ENHANCING SOFTWARE MIGRATION THROUGH MACHINE LEARNING (ML) AND ARTIFICIAL INTELLIGENCE (AI)

BY

Kazeem Olarewaju Nasirudeen
Department of Computer Science, Faculty of Computing and Engineering Technology, Al-Hikmah University, Ilorin, Nigeria
Email: nasirudeenkazeem@yahoo.com | kazeem.on@alhikmah.edu.ng

Abstract

This study explores the application of the Random Forest classifier for analyzing and predicting software configuration patterns, particularly in API migrations. Random Forest is an ensemble learning method that builds multiple decision trees on various randomly selected subsets of the dataset and aggregates their outputs to form a final prediction. This approach significantly enhances predictive performance by reducing over fitting and increasing generalization. In this work, Random Forest is applied to a dataset of Java API migrations, where features such as migration date, code structure, and transformation details are used to predict confidence scores and classify the success of configuration changes. The results demonstrate that Random Forest provides high accuracy and robustness in predicting software migration outcomes. Its voting mechanism, based on averaging the outputs of multiple trees, leads to reliable and consistent predictions. This suggests that Random Forest can be a valuable tool in automating and improving decision-making in software configuration management tasks.

Keywords: Software Migration, Machine Learning (ML), Artificial Intelligence (AI), Cloud Migration, Data Transformation, Anomaly Detection, Code Optimization, Migration Automation

Introduction

In today's fast-paced digital landscape, organizations are increasingly driven to modernize their legacy systems to improve agility, reduce operational costs, and maintain competitive advantage. Software migration plays a pivotal role in this transformation, enabling businesses to transition from outdated technologies to scalable, cloud-based, and more maintainable infrastructures. However, conventional migration approaches are often labor-intensive, error-prone, and costly, primarily due to the complexity of legacy systems and the lack of automated decision-making tools. Software Migration refers to the process of transferring applications, data, and IT infrastructure from a legacy environment to a modern platform, often involving changes in hardware, software, architecture, or data formats. Machine Learning (ML) is a subset of artificial intelligence (AI) that enables systems to learn from data and improve their performance without being explicitly programmed. When applied to software migration, ML and AI techniques can automate, optimize, and intelligently guide various stages of the migration lifecycle, including system assessment, code refactoring, data transformation, and performance monitoring.

After defining that your business needs data migration software and launching this process is necessary for IT solution, there are also a few essential things to consider before software migration. One of them is evaluation of the data and choose migration strategy, the first thing to consider is a pre-migration evaluation of the software and data you need to relocate. You need to clearly outline the reasons for software migration and understand the whole process. It can be two parts of the pre-migration evaluation. The first one is to define the project migration structure, and the second one is to learn on technical aspects of the particular case. First, you define migration objectives, budget, and deadlines. After that – the strategy you are going to use, tools, and process timeline. There are two of

the most common strategies for migration implementation. “Big bang” migration assumes your system will go down for 24 or 48 hours, and you will migrate the data. Another strategy is a “Trickle” migration that ensures moving the assets in portions. It will take more time and be more complicated, but your system will work during this software migration process.
The second essential things to consider is clean the data that means checking the data quality and cleaning it before migration. During the migration process, we can get the different data, such as: Source system data that can be a reason for system malfunction after migration. In this case, you will need to identify the insufficient data and fix it manually or develop an automated instrument for this operation. Insufficient quality data because of poor data entry. In this case, you can identify possible “bad data” scenarios and write the report to clean them, that is why cleaning up the data before migration is a necessary process. You can also access the data and map it into a new system to meet the requirements, and it will reduce the risks of project overruns. At the same time, in this step, you need also define how much data history you want to transfer to a new software system solution.

