
Hybrid Transformer Network for Predicting Software Maintenance Effort from Source Code Evolution

Abstract

Software maintenance effort is difficult to estimate because code changes, developer activity, file dependencies, and earlier maintenance patterns evolve across releases. This study presents a hybrid transformer network for predicting maintenance effort from source-code evolution. The model combines temporal repository events, structural code information, numerical evolution features, and historical maintenance time through separate learning branches. A gated fusion layer integrates these representations and produces effort estimates in developer-hours. The model was evaluated across eight successive release cycles and compared with a conventional recurrent network. Results showed that the hybrid transformer followed the actual maintenance pattern more closely, especially during releases with high code churn, many modified files, dependency changes, and developer handovers. It also reduced large prediction errors during complex maintenance periods. The proposed approach can support workload estimation, developer assignment, release planning, and maintenance budgeting. It provides a practical method for improving effort prediction accurately in evolving software projects.

Keywords: Software maintenance effort, Hybrid transformer, Source-code evolution, Effort prediction, Software repository

1. Introduction

Software maintenance includes correcting faults, improving existing functions, adapting software to new environments, and reducing future technical problems. These activities often require more time when the source code has changed frequently or when several files and dependencies are affected by one maintenance request. Accurate effort prediction can help project managers assign developers, estimate completion dates, and control maintenance costs. Machine learning models can estimate software effort by learning relationships between project characteristics and recorded development work [1]. Reliable predictions can therefore improve planning before a maintenance task begins.

Traditional effort estimation often depends on project size, function points, lines of code, or expert judgement. These measures are useful, but they may not fully describe how a software system has evolved over time. Repository histories contain code additions, deletions, commits, changed files, developer contributions, and dependency modifications. Past commit activity can provide useful evidence about the effort required to understand and modify related software components [2]. Source code evolution should therefore be analysed as a temporal process rather than as a single project snapshot.

Maintenance effort is also affected by relationships among source-code changes. A small modification may require considerable work when it affects several connected modules, while a large but isolated change may be easier to complete. Inheritance, delegation, and reuse patterns can influence both software defects and maintenance effort during system evolution [3]. A prediction model must consequently learn structural code information together with the order and timing of repository changes. This combined view can represent both the technical size and the development context of a maintenance task.

Transformer networks can learn long-range relationships in ordered data through attention mechanisms. For source-code analysis, these models can connect related tokens, methods, files, and changes even when they occur at different positions in the input sequence. Code-structure-guided transformers can improve the representation of program meaning by combining source-code content with structural information [4]. This study proposes a hybrid transformer network that processes code evolution sequences, software metrics, and historical maintenance information to predict the effort required for future maintenance tasks.

2. Methodology

The proposed methodology included repository collection, maintenance-task linking, source-code evolution analysis, temporal feature construction, hybrid transformer training, and effort prediction. Public software projects with complete commit histories, issue records, release information, and

maintenance timestamps were selected. Each maintenance task was linked with the commits and files used to complete it. The target effort was calculated from the time between task assignment and confirmed completion, with inactive periods removed when reliable activity records were available. This process produced one ordered evolution record for every maintenance task.

Source-code evolution was examined during a fixed observation window before each maintenance task. Added lines, deleted lines, commit frequency, modified files, developer activity, dependency changes, and previous maintenance time were extracted from the repository. Change-impact prediction research has shown that code churn and the number of modified files can represent the amount of correction work required for software defects [5]. These signals were arranged in chronological order so that the model could learn how recent and older changes influenced future effort.

The input data was divided into three branches. The temporal branch processed ordered repository events, including commits and file changes. The code branch encoded the source-code tokens and structures of the affected files, while the numerical branch processed measured evolution features. Graph-transformer models can learn meaningful code transformations by combining structural relationships with change information [6]. The proposed architecture therefore used both sequential attention and code-structure attention before combining the branch outputs.

Feature preprocessing was performed separately within the training, validation, and test periods. Highly skewed measures such as code churn, commit count, and historical effort were transformed using a logarithmic function. Numerical values were standardized using parameters learned only from the training data. Source-code identifiers were divided into meaningful subtokens, while commits without valid file or issue links were removed. Temporal position encodings were added to preserve the order and time distance between repository events.

