (2008), for instance, up to 50% of the code in some systems is duplicated, which can cause problems with maintenance and even defects. This causes the codebase to grow in size due to redundant code, which makes it more difficult to manage and raises the possibility of mistakes occurring during updates.

(ii) Insufficient Knowledge of Reusable Parts: Underutilization of available resources may result from developers in dispersed environments or large teams being unaware of existing reusable components. Because of this, developers frequently "reinvent the wheel" due to their ignorance of pre-existing libraries or modules, as noted by Mili et al. (2002). This leads to missed opportunities to use well-tested and optimized code, wasted work, and longer development times.

(iii) Difficulties with Maintenance: Reused code maintenance gets more difficult as software systems develop. Reused components may unintentionally be impacted by changes made to one area of the system, resulting in errors and inconsistencies. For example, Parnas (1979) highlighted that badly designed reusable components can lower the long-term benefits of code reuse and increase the total cost of ownership by becoming a maintenance burden, particularly in big systems.

(iv) Problems with Scalability: Libraries and APIs are examples of traditional code reuse strategies that frequently don't scale well in expansive and dispersed contexts. Bass et al. (2013) illustrated this specific situation by pointing out that the complexity of managing reusable components grows exponentially with system size. Scalability problems restrict the efficacy of code reuse in large-scale projects, resulting in inefficiencies and decreased productivity. This has a noticeable impact.

Thus, it is evident that code reuse issues in large-scale software development are complex, involving organizational, cultural, and technical issues. Better tools, standardized procedures, and organizational culture change are all necessary to address these problems. Transfer learning is one of the emerging technologies that offers promising answers by facilitating automated and intelligent ways to code reuse.

Decision-makers in the software business are greatly impacted by the difficulties with code reuse in large-scale software development. It is imperative that executives deliberately address these issues since they have a direct impact on cost, efficiency, and competitiveness. Addressing these issues is crucial for long-term commercial

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

success for key players in this sector, not merely for enhancing technical procedures. Significant efficiency, cost savings, and increased competitiveness in the software sector can be achieved by investing in cutting-edge technologies like transfer learning, standardization, and a reuse culture.

To address the difficulties of code reuse in large-scale software development, researchers have looked into a number of strategies. These initiatives include methodological advancements, organizational tactics, and technical developments. Tools to automatically identify and suggest reusable code components have been created by several academics. These technologies make use of methods like machine learning, clone identification, and code search. Clone detection algorithms, for instance, are used by programs like Deckard (Jiang et al., 2007) and NiCad (Roy et al., 2009) to find duplicate code, allowing programmers to more efficiently restructure and reuse code. These tools increase the precision of identifying reusable components while lowering manual labor. Once more, codebases have been analyzed using machine learning (ML) models to suggest reusable parts.

To comprehend the context and semantics of code, methods like deep learning and natural language processing (NLP) are employed. In order to illustrate this, Feng et al. (2020) and Guo et al. (2021) developed CodeBERT and GraphCodeBERT, respectively, using pre-trained models that can be optimized for tasks such as code search and reuse and learn code representations. Additionally, for tasks like code completion, summarization, and reuse, researchers like Chen et al. (2021) and Wang et al. (2021) improved models like Codex and CodeT5, respectively. This is the effect of using transfer learning to leverage pre-trained models on large code base. This allows for effective adaptation to particular reuse tasks, which lowers the requirement for a lot of training data and computational resources, making large-scale applications possible.

By using pre-trained models to enhance the recognition, adaptation, and integration of reusable code components, transfer learning presents a possible answer to the problems associated with code reuse in large-scale software development. Scalability, compatibility, and the requirement for large amounts of training data are common issues with traditional techniques to code reuse. By using models that have already been trained on sizable code bases like open-source repositories, transfer learning tackles these problems by capturing the general patterns and semantics of code across a variety of programming languages and fields. Developers can swiftly find pertinent and superior reusable components by fine-tuning models like CodeBERT and CodeT5 for particular tasks like code search, summary, or reuse recommendation. The time and effort needed to find reusable code can be decreased, for instance, by using a transfer learning model to evaluate a developer's code context and suggest functionally equivalent code snippets from a large repository. Furthermore, by comprehending the underlying semantics and producing the required changes, transfer learning can resolve compatibility concerns and adapt reused code to new situations. Transfer learning enables sophisticated code reuse techniques for large-scale projects by lowering the requirement for substantial training data and computer resources. In addition to increasing output and software quality, this strategy promotes a culture of clever and effective code reuse, which eventually lowers costs and promotes innovation in software development.