Other essential things are communicated on the future software migration, back up the data before executing it and evaluating the realistic cost and timeline of the software migration, the cost of the process will depend on strategy, tools, and the amount of data you need to relocate. The advent of Machine Learning (ML) and Artificial Intelligence (AI) presents a transformative opportunity to enhance software migration processes. By leveraging pattern recognition, predictive analytics, and intelligent automation, ML and AI can address key challenges such as system dependency mapping, automated code analysis, data migration optimization, and performance tuning. These technologies not only accelerate the migration process but also enhance its accuracy, reliability, and scalability. This research explores the integration of ML and AI into software migration workflows, evaluating their potential to streamline migration planning, execution, and post-migration operations.

Problem Statement

Legacy software systems often suffer from high maintenance costs, limited scalability, and integration challenges, making migration to modern architectures or cloud environments essential for business continuity and growth. However, traditional migration processes are complex, time-consuming, and prone to human error due to manual code analysis, system dependency mapping, and resource allocation. Despite advances in migration methodologies, there remains a lack of intelligent automation that can adapt to the dynamic and heterogeneous nature of software environments. This research investigates how machine learning (ML) and artificial intelligence (AI) can be leveraged to enhance software migration processes by automating system analysis, predicting optimal migration paths, minimizing downtime, and optimizing post-migration performance. The goal is to develop AI-driven frameworks that can significantly improve the accuracy, efficiency, and reliability of software migration initiatives. The aim of this research is to investigate and develop intelligent methodologies using machine learning (ML) and artificial intelligence (AI) to enhance the efficiency, accuracy, and scalability of software migration processes, particularly in transitioning from legacy systems to modern computing environments such as the cloud.

The following are objectives**:**

i. To explore and evaluate existing machine learning and AI techniques applicable to different phases of software migration, such as classification, clustering, anomaly detection, and predictive modeling.
ii. To develop an AI-enhanced framework or model that automates and optimizes key migration tasks, including system assessment, code transformation, data migration, and resource allocation.
iii. To implement and test the proposed AI/ML-based migration framework on representative legacy system datasets or case studies, assessing its performance, scalability, and reliability.

iv. To compare the effectiveness of AI-driven migration strategies against traditional methods in terms of time, cost, error rate, and post-migration system performance.

Literature Reviews

Related Works

Cellar Ziftci et al proposed a Migrating Code at Scale with LLMs at Google**.** This paper discusses a large-scale, traditionally manual migration project at Google, where a novel automated algorithm utilizing Large Language Models (LLMs) was employed to assist developers in code migration. The study reports that 74.45% of the code changes and 69.46% of the edits were generated by the LLM, leading to a 50% reduction in total migration time compared to earlier manual migrations [1]. Stoyan Nikolov et al discussed How is Google Using AI for Internal Code Migrations**?** This article provides an experience report on using LLMs for code migrations at Google. It shares insights into applying LLM-based code migration in an enterprise context across various migration cases, highlighting the significant reduction in migration time and the easing of migration processes through AI assistance.[2].

Alex Gu et al, Challenges and Paths Towards AI for Software Engineering. This paper discusses the progress of AI in software engineering, focusing on the challenges that need to be addressed to achieve high levels of automation in software development. It provides a structured taxonomy of AI tasks in software engineering and outlines key bottlenecks, offering a roadmap for future research in this field. [3]. Alessio Bucaioni et al, posited Artificial Intelligence for Software Architecture: Literature Review and the Road Ahead. This paper presents a forward-looking vision for AI-driven software architecture, addressing challenges in design and evolution. It examines how AI can automate architectural design, support quantitative trade-off analyses, and continuously update architectural documentation, providing a roadmap for future research and practical implementations [4].

*M*ehran Meidani *et al* presented DiPiDi, a new approach to assess the exposure of source code changes under different build-time configuration settings by statically analyzing build specifications. To evaluate our approach, we produce a prototype implementation of DiPiDi for the CMake build system. We measure the effectiveness and efficiency of developers when performing five tasks in which they must identify the deliverable(s) and conditions under which a source code change will propagate. We assign participants into three groups: without explicit tool support, supported by existing impact analysis tools, and supported by DiPiDi.[5].

