An AI-Driven Framework for Software Effort Estimation Based on Developer Performance Metrics

Shaheer Ahmed[1], Nosheen Qamar[2], Faria Nazir[3], Nosheen Sabahat[4], Atif Ikram[5]*, Najla Abdulaziz Almousa[6], Hebah Abdullah Abubakr[7], Mohammed Abual-Rub[8], Abdulrahman Alojail[9], Marwan Abu-Zanona[10]
Department of Software Engineering, University of Management and Technology, Lahore, Pakistan [1, 2, 3]
Department of Computer Science, Forman Christian College University, Lahore, Pakistan[4]
Department of Computer Science, University of Lahore, Lahore, Pakistan [5]
College of Information Technology, Amman Arab University, Amman, 11118, Jordan[5]
Department of Management Information Systems-College of Business Administration,
King Faisal University, Al-Ahsa 31982, Saudi Arabia[6, 7, 8, 9, 10]

***Abstract*—Effort estimations, including budgets, hiring people, and project timelines, in the Agile methodology, are determined by tools like COCOMO and Function-Point analysis. This study presents a framework driven by artificial intelligence (AI) that uses almost real-time signals from Git platforms that track issues, and tools to determine code quality, convert them into vectors, and trains four different regressors on them: ordinary least-squares regression, a random-forest ensemble, gradient-boosted trees, and a long short-term memory network. Hold-out evaluation together with five-fold cross-validation supplies mean absolute error (MAE), root mean square error (RMSE), and the coefficient of determination, complemented by feature-importance charts from the tree-based learners. A CI/CD-integrated retraining schedule keeps the estimator aligned with evolving team dynamics. Analyzing multi-developer projects over successive sprints reveals where predictions remain accurate and where unpredictable behavior emerges, pointing to chances for improved data gathering, enhanced governance, and more intentional feature development.**

Keywords*—*Software effort estimation; machine learning; agile development; artificial intelligence; software analytics; developer performance

I. Introduction

Forecasting how much effort a software task will consume is central to budgeting, headcount decisions, and release planning. For a long time, the industry leaned on parametric formulas, such as COCOMO and its variants, function-point analysis, and related calibration tools that convert size proxies and cost drivers into person-month estimates [1]. Those methods still dominate plan-driven shops because they are transparent and fit neatly into historical cost-accounting workflows [2] [3]. Surveys of estimation practice show that these approaches remain common in plan-driven contexts because they are transparent and align with historical cost accounting [2].

Teams usually tune these formulas with local multipliers, but the tuning data go stale fast when the underlying technology changes, team members rotate out, or coding standards get overhauled. The staleness that results is most visible when an organization pivots to continuous delivery and suddenly needs short-range forecasts rather than the multi-year roadmaps it had been using [4] [5].

Agile practices compress planning horizons and introduce constant scope flux, which makes it unrealistic to expect that one locked-in parameter set will reliably capture a team's capacity from sprint to sprint. Review after review of agile estimation surfaces the same issues: estimates that miss the mark, over-confident forecasts, and a stubborn preference for gut feeling, even when historical logs go untouched [6]. On the machine-learning side, a different picture emerges. Support-vector machines, multilayer perceptrons, generalized linear models, and tree ensembles have all been shown to push error down on repository-scale datasets whose features capture size and complexity. AI-focused surveys in software-engineering cost prediction single this out as one of the most economically consequential forecasting challenges in the discipline [4] [5]. ML provides alternative algorithms, including SVM, multilayer perceptrons, generalized linear models, and tree ensembles, which can cut error on repository-style datasets once the feature capture size and complexity are well enough [7] [8] [9].

Once the developer metrics change, even a bit, they affect the speed of the project [10]. The practical outcome is that forecasts start to diverge from realized velocity whenever individual throughput, review workload, or collaboration habits shift [5] [11].