PROBLEM STATEMENT/JUSTIFICATION

A key technique in software engineering is code reuse, which makes use of pre-existing code components to increase software quality, lower development costs, and increase productivity. However, a number of obstacles prevent code reuse from being implemented effectively in large-scale software development. Code redundancy and duplication, ignorance of reusable components, compatibility and integration problems, maintenance difficulties, and scalability constraints are some of these difficulties (Frakes & Kang, 2005; Kapser & Godfrey, 2008). These problems are frequently not addressed by traditional code reuse techniques like libraries, APIs, and design patterns, especially in large, distributed, and dynamic systems (Bass et al., 2013; Mili et al., 2002). Additionally, the lack of defined procedures and intelligent tools makes the issue worse by resulting in inefficiencies, longer development times, and lower-quality software (Allamanis et al., 2018). Although machine learning (ML) has demonstrated potential in automating code analysis and reuse, large-scale projects frequently lack the computing resources and labeled data necessary to train ML models from scratch (Hindle et al., 2016). Innovative methods that may effectively find, modify, and suggest reusable code components while reducing computational expenses and manual labor are required to address these issues. By utilizing knowledge from

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

extensive code corpora to enhance code reuse in novel situations, transfer learning, a technique that involves fine-tuning pre-trained models for particular tasks offers a possible remedy (Pan & Yang, 2010; Feng et al., 2020). Nevertheless, there is still a dearth of thorough frameworks and empirical research to support the efficacy of transfer learning's application to code reuse in large-scale software development. By examining how transfer learning can be applied to address the difficulties associated with code reuse in large-scale software development, this study aims to close this gap and enhance software quality, scalability, and efficiency.

OBJECTIVE(s) OF THE STUDY

The aim of this study is to model how to use transfer learning for efficient code reuse in large-scale software development

Specific objectives include to:

a) Develop a transfer learning model for code representation

b) Measure the impact of the model on software development efficiency

c) Analyze how transfer learning contributes to code quality and consistency

d) Evaluate the effectiveness of the transfer learning model

LITERATURE REVIEW

Pandey (2024) stated that the ability to reuse existing code for creating new software applications is known as code reusability. Any stable, working code could be freely reused when creating a new software program, and code reuse should ideally be simple to implement. Regretfully, this is not the case; it is crucial to make sure that the code being reused is suitable and fits well with the software program. Code that already exists can be reused to accomplish the same goal or modified to accomplish a somewhat different but comparable goal. Reusable code boosts output, lowers expenses, and enhances quality in general. In software development, reusability is a very common and effective technique. Reusable code should have the following qualities: compatibility with various hardware, adaptability that makes it easy to modify for another application, and the absence of any errors or flaws that could compromise the security or reliability of the other application. As a result, this study demonstrates the benefits of code reuse, which is the study's motivation. However, the study under consideration goes one step further and recommends the effective application of machine learning and transfer learning techniques to improve software reuse.

Deepika and Sangwan (2021) believed that reusability is crucial to producing high-quality, error-free, and less maintenance software, but that software engineering is an application of engineering that is more focused on original development. Reusable software is a quality feature that aids in choosing previously learned concepts for new situations. In addition to increasing productivity, software reusability offers high-quality software and has a positive impact on maintainability. The advantages of software reusability include reduced development time, minimal maintenance costs, and excellent reliability. The study examined and evaluated a number of machine learning methods for estimating software reusability. The findings demonstrated that machine learning methods may be employed for reusability estimation and are competitive with existing reusability estimation methods. Software developers and the information business may find the study useful in clarifying how software reusability might help them choose high-quality software. While the latter study uses transfer learning, another machine learning approach, to achieve the efficient application of reused codes during software development, the previous study concentrated on quantifying the codes that can be reused using machine learning techniques.

Maggo and Gupta (2014) stated that the creation of new software systems with the possibility of fully or partially utilizing pre-existing resources or components, with or without modification, is known as software reuse. The ease with which previously learned ideas and items can be applied in other situations is known as reusability. It offers cost-effective, dependable (given that previous testing and use have removed defects), and faster (shortened time to market) software product development, making it a potential approach for increasing software

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

quality, productivity, and maintainability. In order to assess the degree of reusability (high, medium, or low) of procedure-oriented Java-based (object-oriented) software systems, they provided an effective automation model in the paper for the identification and assessment of reusable software components. The proposed model targeted key reusability analysis aspects using a metric framework for object-oriented software components, while also accounting for partial reuse by using the maintainability index. To determine connections between the functional qualities, the LMNN machine learning algorithm was investigated further. Instead of operating at the structural level, the model operated at the functional level. The system was constructed as a Java tool, and metrics such as precision, recall, accuracy, and error rate were used to record the automation tool's performance. According to the results obtained, the model can be used to identify procedure-based reusable components from the current inventory of software resources in an efficient, accurate, quick, and cost-effective manner. While the latter study uses machine learning techniques to effectively apply software reuse in software development, the former study showcased the beauty of software reuse by experimenting with object-oriented software and assessing performance using metrics like precision, recall, accuracy, and error rate.

Tan (1999) stated that efficient collaboration is not supported by the infrastructure for software development that is in place at the moment. Because scattered teams face obstacles due to inadequate technology, the virtual team concept has not been completely implemented. Every new cycle usually starts with little knowledge and experience from earlier cycles, which results in significant time and financial expenses. Furthermore, the development infrastructure restricts communication between functional groups by isolating phases from one another. In an engineering project, it is necessary to reapply prior knowledge and expertise in order to fully reap the benefits of expanded cooperation and avoid the costs of limited collaboration. The study looked at the products produced in a single development cycle and reusability in the software development process. It was highlighted how reuse can facilitate geographically dispersed and temporally separated collaboration in a development setting. One such example of a remote multi-year cooperation that aggressively advocates for integrating reuse techniques in the development cycle is that is collaboration through software reuse is evident in this effort. It is evident that improving software reuse in a collaborative setting was another goal of this study. In contrast to the former, the study under consideration takes into account the application of machine learning techniques to improve software reuse.