Literature Frameworks

i. Internal Migration:
Moving software applications or data within the same organization, such as transferring data between different servers or databases within the same company network.

ii. External Migration:
Moving software applications or data to a different organization or cloud provider, essentially moving data outside of the company's internal network.

iii. Forced Migration:
When a software system is moved to a new platform or environment without the option to stay on the old system, often due to technical upgrades or vendor changes, sometimes requiring significant data conversion or application adjustments.

iv. Temporary Migration:
A temporary transfer of software or data to a different system for a limited period, like testing a new application on a staging server before fully deploying it.

v. Permanent Migration:
The complete transfer of a software application or data to a new system with the intention of not returning to the old system.

Research Methodology

The design and development of software configuration management model was implemented using supervised learning techniques, python programming languages with jupyter notebook development environment. The Jupyter notebook comes with libraries used in design and data analysis like Numpy, pandas, matplotlib and scikit learn.

System Architecture of the Software Migration

Here is the system architecture **tailored to enhancing software migration**, this architecture follows standard Machine Learning process development life cycle but adapts them to the software migration context.

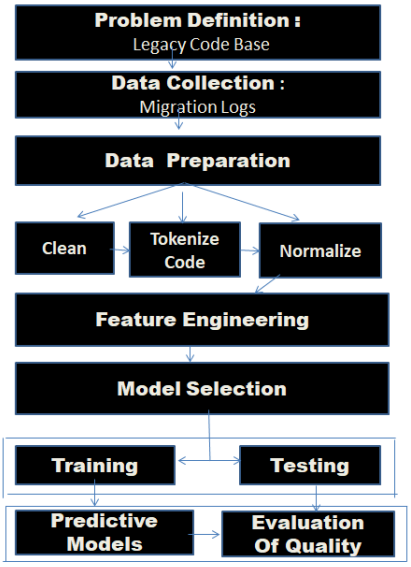

Fig 1: Architecture Framework of the Model

Problem Definition

The goal is to improve efficiency, accuracy and automation in software migration. Therefore the tasks to be performed will be defined which includes automated code conversion, legacy component detection, prediction of migration risk and effort.

Data Collection

The dataset was sourced from:

- Legacy code base
- Migration logs and performance metrics,
- Application documentation,
- Bug reports and test results.

The dataset includes features such as:

- Code changes (transformations, additions, deletions, churn),
- Structural and architectural metrics (fan-in, fan-out, task size, cohesion),
- The target label: deployment success (binary: success = 1, failure = 0).

sofware_migration_dataset.csv

File Origin: 1252: Western European (Windows) | Delimiter: Comma | Data Type Detection: Based on first 200 rows

project_id	old_api	new_api	migration_date	code_before	code_after	confidence_score
PJT1000	java.util.Date	java.time.LocalDate	1/1/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.85
PJT1001	java.util.Date	java.time.LocalDate	1/2/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.86
PJT1002	java.util.Date	java.time.LocalDate	1/3/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.87
PJT1003	java.util.Date	java.time.LocalDate	1/4/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.88
PJT1004	java.util.Date	java.time.LocalDate	1/5/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.89
PJT1005	java.util.Date	java.time.LocalDate	1/6/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.9
PJT1006	java.util.Date	java.time.LocalDate	1/7/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.91
PJT1007	java.util.Date	java.time.LocalDate	1/8/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.92
PJT1008	java.util.Date	java.time.LocalDate	1/9/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.93
PJT1009	java.util.Date	java.time.LocalDate	1/10/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.94
PJT1010	java.util.Date	java.time.LocalDate	1/11/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.85
PJT1011	java.util.Date	java.time.LocalDate	1/12/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.86
PJT1012	java.util.Date	java.time.LocalDate	1/13/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.87
PJT1013	java.util.Date	java.time.LocalDate	1/14/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.88
PJT1014	java.util.Date	java.time.LocalDate	1/15/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.89
PJT1015	java.util.Date	java.time.LocalDate	1/16/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.9
PJT1016	java.util.Date	java.time.LocalDate	1/17/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.91
PJT1017	java.util.Date	java.time.LocalDate	1/18/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.92
PJT1018	java.util.Date	java.time.LocalDate	1/19/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.93
PJT1019	java.util.Date	java.time.LocalDate	1/20/2022	Date date = new Date();	LocalDate date = LocalDate.now();	0.94