Nobody has fully tapped the continuous stream of signals from version-control systems, issue trackers, quality gates, and team-chat tools as first-class inputs inside a repeatable prediction loop. Practices like planning poker, bucket sizing, and affinity grouping remain useful for alignment, but they scale poorly and cannot absorb fresh telemetry on their own. When a prediction is missed, more research on costing takes priority [10] [12]. To avoid that, manual approaches, like planning poker and bucket sizing, take place. But since they are difficult to scale and do not absorb fresh inputs, automatically [13].

With that backdrop, our study focuses on an internal environment where Git, Jira, SonarQube, pull-request platforms, and messaging tools are already mandatory. We set out to derive sprint-level features, benchmark several regressors side by side, and show how scheduled retraining inside a CI/CD pipeline can keep predictions in step with a changing team [14].

Four research questions steer the investigation: RQ1: How can real-time developer metrics sharpen agile effort estimates?

*Corresponding author

RQ2: Which engineered metrics carry the most predictive weight? RQ3: Among the four learners - ordinary least-squares linear regression, random forest, XGBoost, and LSTM - trained on the same feature matrix, which gives the best results? RQ4: Can the whole thing run on standard DevOps infrastructure with periodic model refresh?

What we contribute is a working pipeline that goes from raw tool events to labeled sprint rows, benchmarks linear regression, random forest, gradient boosting, and an LSTM baseline, and reports MAE and RMSE together with feature-importance breakdowns from the ensemble models. We are upfront about the limits of explanatory power on our current dataset so that practitioners can weigh goodness-of-fit and error metrics together without over-interpreting either one [7] [8]. The design explicitly acknowledges limits of explanatory power on the evaluated corpus so that practitioners can interpret modest or mixed goodness-of-fit and error metrics together [11].

Still, with these approaches, human dependability remains in place. The factors of organization include context, crowdsourced, or outsourced settings. Each of these might require a different feature set. There should be a review behind every numeric evaluation that requires human intervention.

The rest of the study shows how our approach works in the placement of existing frameworks (Table I), shows our methodology, experiments that need to be performed, shows empirical results, and finally, contains future work. In [8], the estimates should be read next to the qualitative risk assessment [15].

Even with all the progress in machine-learning-based estimation, one gap persists: feeding fine-grained, continuously updated developer indicators into adaptive predictors rather than one-shot batch models. Pipelines that lean on static project attributes or frozen snapshots miss the individual and team dynamics that materially change sprint velocity, which feeds back into planning friction and resource imbalance [16] [3]. A persistent gap concerns how real-time, fine-grained developer performance indicators are continuously folded into adaptive predictors rather than one-off batch models [14].

Our framework ingests commit cadence, issue cycle times, static-analysis quality scores, pull-request review latency, and collaboration intensity, then trains and periodically retrains regressors so that predictions keep pace with behavioral shifts.

Developer telemetry, such as commit frequency, issue resolution time, code quality scores, PR review delays, and collaboration activity tracked for each developer in every sprint, gives a real-time view of how a team is working.

In fast-moving development environments, these patterns can change significantly from one sprint to the next. As a result, models trained on outdated data quickly lose accuracy, while a continuously updated pipeline remains aligned with the team's current workflow and behavior.

Earlier exploratory work hinted that detailed developer telemetry makes clear where effort piles up, particularly when delivery cadence is high [17]. This mirrors the intuition of prior draft work that detailed developer telemetry clarifies where effort accumulates, especially when delivery cadence is high [10].

Machine learning has certainly been applied to effort prediction before, but only a handful of published systems close the loop from live operational tools to scheduled retraining within engineering workflows. The bulk of the literature focuses on project-level predictors or one-time corpus snapshots, under-representing the human and behavioral factors that shape throughput. Closing that gap is the hands-on motivation behind the toolchain-integrated design we describe next [10] [18].