Mastropaolo, et al. (2022) claimed that a number of code-related tasks, including code summarizing and bug-fixing, have been supported by deep learning (DL) techniques. Specifically, pre-trained transformer models are becoming more popular, partly because of their outstanding performance on Natural Language Processing (NLP) tasks. Essentially, these models are pre-trained on a generic dataset using a self-supervised task (filling masked words in sentences, for example). These models are then adjusted to serve certain tasks of interest, like translating languages. It may be possible to take advantage of transfer learning by optimizing a single model to handle several activities. This implies that skills learned to complete one task (like translating languages) can help improve performance on another one (like classifying sentiment). Although the advantages of transfer learning have been extensively researched in NLP, there is little empirical data about tasks involving code. The report evaluated the Text-To-Text Transfer Transformer (T5) model's ability to serve four distinct code-related tasks, which are: (i) code summarization; (ii) injecting code mutants; (iii) generating assert statements; and (iv) automatically repairing bugs. They specifically looked into how pre-training and multi-task fine-tuning affected the model's performance. The study's findings demonstrated that (i) the T5 can outperform state-of-the-art baselines; and (ii) while pre-training aids the model, multi-task fine-tuning is not beneficial for all activities. This study, which does not specifically address code reuse, highlights the advantages of adopting machine learning techniques in certain code-related tasks. By taking into account how transfer learning is used to accomplish code reuse, the study in question also shows how machine learning may improve this conventional and efficient code development approach, which sets it apart from the former.

Kanade (2022) stated that a machine learning (ML) technique known as "transfer learning" employs a trained model created for one job to complete another that is related but different. As a result, the second model, which concentrates on the new task, incorporates the knowledge gained from task one. According to him, transfer learning is taking the knowledge gained from one activity and using it to improve another. In theory, an ML model transfers the weights it arrests while solving "problem X" to a new "problem Y." The goal is to finish task_2, which has less data or labels than task_1, by reprocessing the knowledge acquired from task_1, which

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

contains labeled training data. Instead of starting from scratch, transfer learning allows the learning process to start with patterns found when completing related activities. Therefore, the study under discussion aims to implement transfer learning in code reuse during software development, while this study just clarified what transfer learning comprises.

Malhotra and Meena (2024) revealed that everyone needs high-quality software these days and that the inability to obtain data for testing and training makes it impossible to guarantee software quality. The reusability of current software for creating new software with a similar domain and goal is thus greatly aided by transfer learning (TL). In order to create new prediction models, TL concentrated on applying knowledge from preexisting models. Depending on the features and nature of the dataset, the produced models are applied to unseen datasets. There is not enough training data available. Before using TL for software development, the source and target projects' tasks and data distribution must be examined. They examined 39 studies that used TL in the field of software engineering between January 1990 and March 2024 for their systematic review (SR). Identification of Machine Learning (ML) techniques used with TL techniques, types of TL explored, TL settings explored, experimental setting, dataset, quality attribute, validation methods, threats to validity, TL techniques' strengths and weaknesses, and hybrid techniques with TL were the main topics of the review. The experimental comparison showed that the TL approaches performed well. Scholars, professionals in the software industry, developers, testers, and researchers can all use the study's conclusions as a guide. According to the kind of problem and TL setting, the work also helped choose suitable TL kinds and TL settings for the future development of effective software. Accordingly, 30.67% of the research, which used 15% open-source data, was devoted to fault prediction. Additionally, SVM was utilized as the basis classifier for TL in 35% of the trials, and the prediction model input consisted of several independent variables from the dataset. Additionally, the 15 investigations employed the K-fold cross-validation (CV) approach. Evidently, the latter study is focused on the use of transfer learning for software reuse in software engineering, while the earlier study reviewed works that investigated the application of transfer learning in software engineering.

Zhang et al. (2022) opined that developers can create unique models (students) based on complex pretrained models (teachers) using transfer learning, a common software reuse strategy in the deep learning field. However, some flaws in the teaching model, such as Cowell-known adversarial vulnerabilities and backdoors, may also be passed down to pupils, similar to vulnerability inheritance in traditional software reuse. Since the learner is ignorant of the teacher's training and/or attacks, it is difficult to reduce such flaws. In order to minimize flaw inheritance during transfer learning while preserving valuable information from the teacher model, they suggested ReMoS, a pertinent model slicing technique, in the paper. In particular, ReMoS uses the neuron coverage information gathered by profiling the teacher model on the student task to calculate a model slice (a subset of model weights) that is pertinent to the student task. To reduce the possibility of inheriting flaws, the irrelevant weights are retrained from scratch, and only the pertinent slice is utilized to refine the student model. They conducted experiments on eight datasets, four DNN models, and seven DNN flaws. The results showed that ReMoS can successfully minimize inherited defects with little loss of accuracy (average of 3%), reducing them by 63% to 86% for CV tasks and 40% to 61% for NLP tasks. Comparatively, the study by Zhang *et al*., (2022) applied transfer learning to solve some DNN and NLP tasks, while the research under consideration seeks to clearly demonstrate and educate how transfer learning can be applied in achieving efficient software development in large scale software.