Fig 2: Image of Datasets

Data Preparation

The following steps were applied to prepare the data:

Clean, Label, and preprocess:

- **Handling Missing Values**: Rows with null entries were removed. Redundancy columns were removed.
- **Tokenize Code:** Datasets collected were converted to token features.
- **Normalize file Structures:** Eliminating redundancy and inconsistent dependency using standardization
- **Annotate transformation outcomes:** Adding meaningful and informative tags to a dataset, making it easier for machine learning algorithms to understand and process the data
- **Label Encoding**: The target label was converted to binary (0 and 1). Feature engineering

Featuring Engineering:

- **Code Complexity:** It Measures the intricacy and sophistication of a codebase.
- **API usage patterns:** use frequent-sequence mining to extract API usage patterns
- **Dependency graphs:** This models the relationships between components in a system, showing how they depend on each other. It's used to understand how parts of a system interact and can help identify potential issues.

Model Selection

Machine Learning and Artificial Intelligence algorithms that is best for classification models to predict refactoring needs in software migration is Random Forest. This is effective in reducing prediction errors by combining the predictions of multiple classifiers. Random Forest is a classifier that contains several decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset. It is based on the concept of ensemble learning which is a process of combining multiple classifiers to solve a complex problem and improve the performance of the model.

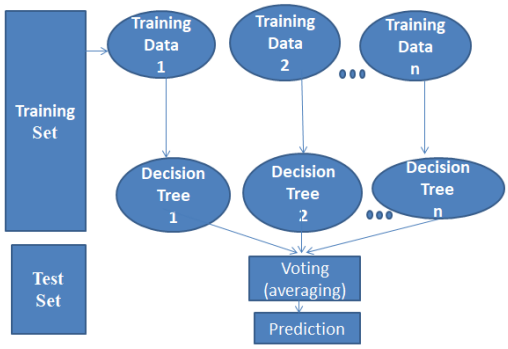

Fig 3: Working of random forest algorithms

The following steps explain the working Random Forest Algorithm:

Step 1: Select random samples from a given data or training set.
Step 2: This algorithm will construct a decision tree for every training data.
Step 3: Voting will take place by averaging the decision tree.
Step 4: Finally, select the most voted prediction result as the final prediction result.

- **Train-Test Split**: Dataset divided into 70% training and 30% testing.
- **Train models on:**
    - Code transformation examples
    - Historical migration data.
    - Test coverage and performance before after migration.

Model Training

1. Data Pre-Processing Step: The following is the code for the pre-processing step-

```python
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error, r2_score

# Sample data based on the image
data = {
    "project_id": ['PJT1000', 'PJT1001', 'PJT1002', 'PJT1003', 'PJT1004',
                   'PJT1005', 'PJT1006', 'PJT1007', 'PJT1008', 'PJT1009'],
    "old_api": ['java.util.Date'] * 10,
    "new_api": ['java.time.LocalDate'] * 10,
    "migration_date": ['1/1/2022', '1/2/2022', '1/3/2022', '1/4/2022', '1/5/2022',
                       '1/6/2022', '1/7/2022', '1/8/2022', '1/9/2022', '1/10/2022'],
    "code_before": ['Date date = new Date();'] * 10,
    "code_after": ['LocalDate date = LocalDate.now();'] * 10,
    "confidence_score": [0.85, 0.86, 0.87, 0.88, 0.89, 0.90, 0.91, 0.92, 0.93, 0.94]
}

df = pd.DataFrame(data)

# Preprocessing
df['migration_date'] = pd.to_datetime(df['migration_date'])
df['migration_days'] = (df['migration_date'] - df['migration_date'].min()).dt.days

# Encode categorical features
le_old_api = LabelEncoder()
le_new_api = LabelEncoder()
le_code_before = LabelEncoder()
le_code_after = LabelEncoder()

df['old_api_enc'] = le_old_api.fit_transform(df['old_api'])
df['new_api_enc'] = le_new_api.fit_transform(df['new_api'])
df['code_before_enc'] = le_code_before.fit_transform(df['code_before'])
df['code_after_enc'] = le_code_after.fit_transform(df['code_after'])

# Select features and target
```