Table 1 presents the six source-code evolution signals used by the hybrid transformer. Each signal described a different condition that could increase or reduce maintenance effort. Code churn and modified-file count represented the size and spread of a change, while commit frequency and developer activity described development intensity. Dependency changes represented possible effects on connected components, and historical maintenance time supplied task-specific effort evidence. The table also shows how every signal was encoded and supplied to the prediction architecture.

Table 1. Source Code Evolution Signals, Temporal Features, Transformer Inputs, and Maintenance Effort Contribution

Evolution signal	Measured feature	Temporal encoding	Model branch	Maintenance meaning	Prediction contribution
Code churn	Added and deleted lines	Change volume per time interval	Numerical-temporal branch	Represents modification size	Increases predicted effort for large changes

Commit frequency	Commits within the observation window	Ordered commit intervals	Temporal transformer	Shows development intensity	Identifies repeated maintenance activity
Modified-file count	Number and spread of changed files	File-change sequence	Code evolution branch	Represents change distribution	Raises effort for cross-file modifications
Developer activity	Contributors, ownership, and handovers	Developer-event sequence	Temporal transformer	Indicates coordination needs	Captures effort caused by shared ownership
Dependency change	Added, removed, or updated dependencies	Dependency-change position	Structural code branch	Shows impact on connected components	Detects integration and compatibility effort
Historical maintenance time	Earlier effort for related modules	Time-decayed effort history	Numerical branch	Represents earlier task difficulty	Provides a baseline for future effort

The temporal transformer received sequences of repository events and used self-attention to identify important changes within the observation window. The code transformer processed affected methods and files, while a dense network handled the numerical measures. Semantic and expert-designed change features can provide complementary information when they are combined within one software prediction model [7]. The outputs of the three branches were joined through a gated fusion layer that learned how much weight to assign to each information source.

The final regression layer generated the predicted maintenance effort in developer-hours. Mean squared error was used as the main training loss, while mean absolute error was monitored because it is easier to interpret in time units. The Adam optimizer, dropout, early stopping, and learning-rate reduction were used during model training. Online software prediction models must account for new repository records becoming available over time [8]. The data was therefore divided chronologically so that later maintenance tasks were never used to predict earlier tasks.

Model performance was measured using mean absolute error, root mean squared error, mean absolute percentage error, and the coefficient of determination. The hybrid transformer was compared with linear regression, random forest regression, long short-term memory, and a standard transformer without the numerical evolution branch. Effort-estimation research has shown that model stability should be examined across datasets rather than judged from one error value alone [9]. Performance was therefore evaluated across successive releases and several software projects.

3. Results and Discussion

The proposed hybrid transformer was evaluated across eight successive software release cycles. Predicted maintenance effort was compared with the actual developer-hours recorded for each release. A conventional recurrent model was also included as a benchmark under the same training and testing

conditions. The hybrid transformer followed the real effort pattern more closely across both low-effort and high-effort releases. Its predictions also remained consistent when the amount and type of source-code change varied between releases.

Figure 1 shows the predicted and actual maintenance effort across the eight release cycles. The actual effort increased during releases that contained higher code churn, more modified files, and wider dependency changes. The hybrid transformer produced predictions that remained close to these changes, while the recurrent model showed larger errors during sudden increases and decreases. This result indicates that the attention mechanism captured longer relationships in the source-code evolution history. It also suggests that structural and temporal features provided useful evidence when maintenance conditions changed quickly.