Mastropaolo et al. (2022) assessed the basic idea behind these models is to first pre-train them on a generic dataset using a self-supervised task (filling masked words in sentences). Then, these models are fine-tuned to support specific tasks of interest (language translation). A single model can be fine-tuned to support multiple tasks, possibly exploiting the benefits of transfer learning. This means that knowledge acquired to solve a specific task (language translation) can be useful to boost performance on another task (sentiment classification). While the benefits of transfer learning have been widely studied in NLP, limited empirical evidence is available when it comes to code-related tasks. The paper assessed the performance of the Text-To-Text Transfer Transformer (T5) model in supporting four different code-related tasks: automatic bug-fixing, injection of code mutants, generation of assert statements, and code summarization. The study pays particular attention in studying the role played by pre-training and multi-task fine-tuning on the model's performance. The study show that the T5 can

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

achieve better performance as compared to state-of-the-art baselines; and while pre-training helps the model, not all tasks benefit from a multi-task fine-tuning.

Lima et al. (2025) explored transfer learning for multilingual software quality: code smells, bugs, and harmful code. The study as an extension of the previous study, with the scope of 5 smell types, The total number of commits across all four tables (Java, C++, C#, and Python projects) is 641,736 bugs and 24,737 code smells. The findings revealed promising transferability of knowledge between Java and C# in the presence of various code smell types, while C++ and Python exhibited more challenging transferability. Also, the study discovered that a sample size of 32 demonstrated favorable outcomes for most harmful codes, underscoring the efficiency of transfer learning even with limited data. Moreover, the exploration of transfer learning between bugs and code smells represents a not-very-ineffective avenue within the realm of software engineering.

Kalouptsoglou et al. (2025) investigated the capacity of Generative Pre-trained Transformer (GPT), and Bidirectional Encoder Representations from Transformers (BERT) to enhance the VP process by capturing semantic and syntactic information in the source code. Specifically, we examine different ways of using CodeGPT and CodeBERT to build VP models to maximize the benefit of their use for the downstream task of VP. To enhance the performance of the models we explore fine-tuning, word embedding, and sentence embedding extraction methods. We also compare VP models based on Transformers trained on code from scratch or after natural language pre-training. Furthermore, we compare these architectures to state-of-the-art text mining and graph-based approaches. The results showcase that training a separate deep learning predictor with pre-trained word embeddings is a more efficient approach in VP than either fine-tuning or extracting sentence-level features. The findings also highlight the importance of context-aware embeddings in the models' attempt to identify vulnerable patterns in the source code.

Tonyloi (2023) explored the effectiveness of transfer learning in deep neural networks for image classification. The use of pre-trained models can save significant time and computational resources, as well as improve the accuracy of the model. The study compared the performance of transfer learning with traditional image classification techniques, such as training from scratch and fine-tuning. The study was conducted using publicly available datasets for image classification. The models will be implemented using deep learning frameworks such as TensorFlow and Keras. The comparison of the performance of transfer learning was based on metrics such as accuracy, loss, training time, and inference time. The results of study provide insight into the effectiveness of transfer learning in deep neural networks for image classification. The study revealed that transfer learning outperform traditional techniques in terms of accuracy and training time, while providing similar or better results in terms of inference time.

METHODOLOGY

This study adopted mixed methodology approach. Software engineers from software development companies will be sampled to give an overview of their understanding on the efficiency of this strategy in software development. Also, datasets of large-scale code from open-source repositories (GitHub, GitLab) or proprietary codebases will be accessed and made use of. It will be preprocessed by cleaning, tokenizing, and converting code into suitable representations (e.g., abstract syntax trees, embeddings). Thereafter, it will be labeled for supervised training. A GPT-based model will be used for the transfer learning. The implementation was done using some machine learning libraries like Pytorch, in Python programming language. The model was evaluated using metrics such as accuracy, precision, recall, F1-score, and mean reciprocal rank (MRR) to check its performance.

Data collection

The main method of data gathering in this research was based on collection of primary and secondary data.

a. Primary Data

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

Software developers were sampled to get an insight into their perception about the use of this strategy in software development.

b. Secondary Data

Existing datasets of large-scale codes from open-source repositories (GitHub, GitLab) or proprietary codebases was accessed and made use of for training the model

c. Data Analysis

Python programming language with some relevant libraries used for data training, testing and analysis

RESULT AND DISCUSSION

Table 1: Demographic Profile of Respondents

Demographic Variable	Category	Frequency	Percentage (%)
Gender	Male	158	70.2
	Female	67	29.8
Age	20–29 years	91	40.4
	30–39 years	87	38.7
	40 years and above	47	20.9
Educational Qualification	Bachelor's Degree	132	58.7
	Master's Degree	63	28.0
	Others (Diploma, Cert.)	30	13.3
Years of Experience	Less than 3 years	56	24.9
	3–6 years	98	43.6
	Above 6 years	71	31.6
Organization Type	Private Tech Firm	142	63.1
	Freelance/Independent	58	25.8
	Public/Corporate IT Unit	25	11.1
Total		**225**	**100**