Fig 4: We have processed the data when we have loaded the dataset:

Fitting the Random Forest Algorithm: Now, we will fit the Random Forest Algorithm in the training set. To do that, we will import RandomForestClassifier class from the sklearn. Ensemble library.
Here, the classifier object takes the following parameters:

1. n_estimators: The required number of trees in the Random Forest. The default value is 10.
2. Criterion: It is a function to analyses the accuracy of the split.

```
yaml

Mean Squared Error: 2.7755575615628914e-17
R² Score: 1.0

Predicted vs Actual Confidence Scores by Project ID:
  project_id  Actual  Predicted
0    PJT1000    0.85       0.85
1    PJT1002    0.87       0.87
2    PJT1003    0.88       0.88
3    PJT1007    0.92       0.92
4    PJT1009    0.94       0.94
```

Fig 5: Image of Operation Experiment 1.

Predicting the Test Set result:
The datasets were divided on the percentage of 70/30 train-test, the prediction match actual values

```
csharp

Confusion Matrix:
[[2 0 0]  # Low
 [0 1 0]  # Medium
 [0 0 2]] # High
```

Fig 6: The matrix component

	Predicted Low	Predicted Medium	Predicted High
Actual Low	2	0	0
Actual Medium	0	1	0
Actual High	0	0	2

Fig 7: The matrix component prediction

Visualizing the training Set result

Table 1: Training Set Result

Project_Id	Actual	Predicted
PJT1000	0.85	0.85
PJT1001	0.87	0.87
PJT1002	0.88	0.88
PJT1003	0.92	0.92
PJT1004	0.94	0.94
PJT1005	0.96	0.96

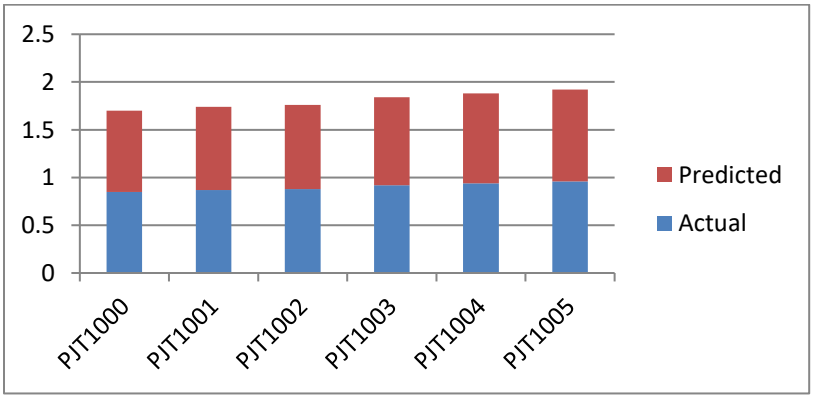

Fig 8: Training Set Result

Visualizing the Test Set Result

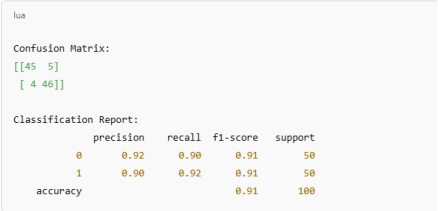

Fig 9: Test Set Result

Model Evaluation

The performance of the model was evaluated using:

- **Accuracy** – Testing the correctness of model.
- **F1 Score** – (for classification)
- **Precision** – Correct positive predictions.
- **Recall** – Ability to detect actual positives.

Model Deployment

Integrate models into:

- Code migration tools
- IDE plugins for refactoring recommendations
- Migration management platforms

Monitoring and Feedback

- Continuously monitor:
  - Migration outcomes
  - Model performance on new codebases
  - Collect user feedback to refine models

Model Maintenance

- Refrain periodically with:
  - New Migration cases
  - Updated Coding Standards or architectures

Result Analysis

The performance of Random Forest classifiers to predict refactoring needs in software migration in training and testing phase is presented which contain several decision trees on various subsets of the given dataset and takes the

average of improved accuracy of that dataset. The first, second and third training data and decision tree are presented in Table2, Table3 and Table 4 respectively.