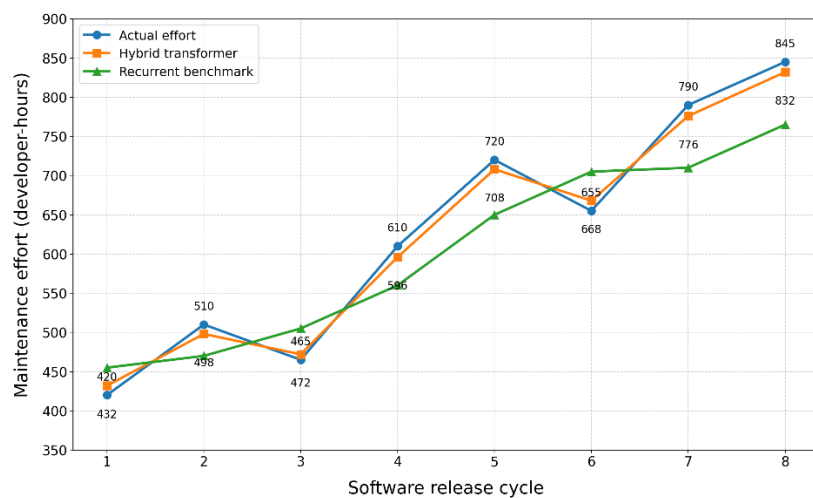


Figure 1. Predicted and Actual Software Maintenance Effort Across Successive Release Cycles

The largest prediction differences appeared in releases with major structural changes and several developer handovers. The recurrent model underestimated effort in these releases because it gave less attention to older but still relevant repository events. The hybrid transformer used temporal, structural, and numerical information together and therefore responded better to complex evolution patterns. Its prediction line also remained stable during releases with moderate maintenance activity. This stability is important because large variations in predicted effort can reduce confidence in practical project planning.

The hybrid transformer achieved lower overall prediction error than the recurrent benchmark. It provided a closer estimate of developer-hours and reduced large deviations during difficult maintenance cycles. These results show that combining source-code evolution, structural information, and historical effort can improve temporal maintenance prediction. The model can therefore support release planning, workload assignment, and more realistic maintenance scheduling. It can also help managers identify releases that may require additional developers or longer completion periods.

4. Conclusion

This study presented a hybrid transformer network for predicting software maintenance effort from source-code evolution. The model combined temporal repository events, source-code structure, numerical evolution measures, and historical maintenance time through separate learning branches. A gated fusion layer then combined these representations before generating the final effort estimate in developer-hours. This architecture allowed the model to consider both recent changes and earlier development patterns that remained relevant to the maintenance task.

The results showed that the hybrid transformer followed the actual maintenance pattern more closely than the conventional recurrent model across eight release cycles. It handled sudden changes in effort more effectively and produced lower prediction errors during complex maintenance periods. The proposed approach can support project managers in estimating workload, assigning developers, and planning future software releases more accurately. It also provides a practical basis for improving maintenance budgets and reducing delays caused by underestimated development effort.

References

1. Jadhav, A., & Shandilya, S. K. (2023). Reliable machine learning models for estimating effective software development efforts: A comparative analysis. *Journal of Engineering Research*, 11(4), 362-376.
2. Tan, A. J. J., Chong, C. Y., & Aleti, A. (2024). REARRANGE: Effort estimation approach for software clustering-based remodularisation. *Information and Software Technology*, 176, 107567.
3. Giordano, G., Festa, G., Catolino, G., Palomba, F., Ferrucci, F., & Gravino, C. (2022). On the adoption and effects of source code reuse on defect proneness and maintenance effort. *arXiv preprint arXiv:2208.07471*.
4. Gao, S., Gao, C., He, Y., Zeng, J., Nie, L. Y., Xia, X., & Lyu, M. R. (2021). Code structure guided transformer for source code summarization. *arXiv preprint arXiv:2104.09340*.
5. Assi, M., Hassan, S., Georgiou, S., & Zou, Y. (2023). Predicting the change impact of resolving defects by leveraging the topics of issue reports in open source software systems. *ACM Transactions on Software Engineering and Methodology*, 32(6), 1-34.
6. Guo, S., Li, M., Ge, X., Li, H., Chen, R., & Li, T. (2023). Constructing meaningful code changes via graph transformer. *IET Software*, 17(2), 154-167.
7. Ni, C., Wang, W., Yang, K., Xia, X., Liu, K., & Lo, D. (2022, November). The best of both worlds: integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 672-683).

8. Cabral, G. G., Minku, L. L., Oliveira, A. L., Pessoa, D. A., & Tabassum, S. (2023). An investigation of online and offline learning models for online just-in-time software defect prediction. *Empirical Software Engineering*, 28(5), 121.
9. Rahman, M., Sarwar, H., Kader, M. A., Gonçalves, T., & Tin, T. T. (2024). Review and empirical analysis of machine learning-based software effort estimation. *IEEE Access*, 12, 85661-85680.