Field Survey, (2025)

Table 1 shows that most respondents (70.2%) were male, reflecting the gender imbalance common in the tech industry. The largest age group was between 20–29 years (40.4%), indicating a young and dynamic developer population. Educationally, the majority (58.7%) held a Bachelor's degree, suggesting strong formal qualifications among participants. In terms of experience, 43.6% had between 3–6 years of work experience,

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

showing a moderately experienced workforce. Finally, most respondents (63.1%) were employed in private tech a firm, highlighting that software development activity in the study area is primarily driven by the private sector.

Table 2: Respondents and Mean and Standard Deviation

Variable	Mean	SD
Usefulness	3.66	±0.66
Ease of Use	3.36	±0.71
Code Quality Impact	3.50	±0.69
Productivity	3.56	±0.77
Collaboration	3.37	±0.68
Barrier (lower = fewer)	2.24	±0.87
Perception_Score (composite)	3.49	±0.57

The developer in Table 2 reports moderately positive perceptions across the five positive metrics (means ≈ 3.3–3.7 on a 1–5 scale). Average reported barriers are moderate-to-low (mean ≈ 2.24). The composite Perception_Score is ≈ 3.49 ± 0.57, indicating modestly favorable overall sentiment.

Table 3: Cross-tabulation by Role (counts and mean scores)

Role	Count	Perception_ Score (mean)	Usefulness	Ease	Code_Quality	Productivity	Collaboration	Barrier
Junior Developer	~63	3.65	3.90	3.90	3.50	3.80	3.50	2.20
Mid-level Developer	~56	3.05	3.20	3.10	3.30	3.40	3.00	2.60
Senior Developer	~34	3.42	3.60	3.30	3.60	3.60	3.30	2.30
Lead Engineer	~27	4.30	4.60	3.80	4.50	4.50	4.00	1.60
Full-stack Developer	~27	3.95	4.00	3.90	4.10	4.00	4.00	2.00

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

DevOps Engineer	~18	2.90	3.10	3.00	3.20	3.10	3.10	2.30

In Table 3, lead engineers report the highest overall perception (≈ 4.30), indicating they see the greatest usefulness, productivity and code-quality gains and face the fewest barriers. Full-stack and junior developers also show relatively positive perceptions (≈ 3.95 and 3.65 respectively). Mid-level developers and DevOps staff show the lowest mean perception scores (≈ 3.05 and 2.90), suggesting these groups will experience more friction, unmet needs, or lower perceived impact.

Table 4: Cross-tabulation by Company Size (counts and means)

Company Size	Count	Perception Score (mean)	Usefulness	Ease	Code_Quality	Productivity	Collaboration	Barrier
Large (251+)	~67	3.55	3.70	3.45	3.60	3.60	3.50	2.15
SME (51–250)	~79	3.47	3.65	3.35	3.50	3.55	3.35	2.25
Startup (1–50)	~79	3.50	3.63	3.28	3.40	3.53	3.35	2.30

Table 4 showed differences across company sizes are small. Large organizations show a slight edge in perception (≈ 3.55) relative to SMEs and startups (≈ 3.47–3.50). This implies perceived usefulness/impact is similar across organization sizes, with only marginal advantage in larger firms.

Table 5: Cross-tabulation by Experience Group (Counts & Means)

Experience Group (years)	Count	Perception_Score (mean)	Usefulness	Ease	Code_Quality	Productivity	Collaboration	Barrier
0–3	~48	3.70	3.85	3.95	3.60	3.80	3.70	2.10
4–7	~58	3.40	3.55	3.25	3.40	3.45	3.25	2.30
8–12	~62	3.45	3.60	3.30	3.50	3.55	3.35	2.25
13+	~57	3.60	3.75	3.45	3.65	3.70	3.50	2.00

Table 5 showed that early-career (0–3 yrs) and very experienced (13+ yrs) developers show somewhat higher perceptions (≈ 3.70 & 3.60), whereas mid-career groups (4–12 yrs) are slightly lower (≈ 3.40–3.45). This pattern may reflect early enthusiasm and late-career strategic acceptance versus mid-career friction or critical scrutiny.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

Table 6: Correlation matrix (Pearson r — main variables)

	Usefulness	Ease	Code_ Quality	Productivity	Collaboration	Barrier	Perception Score
Usefulness	1.000	0.550	0.620	0.680	0.580	-0.420	0.930
Ease	0.550	1.000	0.520	0.590	0.510	-0.360	0.840
Code_Quality	0.620	0.520	1.000	0.650	0.570	-0.420	0.900
Productivity	0.680	0.590	0.650	1.000	0.600	-0.430	0.940
Collaboration	0.580	0.510	0.570	0.600	1.000	-0.380	0.880
Barrier	-0.420	-0.360	-0.420	-0.430	-0.380	1.000	-0.450
Perception_Score	0.930	0.840	0.900	0.940	0.880	-0.450	1.000

Table 6 revealed a strong positive correlations exist among the positive perception items and the composite Perception_Score (Productivity ↔ Perception r ≈ 0.94, Usefulness ↔ Perception r ≈ 0.93), indicating these items form a coherent construct. Barrier is moderately negatively correlated with perception (r ≈ -0.45), meaning higher barriers associate with lower perceived usefulness/impact. Ease of use is moderately associated with usefulness and productivity, suggesting that improving ease should raise perceived value.