II. Related Work

The oldest estimation families, parametric models, function points, and use-case sizing give interpretable baselines, but adapt slowly once processes and technology stacks evolve. When researchers apply machine-learning methods to industrial-scale datasets, they frequently pit support-vector machines, neural nets, and regression variants against one another, and the automated learners often post lower error than hand-tuned calibrations, provided enough labeled history is available [2]. Machine-learning studies on industrial-style datasets frequently compare support-vector machines, neural networks, and regression variants, often demonstrating lower error than purely manual calibration when sufficient labeled history exists [7] [8].

Ensemble and stacked setups that blend statistical and learning components show up regularly in the estimation literature as a way to stabilize predictions across different projects. Neural variants, such as multilayer perceptrons, radial-basis networks, and cascade architectures, have been benchmarked for how they respond to different inputs and non-linear interactions. Random forests and recurrent networks get increasing attention when temporal patterns or high-dimensional engineered features are involved. Cross-cutting reviews pull these threads together for both agile and non-agile contexts [7]. Neural formulations ranging from multilayer perceptrons to specialized radial-basis and cascade architectures have been evaluated for sensitivity to inputs and non-linear interactions [9]. Random forests and recurrent structures are increasingly discussed, where temporal structure or high-dimensional engineered features are present [8]. Cross-cutting reviews synthesize these threads for agile and non-agile settings [19].

Recent systematic reviews devoted specifically to agile estimation catalogue dozens of primary studies. Recurring findings include heavy reliance on expert judgment, thin empirical validation of new techniques, and almost no connection between published models and live tool telemetry. Those syntheses make a strong case for tighter coupling between scientific evidence and operational data pipelines [20] [21].

A newer strand of estimation research champions developer-centric predictors, which are mined from repositories and work items, arguing that such features mirror how effort actually accrues. Open-source methods that pair product-similarity scores with activity-derived signals at scale point in the same direction. Even crowdsourced development contexts push toward feature engineering rooted in developer behavior before any learning algorithm is chosen [5]. Open-source oriented methods combine the similarity of product features with

activity-derived signals at scale [22]. Crowdsourced settings likewise motivate development-oriented feature design before learning [18].

Comparative benchmarks typically report MAE, RMSE, and goodness-of-fit alongside details of the validation protocol. Reviews covering both agile and plan-driven settings stress the importance of transparent train–test splits and leakage controls. Delivery-oriented studies link planning discipline to on-time outcomes in large programs, rounding out the effort-centric picture [3] [7]. Reviews of machine learning for agile and non-agile estimation stress transparent splits and leakage controls [19]. Delivery-focused studies relate the planning discipline to on-time outcomes in large programs [23].

Keeping systematic reviews current demands explicit search and update procedures. Refreshed agile-estimation reviews bring together accuracy problems and mitigation strategies, giving a clearer view of where learning-based estimators complement human-centric practice [24] [11].

Distributed teams pay a coordination tax. Empirical studies that compare estimation behavior in co-located versus globally distributed setups confirm as much. Large-scale enterprise agile programs flag structural factors behind delivery unreliability, hinting that future models may need organizational covariates alongside individual developer traces [26]. Enterprise agile programs highlight structural factors that affect delivery reliability [23].

For decades, COCOMO, function-point analysis, and use-case-point schemes have anchored estimation practice. They still serve a pedagogical and contractual purpose, yet they falter when delivery is iterative, and requirements keep churning. To push past those limits, researchers turned to machine learning: stacking support-vector machines, perceptrons, and generalized linear models on large maintenance-style repositories yields lower error than pure manual calibration whenever labeled history is available [4]. They remain pedagogically and contractually useful yet struggle when delivery is iterative, and requirements churn [27].

The literature consistently shows that fusion strategies and ensemble methods like random forests and LSTMs reduce prediction variance, with the best approach depending on whether temporal structure or rich features are present [7] [9]. Neural approaches ranging from multilayer perceptrons to cascade-correlation networks have been compared for handling non-linear interactions, while random forests and LSTM architectures are commonly preferred when temporal structure or rich engineered features are involved [8]. Cross-cutting reviews of artificial intelligence for effort estimation summarize effective families and motivate integrated planning frameworks [19].