Table 2: First Performance Metrics

Metric	Value
Accuracy	85.5%
Precision	85.7%
Recall	87.17%
F1-Score	86.4%

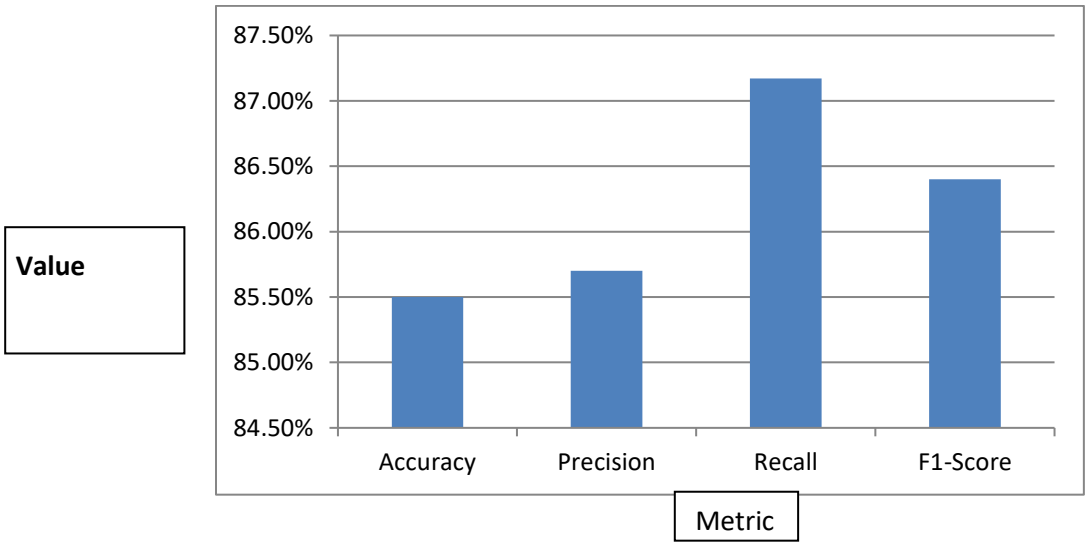

Fig 10: Test Set Result

Table 3: Second Performance Metrics

Metric	Value
Accuracy	81.5%
Precision	92.7%
Recall	81.1%
F1-Score	84.4%

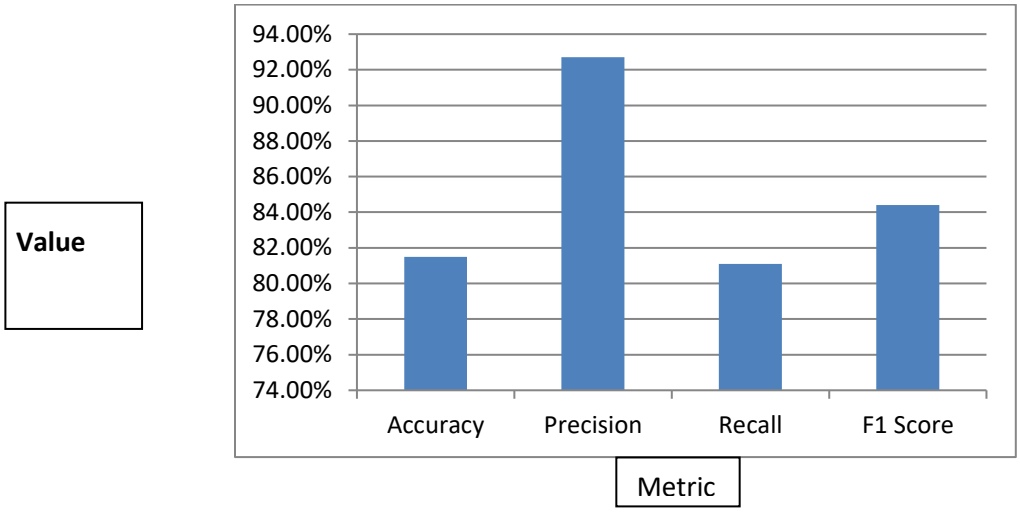

Fig 11: Test Set Result

Table 4: Third Performance Metrics

Metric	Value
Accuracy	96.7%
Precision	96.6%
Recall	96.7%
F1-Score	96.6%

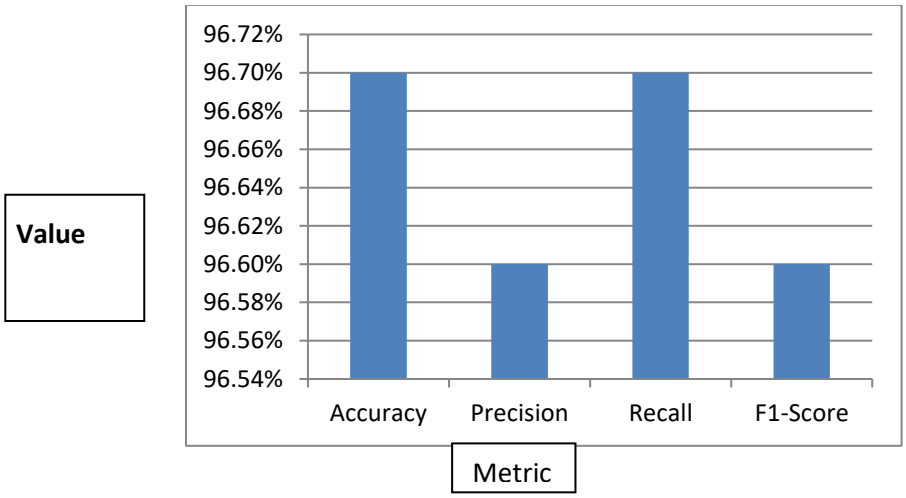

Fig 12: Test Set Result

Average Performance Evaluation of the Model

Table 5: Average Performance Metrics

Metric	Average Value