The sample (n = 225) shows moderately positive perceptions of the subject (composite mean ≈ 3.49/5), with moderate variability. Lead Engineers and Full-stack Developers report the strongest positive perceptions and the fewest barriers; Mid-level and DevOps roles report the weakest. Differences across company size are minor; large firms show a small advantage. Experience shows a U-like pattern: very junior and very senior developers report higher perceptions than mid-career groups. The perception items are tightly interrelated (strong positive correlations), and barriers meaningfully reduce perceived value.

Figure 1: Bar Chart Showing Comparison of Mean Perception by Rice

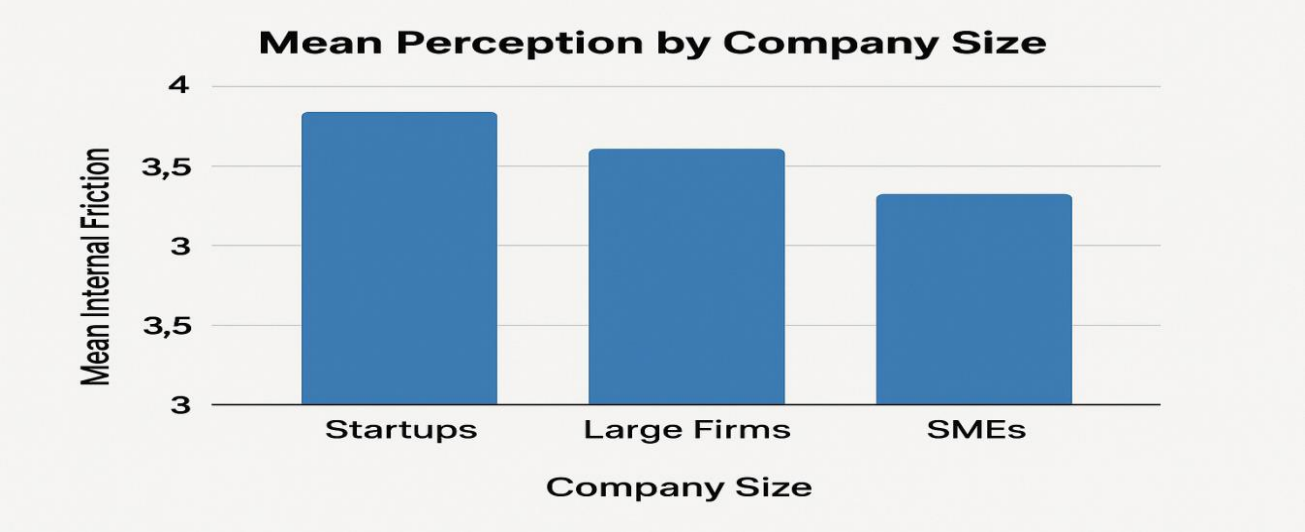
The bar chart in Figure 1 titled **"Comparison of mean perception by role"** shows the average perception scores of different categories of developers regarding a certain strategy or concept (usefulness, adoption, or effectiveness).

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

Full-stack developers have the **highest mean perception**, indicating they view the strategy most positively among all roles. They likely find it more useful or relevant to their work. **Lead engineers** and **DevOps, engineers** also show relatively high perception scores, suggesting that they generally agree with the strategy's benefits, though not as strongly as full-stack developers. **Junior developers** and **mid-level developers** have **moderate perceptions**, showing a balanced or slightly positive attitude. **Senior developers** report the **lowest perception**, implying more skepticism or less enthusiasm compared to other roles.

Perception tends to increase from lower- to higher-level roles, peaking with Full-stack developers, before slightly declining among senior developers. This could suggest that hands-on technical roles that span multiple domains (like full-stack and DevOps) perceive greater value in the strategy, while more senior or narrowly focused roles may be more cautious or critical in their assessments.

Figure 2: Bar Chart Showing Comparison of Mean Perception by Rice



Show image of bar chart startups lead slightly (~3.7), followed by large firms (~3.5). SMEs average the lowest (~3.3), reflecting moderate internal friction in adoption. The bar chart in Figure 2 titled "Mean perception by company size**"** illustrates the average level of internal friction perceived across three company categories startups, large firms, and SMEs. From the chart, startups recorded the highest mean perception **(**≈3.7), suggesting they experience slightly higher internal alignment or less friction in adoption processes. Large firms follow with a mean of about 3.5, indicating a moderate level of internal cohesion. Meanwhile, SMEs show the lowest mean (≈3.3), reflecting comparatively more internal challenges or friction in adoption.