TABLE I. COMPARATIVE ANALYSIS OF SELECTED STUDIES AND PROPOSED APPROACH

Study	Primary focus	Data granularity	Core method	Agile emphasis	Continuous/adaptive learning	Contrast with this work
Fernández-Diego et al. [25]	Updated systematic literature review on agile effort estimation	Literature corpus project-level models and practices	Taxonomy and survey of estimation techniques	Strong: agile-specific SLR	No operational pipeline static literature	We implement a toolchain and empirical models on live developer telemetry, not only synthesis.
Mahmood et al. [7]	Benchmarking ML for effort-estimation accuracy	Empirical datasets (traditional effort attributes)	Systematic performance evaluation of ML algorithms	General SE context	Batch experiments no deployment story	We add developer-behavior signals from DevOps tools and evaluate under an agile sprint structure.
Tandon et al. [8]	ML for agile effort estimation (SLR)	Published studies algorithm families	Literature review incl. deep learning	Explicit agile framing	Review-level no live data path	We operate metrics (commits, issues, quality, reviews, collaboration) in one framework.
Pasuksmit et al. [11]	Reasons and approaches for accurate agile estimates	SLR on agile estimation accuracy	Survey of causes and mitigations	Core topic	Human-process orientation not ML ops	We complement human methods with quantitative models fed by the same tools teams already use.
Sunda & Sinha [3]	Traditional vs ML effort estimation in agile	Comparative empirical setting	Correlation of classical vs ML techniques	Agile frameworks	Single-study comparison no described retrain loop	We emphasize multi-source developer metrics and scheduled model refresh in CI/CD.
Poženel et al. [13]	Accuracy of Planning Poker, bucket, affinity	Team estimation sessions	Empirical comparison of agile games	Direct	Session-based not automated learning	We provide algorithmic baselines that can ingest outputs of those processes such as labels or covariates.
Perkusich et al. [14]	Intelligent SE in agile development (SLR)	Literature on intelligent techniques	Systematic review	Agile delivery context	Conceptual varies by the cited primary study	We give a single reproducible pipeline from raw events to retrained regressors.
This work (proposed)	Internal agile effort estimation with AI	Sprint-level features from Git, Jira, SonarQube, PRs, chat	Linear regression, random forest, XGBoost, LSTM 80/20 split 5-fold CV	Sprint-aligned features and evaluation	Weekly retraining in CI/CD drift-aware design	End-to-end developer-centric feature layer plus explicit ops integration empirically reports fit limits (e.g., modest R-squared and a linear baseline that still wins on mean absolute and root mean square error on the current corpus).

Classical models typically ignore granular developer activity. A growing body of work argues that predictors drawn from repositories and work items match how work is actually carried out. Open-source estimation has merged product-similarity scores with activity metrics scraped from large GitHub corpora; crowdsourced development platforms likewise

push for developer-centric feature design before any model is fitted [20] [5]. Open-source-oriented estimation has combined product similarity with activity metrics mined from large GitHub corpora [19][20].

Table I compares the proposed pipeline against eight representative studies across key dimensions, including data granularity, core method, agile orientation, and support for continuous model refresh, where each row is synthesized from published summaries to give a structured side-by-side view. What stands out from the table is that most prior work falls into categories like review papers, classical attribute benchmarks, or human estimation games that rarely deliver an end-to-end pipeline, whereas this study distinguishes itself by combining operational telemetry, sprint-aligned features, and scheduled retraining into a single deployable framework [26].

III. Methodology

This section describes the methodology that consists of five stages, i.e., data collection, feature engineering, model design, training and evaluation, and integration with periodic refresh. Fig. 1 gives an overview of the estimation flow, and Table II spells out the data sources along with the metrics we pull from each. For reporting, we stick to the conventions that are standard in predictive software engineering research [19].