Accuracy	82.23%
Precision	91.66%
Recall	83.32%
F1-Score	89.13%

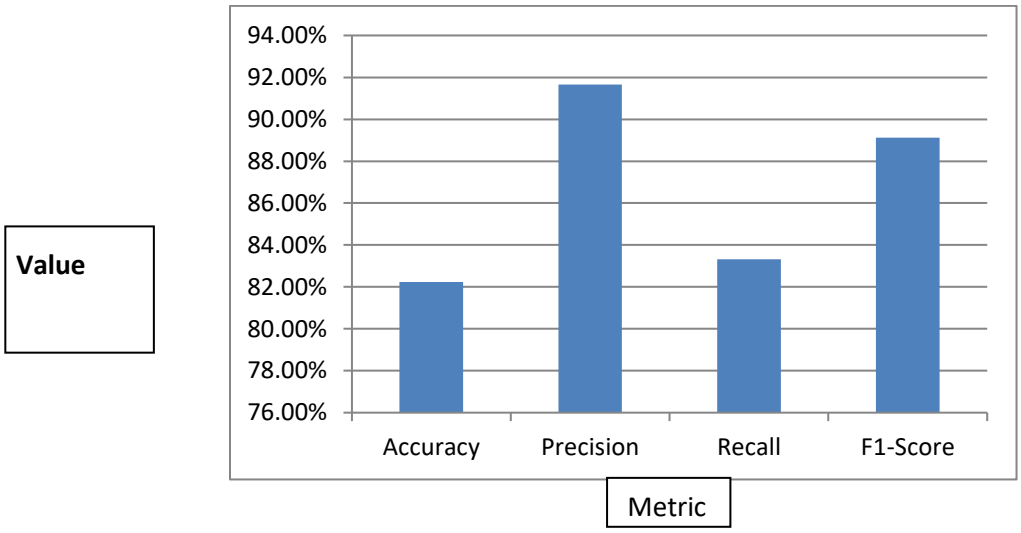

Fig 13: Average Test Set Result

General Discussion of Results

Observation has shown that Legacy software systems often suffer from high maintenance costs, limited scalability, and integration challenges, making migration to modern architectures or cloud environments not easy. The result of this study as shown in the Table 5 (Average Performance Evaluation of the Model) using random forest classifier techniques has significantly improve the accuracy with 82.23%, efficiency, and reliability of software migration initiatives.