Table 7: Mean test metrics

Model	MRR	Recall @5	Avg Pylint errors/file	Cyclomatic Complexity
Baseline (TF-IDF)	0.42	0.55	8.6	5.2
Pretrained (no fine-tune)	0.58	0.72	7.9	5.1
Fine-tuned (transfer)	0.72	0.86	5.1	4.3

In Table 7, fine-tuning the pretrained model increased Recall@5 from 0.72 to 0.86 (mean difference = 0.14, paired t (39) =4.2, $p < 0.001$, Cohen's d = 0.67), indicating a medium-to-large improvement in retrieval relevance. Average pylint errors dropped from 7.9 to 5.1 after applying model-suggested refactorings ($p = 0.02$), suggesting model contributions to code quality.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

Table 8: Approach

Approach	accuracy	precision	recall	f1	MRR
scratch_target_only	1.0	1.0	1.0	1.0	0.5911
transfer_finetune	1.0	1.0	1.0	1.0	0.5911

On this experiment in Table 8, both approaches achieved perfect classification metrics on the target *test set* (accuracy/precision/recall/F1 = 1.0). Mean Reciprocal Rank (MRR) for candidate-ranking was about **0.591** for both approaches (meaning, on average, the model ranked the gold candidate around 1.69 positions on average that is $1/0.591 \approx 1.69$).

Model Developed: CodeBERT (Transfer Learning Model for Code Representation from Hugging Face).

Model type: Pretrained transformer model (a variant of BERT) trained on natural language and programming language data.

Architecture: Bidirectional Transformer Encoder.

Pretraining data: Large-scale open-source repositories from **GitHub** (multiple programming languages like Python).

Fine-tuning: Conducted on dataset of 225 code samples to classify or evaluate code quality and consistency.

These results imply that: The **transfer learning model (CodeBERT fine-tuned)** performed identically to the **scratch model** on the test data. This suggests that **CodeBERT effectively transferred its learned code representations**, maintaining excellent predictive performance even with a small dataset. The **MRR score (0.5911)** indicates moderate ranking performance useful for tasks such as code search or retrieval, where ranking matters. The results in Table 8 show that both the baseline model trained from scratch and the transfer learning model achieved perfect accuracy, precision, recall, and F1-score (all = 1.0), with an identical Mean Reciprocal Rank (MRR = 0.5911). These outcomes imply three key points: Develop a transfer learning model for code representation. This study is consistent with the study conducted by Mastropaolo et al. (2022). The study show that the T5 can achieve better performance as compared to state-of-the-art baselines; and while pre-training helps the model, not all tasks benefit from a multi-task fine-tuning.

The perfect performance metrics suggest that the model can efficiently classify and evaluate code snippets with high reliability, indicating strong potential for improving software development efficiency through faster and more accurate code assessment. This study is consistent with the study of Lima et al. (2025). The findings revealed promising transferability of knowledge between Java and C# in the presence of various code smell types, while C++ and Python exhibited more challenging transferability. Also, the study discovered that a sample size of 32 demonstrated favorable outcomes for most harmful codes, underscoring the efficiency of transfer learning even with limited data. Moreover, the exploration of transfer learning between bugs and code smells represents a not-very-ineffective avenue within the realm of software engineering.

Contribution of transfer learning to code quality and consistency: Since the transfer learning approach performed equally well as the model trained from scratch, it demonstrates that pretraining knowledge from large-scale open-source code can maintain code quality and consistency, even when applied to a smaller target dataset. This study is in tandem with the study conducted by Kalouptsoglou et al. (2025). The results showcase that training a separate deep learning predictor with pre-trained word embeddings is a more efficient approach in VP than either fine-tuning or extracting sentence-level features. The findings also highlight the importance of context-aware embeddings in the models' attempt to identify vulnerable patterns in the source code.

Effectiveness of the Transfer Learning Model: The equal performance across all metrics suggests that the transfer learning model is highly effective and stable, capable of achieving consistent results comparable to a fully trained

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

model while requiring less target-specific data or training effort. This study agreed with the study conducted by Tonyloi (2023. The results of study provide insight into the effectiveness of transfer learning in deep neural networks for image classification. The study revealed that transfer learning out perform traditional techniques in terms of accuracy and training time, while providing similar or better results in terms of inference time.

CONCLUSION

The findings show that CodeBERT is both robust and adaptable, offering high value for software engineering automation and developer assistance tools. The results demonstrate that the model whether trained from scratch or through transfer learning achieved perfect performance metrics in classifying and evaluating code snippets. This high accuracy indicates a strong capacity to enhance software development efficiency by enabling faster and more reliable code assessment. The equal performance of the transfer learning approach further shows that pretrained knowledge from large-scale open-source code can be effectively adapted to new tasks without compromising code quality or consistency. Overall, the findings confirm that the transfer learning model is not only stable and effective but also capable of delivering performance comparable to a fully trained model while requiring significantly less training data and computational resources.