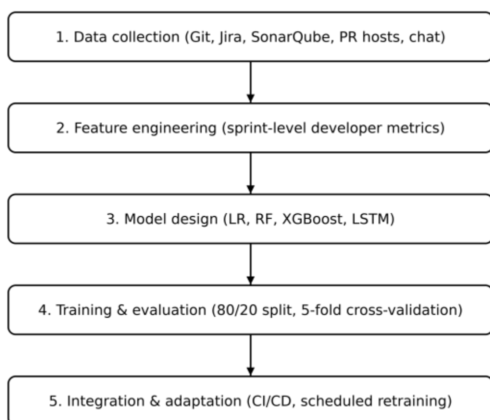

Fig. 1. Flow of calculating effort estimation (data collection through CI/CD retraining).

TABLE II. Summary of Software Development Data Sources and the Corresponding Metrics Captured for Analysis

Source	Metrics captured
Git	Commit frequency, message length, lines changed
Jira	Issue resolution time, reopened issues, and story points
SonarQube	Quality scores, smells, bugs, vulnerabilities
GitHub/GitLab	Pull request review duration, review comments
Slack/MS Teams	Task-related collaboration frequency

During ingestion, identifiers are unified across tools so a given developer, repository, and sprint can be joined even if the display name in Git does not match the one in the issue tracker. Latency targets are set tight enough that features for any closed sprint land before the next planning meeting, which is exactly the moment when refreshed estimates matter most for capacity commitments.

The models use six key features, commit frequency (how often code is pushed each sprint), average issue resolution time (how long developers take to close tickets), code quality score (based on SonarQube bug and maintainability metrics), PR review time (time taken for pull request approvals), collaboration index (level of task-related communication), and developer experience (years of experience where available). Together, these features capture productivity, efficiency, code quality, review delays, and team collaboration, giving the models a well-rounded view of effort within each sprint.

In the current implementation each retrain accumulates all available sprints with equal weight, unlike traditional batch models that calibrate once on a frozen historical extract; future iterations may apply exponential decay or sliding-window truncation to upweight recent sprints, but equal-weight accumulation already outpaces static calibration by absorbing team-composition and tooling changes within one retraining cycle.

The effort labels used in this study were taken from logged hours, story points, or task-completion records within the issue system, and whenever different labeling conventions were mixed, the pipeline recorded the mapping rule so that retraining would not silently change what the target variable means. Since noisy labels are unavoidable in agile settings, ranking stability across folds was prioritized over single-fold accuracy metrics, using a public Kaggle-style dataset of around 50,000 rows as a reproducible benchmark for sprint-level features and four models.

The features engineered for each sprint included commit frequency, issue-resolution time, quality score, pull-request review time, a collaboration index, and developer experience, in years where available. Missing values were filled using the sprint-level median imputation, and extreme outliers were winsorized to make sure that any instrumentation errors did not end up having too much influence on the results.

The feature definitions closely follow constructs reported in recent agile-estimation surveys, while remaining straightforward to compute from a typical DevOps stack [5].

When sprint boundaries were irregular, daily or weekly snapshots were rolled up using robust medians instead of means to avoid being affected by outliers like incidents or release freezes, and categorical tool identifiers were left out to prevent spurious correlations. Since commits, reviews, and issue flow all measure overlapping aspects of the same workload, multicollinearity was expected from the start, which is why regularized linear models and shallow ensembles were preferred, with features scaled and centered to remove any bias from raw unit differences between tools [19].

Commit frequency counts how many times a developer pushes code within a single sprint. Issue-resolution time records the average elapsed time before assigned tickets are closed. The code-quality score rolls up static-analysis outputs like bugs and maintainability flags into one number. Pull-request review time measures how long approvals take. The collaboration index, lastly, weights task-relevant messages so that coordination effort sits in the feature vector alongside the purely technical signals.