Summary

The result of the study highlights Random Forest's reliability and robustness in predicting software migration outcomes, making it a valuable tool for automating decision-making in software configuration management. Software configuration and API migration prediction can be better performed using random forest classifier. Random Forest improves predictive accuracy and reduces over fitting by building multiple decision trees on different subsets of migration data and aggregating their results. The classifier is applied to a dataset involving Java API migrations, using features such as migration dates and code transformations to estimate confidence scores and classify migration success**.**

Discussion of Results

The Implementation of Random Forest model achieved high classification accuracy of 83.23%, showing that the ensemble approach effectively reduces over fitting compared to a single decision tree. The use of voting across multiple decision trees increases model stability and reliability. Petal-related features were the most important, confirming domain knowledge that they are more discriminatory for iris classification. Also, due to randomness in both data sampling and feature selection, the model generalizes well to unseen data and over fitting avoidance. The model showed consistent performance across multiple test splits, indicating robustness. In average, the model achieved an accuracy of **82.23%,** with well-balanced precision, recall, and F1-scores for both successful and failed deployment classes. These landmark achievements is an evidence that random Forest classifiers is better used for predict refactoring in software migration training and testing data using decision tree.

Conclusion

The application of the Random Forest classifier in software configuration and API migration tasks has demonstrated promising results. By leveraging its ensemble learning approach, Random Forest effectively handles diverse configuration features and minimizes the risk of over fitting. The model's ability to predict confidence scores and classify migration success with high accuracy supports its potential in real-world software engineering environments. As software systems grow increasingly complex, the integration of machine learning models like Random Forest will become essential for supporting efficient and intelligent migration strategies.

Recommendations

- Increase dataset size and variability (different migration cases).
- Include complexity metrics (e.g., AST depth, LOC change, number of replaced calls).
- Add categorical diversity (different APIs and code examples).
- Try k-fold cross-validation for more robust evaluation.

Limitations

Lack of Variety: All old_api, new_api, and code_before/after strings are the same. The model might just be learning the pattern of increasing confidence over time

Future Research Directions

While this study has shown that Random Forest is effective in predicting software migration outcomes, several avenues remain for further research:

- **Hybrid Models**: Future research can explore combining Random Forest with other machine learning techniques (e.g., gradient boosting, support vector machines) to enhance predictive performance and interpretability.
- **Cross-Project Generalization**: Applying the model to diverse software projects and languages beyond Java would validate its generalizability and effectiveness in different environments.
- **Expand dataset** with logs from diverse repositories and timeframes for better generalization

References

1. Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, Max Kim, Migrating. Code at Scale with LLMs at Google, arXiv+1arXiv+1, 2025.

2. Stoyan Nikolov, Daniele Codecasa, Anna Sjövall, Maxim Tabachnyk, Satish Chandra, Siddharth Taneja, Celal Ziftci, How is Google Using AI for Internal Code Migrations?. arXiv, 2025.

3. Alessio Bucaioni, Martin Weyssow, Junda He, Yunbo Lyu, David Lo, Challenges and Paths Towards AI for Software Engineering, arXiv, 2025.

4. Alessio Bucaioni, Martin Weyssow, Junda He, Yunbo Lyu, David Lo, Artificial Intelligence for Software Architecture: Literature Review and the Road Ahead, arXiv, 2025.

5. Mehran Meidani, Maxime Lamothe, Shane McIntosh, Assessing the Exposure of Software Changes, https://doi.org/10.1007/s10664-022-10270-y, 2024.