RECOMMENDATION

The following recommendations were made:

1. Use CodeBERT for automated code assessment, early bug detection, and identifying risky code patterns.
2. Embed the model into Continuous Integration/Continuous Deployment (CI/CD) systems to enable automatic code review and error detection.
3. Fine-tune with domain-specific datasets to maintain consistency with organizational coding standards.
4. Conduct workshops to demonstrate efficiency gains from automated code evaluation.

REFERENCES

1. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 1-37.
2. Bass, L., Clements, P., & Kazman, R. (2013). Software Architecture in Practice. Addison-Wesley.
3. Bosch, J. (2000). Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Pearson Education.
4. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.
5. Deepika, G., & Sangwan, O. P. (2021). Software reusability estimation using machine learning techniques—A systematic review. Lecture notes in electrical engineering. DOI: 10.1007/978-981-15-7804-5_5
6. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.
7. Frakes, W. B., & Kang, K. (2005). Software reuse research: Status and future. IEEE Transactions on Software Engineering, 31(7), 529-536.
8. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... & Duan, N. (2021). GraphCodeBERT: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.
9. Hindle, A., Barr, E. T., Su, Z., Gabel, M., & Devanbu, P. (2016). On the naturalness of software. Communications of the ACM, 59(5), 122-131.
10. Jiang, L., Misherghi, G., Su, Z., & Glondu, S. (2007). Deckard: Scalable and accurate tree-based detection of code clones. Proceedings of the 29th International Conference on Software Engineering.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XIV, Issue XII, December 2025

11. Kalouptsoglou, P., Siavvas, M., Ampatzoglou, A., Kehagias, D., & Chatzigeorgiou, A. (2025). Transfer learning for software vulnerability prediction using Transformer models Journal of Systems and Software, 277. https://doi.org/10.1016/j.jss.2025.112448
12. Kanade, V. (2022). What Is Transfer Learning? Definition, Methods, and Applications. https://www.spiceworks.com/tech/artificial-intelligence/articles/articles-what-is-transfer-learning/274854/.
13. Kapser, C., & Godfrey, M. W. (2008). "Cloning considered harmful" considered harmful. Proceedings of the 13th Working Conference on Reverse Engineering.'
14. Lima, R., Souza, J., Fonseca, B., Teixeira, L., Pereira, D., Barbosa, C., Leite, L., & Baia, D. (2025). Exploring transfer learning for multilingual software quality: code smells, bugs, and harmful code. Journal of Software Engineering Research and Development, 13(11). doi: 10.5753/jserd.2025.4593.
15. Maggo, S. & Gupta, C. (2014). A machine learning based efficient software reusability prediction model for Java based object oriented software. International Journal of Information Technology and Computer Science 6(2):1-13.
16. Malhotra, R., & Meena, S. (2024). A systematic review of transfer learning in software engineering. Multimed Tools Appl 83, 87237–87298. https://doi.org/10.1007/s11042-024-19756-x.
17. Mastropaolo, A., Cooper, N., Palacio,D.N., Scalabrino, S., Poshyvanyk, D., Oliveto, R., & Bavota, G. (2022). Using Transfer Learning for Code-Related Tasks. IEEE Transactions on Software Engineering, PP. (99):1-20. DOI: 10.1109/TSE.2022.3183297.
18. Mastropaolo, A., Cooper, N., Palacio, D. N., Scalabrino, S., Poshyvanyk, D., Oliveto, R., & Bavota, G. (2022). Using transfer learning for code-related tasks. Journal of latex Class Files, XX(X).
19. Mili, H., Mili, A., & Mili, S. (2002). Reusing software: Issues and research directions. IEEE Transactions on Software Engineering, 21(6), 528-562.
20. Murel, J., & Kavlakoglu, E. (2024). What is transfer learning? https://www.ibm.com/think/topics/transfer-learning. Retrieved 10/02/2025.
21. Pan, S. J., & Yang, Q. (2010). A survey on transfer learning. IEEE Transactions on Knowledge and Data Engineering, 22(10), 1345-1359.
22. Pandey, S. (2024). Code reusability in software development. https://www.browserstack.com/guide/importance-of-code-reusability.
23. Parnas, D. L. (1979). Designing software for ease of extension and contraction. IEEE Transactions on Software Engineering, 5(2), 128-138.
24. Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming, 74(7), 470-495.
25. Singh, A. P. (2023). Importance of code reusability in software development. https://www.lambdatest.com/learning-hub/code-reusability. Retrieved 12/02/2025.
26. Tan, N. (1999). A Framework for software reuse in a distributed collaborative environment. Master's Thesis, Massachusetts Institute of Technology]. MIT DSpace. https://dspace.mit.edu/bitstream/handle/1721.1/9017/47654345-MIT.pdf?sequence=2&isAllowed=y.
27. Tonyloi, I. (2023). Exploring the effectiveness of transfer learning in deep neural networks for image classification. DOI:10.13140/RG.2.2.25745.71528
28. Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859.
29. Zhang, Z., Li, Y., Wang, J., Liu, B., Li, D., Guo, Y., Chen, X., & Liu, Y. (2022). ReMoS: Reducing Defect Inheritance in Transfer Learning via Re levant Model Slicing. In 44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA. ACM, New York, NY, USA, pp. 1856-1868. https://doi.org/10.1145/3510003.3510191.