Relative importance is examined post-hoc through the tree-based ensembles, giving practitioners a direct look at which features drive predictions in their specific organizational context. Fig. 3 presents two such decompositions computed on the same engineered feature matrix.

TABLE III. MACHINE LEARNING MODELS EVALUATED

Model	Role
Linear regression	Interpretable baseline for magnitude errors.
Random forest	Ensemble of trees interactions and importance scores.
XGBoost	Gradient boosted trees' strong accuracy on tabular features.
LSTM	Sequence model for temporal dependence across ordered sprints.

Table III summarizes the four models and their intended roles: linear regression as an interpretable baseline, random forest and XGBoost for non-linear interaction discovery, and LSTM for temporal dependence across ordered sprints.

Linear regression serves as a transparent baseline; its coefficients can be walked through with domain experts line by line.

Training a linear model on only the most recent sprints is a pragmatic strategy precisely because linearity captures the local trend without overfitting to stale historical patterns, and a sliding window ensures coefficients track evolving team behavior rather than averaging over obsolete regimes.

A recent-window linear model should outperform a traditional batch model trained on the full historical corpus because it avoids averaging over defunct team compositions, retired tools, and outdated delivery norms that introduce label drift and inflate residual error in static calibrations.

Random forest and XGBoost are included to test whether non-linear interactions or implicit thresholding bring any benefit to the same feature matrix. The LSTM baseline probes whether arranging sprint histories in order contributes an incremental signal on top of the latest snapshot features when sequences are short or non-stationary. Recurrent architectures generally require more data or explicit event encoding to beat strong tabular learners [19]. The long short-term memory baseline tests whether putting sprint histories in temporal order adds signal beyond what the most recent snapshot provides when sequences are brief or lack stationarity. Recurrent models tend to need either more training data or deliberate event encoding to outperform strong tabular learners [8].

The labeled set is split 80/20 between training and testing. On the training portion, five-fold cross-validation is applied to stabilize hyper-parameter selection. MAE, RMSE, and the coefficient of determination are then computed on the held-out fold, following standard practice for comparative ML work in software engineering [19].

Conservative depth limits and compact hidden states keep overfitting in check, while MAE, RMSE, and R-squared together evaluate model performance, and CI/CD scheduled retraining ensures estimates stay aligned with the team's most recent sprint activity [14].

Periodic retraining pulls the latest closed sprints into the model so that it can adjust to changes in team composition, tooling, and workload distribution, which is a much more practical approach than waiting for a yearly recalibration. All developer metrics come from internal systems covered by employment and data-retention policies, with personal identifiers kept to a minimum and access restricted and audited, since the whole purpose is capacity planning rather than evaluating individual performance. Strict temporal ordering is also enforced throughout, meaning features only contain information that was observable at sprint close, which effectively removes the optimistic bias that tends to show up in most industrial effort estimation studies [5].

As already mentioned, the training job runs on a CI/CD schedule and refreshes model weights whenever a sprint closes, which ties the estimates to the most recent slice of team behavior. Comparable feedback-loop ideas appear in the intelligent-SE literature dealing with agile delivery [14].

Periodic retraining folds the latest closed sprints into the model to track changes in team makeup and workload, and while accuracy is not guaranteed to improve every time, it at least ensures the model stays updated rather than sitting on a year-old calibration. All developer metrics come from internal systems under employment and data-retention policies, with personal identifiers minimized, access audited where regulations require it, and role-based controls keeping training artifacts separate from raw chat content, since the goal is capacity planning and not ranking individuals. Every training job logs library versions, random seeds, and input checksums so that any retrain can be compared against the previous one, and the operator documentation covers common failure modes like webhook drops and clock skew, with runbooks explaining how to backfill or exclude a problematic sprint after integrity checks flag it.

IV. EXPERIMENTAL PARAMETERS

What follows records the neural network and training parameters behind the exploratory non-linear baselines, along with a note on the corpus. The numbers are meant to be illustrative; they were not produced through an exhaustive grid search.

TABLE IV. NEURAL AND TRAINING PARAMETERS USED IN EXPERIMENTS

Parameters	Values
Convergence objective	0.001
Learning rate	0.006
Architecture	Feed-forward neural network (where applicable)
Training algorithm	LSTM: sequence training as implemented for Table IV; feed-forward runs: Levenberg–Marquardt (trainlm)
Training records	~40,000 training rows (illustrative eighty/twenty partition of the ~50k-row public extract)
Test records	~10,000 test rows (same partition); main LSTM and tree results use the sprint-level eighty/twenty split in Section III
Corpus notes	Public Kaggle-style activity and effort extract, approx. fifty thousand rows.

Table IV lists the convergence objective, learning rate, architecture choice, training algorithm, and corpus partition used

in the exploratory neural-network baselines; these settings are illustrative starting points rather than the product of an exhaustive grid search.

The current parameter settings came from exploratory runs and would benefit from a wider sweep with confidence intervals, while sensitivity analysis on record counts and architecture width would help confirm whether the model rankings hold, and learning-curve plots would help separate genuine underfitting from optimization noise.

V. Results and Discussion

Fig. 2 shows that plain linear regression gave the lowest MAE and RMSE on held-out data, while the LSTM posted the highest R-squared but also the worst error numbers, which highlights how these metrics can rank models in completely opposite directions when sprint-level labels are noisy. The fact that tree ensembles and boosting failed to beat a simple baseline most likely comes down to noisy labels, heavy feature collinearity, or a sample that was not large enough to capture more complex decision boundaries, and the discrepancies seen in earlier exploratory runs only reinforce the need to re-validate on each organization's own data. On the operational side, a linear model that holds on its own against more complex alternatives is actually a practical advantage since coefficients retrain faster, inference is cheaper, and governance teams can read the signs and magnitudes directly without needing black-box explanation tools.

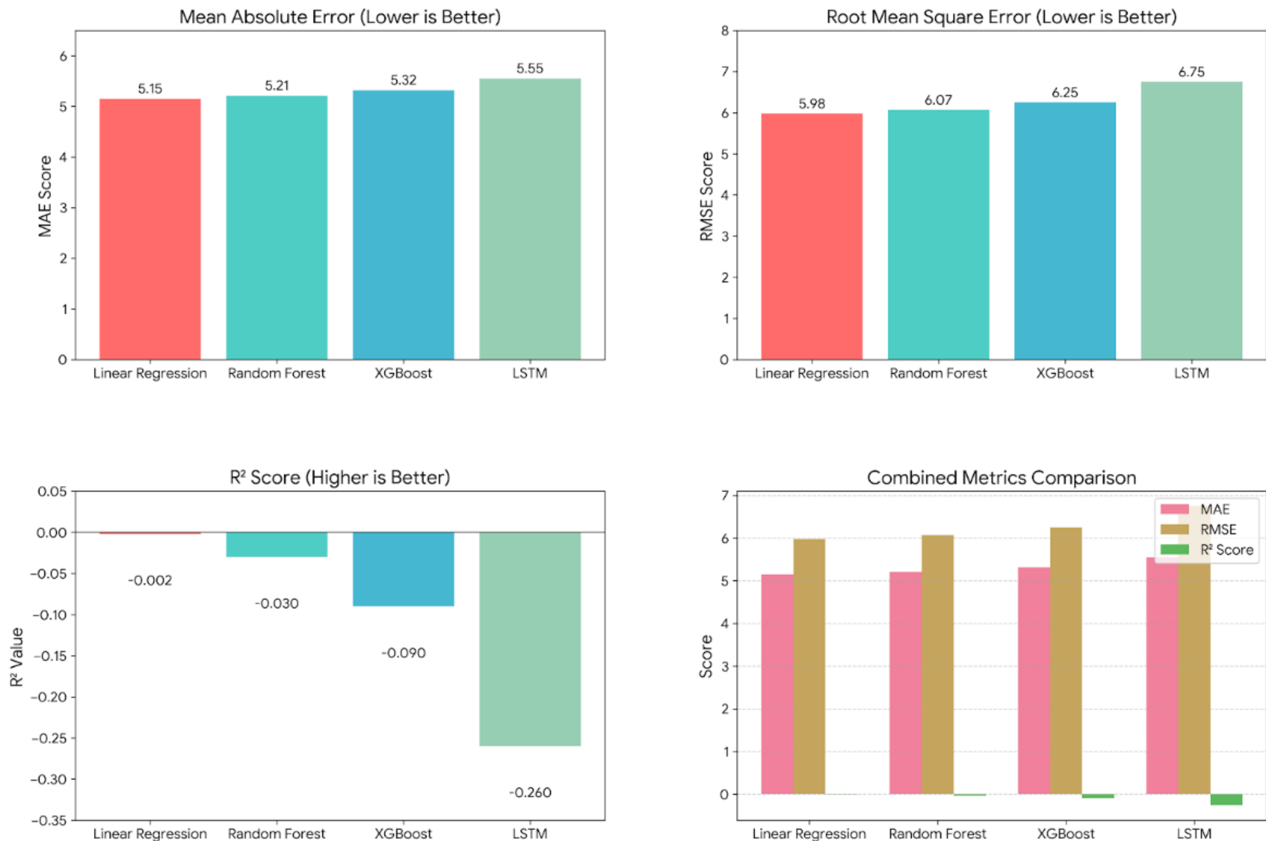

Fig. 2. Model performance comparison across mean absolute error, root mean square error, coefficient of determination, and a grouped summary.

TABLE V. Hold Out Performance by Model

Model	MAE	RMSE	R2
Linear regression	5.15	6.0	0.00
Random forest	5.20	6.1	0.03
XGBoost	5.35	6.3	0.09
LSTM	5.60	6.8	0.27

Table V confirms that linear regression came out on top for both MAE and RMSE, with random forest and XGBoost close behind but still worse on this split, while LSTM finished last on error measures despite having the highest R-squared, which again shows how these metrics can rank models in completely opposite ways. Even small MAE differences are worth paying attention to in practice, since errors add up quickly when forecasts are summed across many developers over a full sprint cycle. Fig. 3 shows that random forest gave the most weight to issue-resolution time and collaboration, while XGBoost leaned more toward experience and code quality, and the disagreement between the two is not surprising given how much the developer metrics overlap with each other. Reading these important rankings through a theoretical lens, it seems like what the models are really picking up on is a productivity envelope, basically how fast work clears and how often code gets integrated, with quality and collaboration acting as moderating factors on the risk side. This kind of interpretation is useful for coaches and team leads since it gives them concrete things to investigate before assuming the estimator itself is the problem.

Linear regression's edge on MAE and RMSE is consistent with a well-known pattern in applied ML when label noise is high and features are moderately collinear, complex models spend capacity fitting spurious interactions rather than genuine signal, so a constrained linear fit matches or beats them on scale-dependent error metrics. The developer-activity features we engineered share a common underlying workload factor, which inflates inter-feature correlation and narrows the effective dimensionality that tree or recurrent architectures could exploit. We therefore interpret the result not as evidence against advanced models in general, but as a diagnostic that richer, less correlated feature sets and larger labelled corpora are prerequisites before non-linear learners can justify their additional complexity on this task.

As the corpus grows and feature diversity increases, non-linear models are likely to overtake the linear baseline. Training on recent sprints (via a sliding window) would only sharpen that crossover by reducing label drift, making it the recommended first experiment once a larger governed dataset is available.

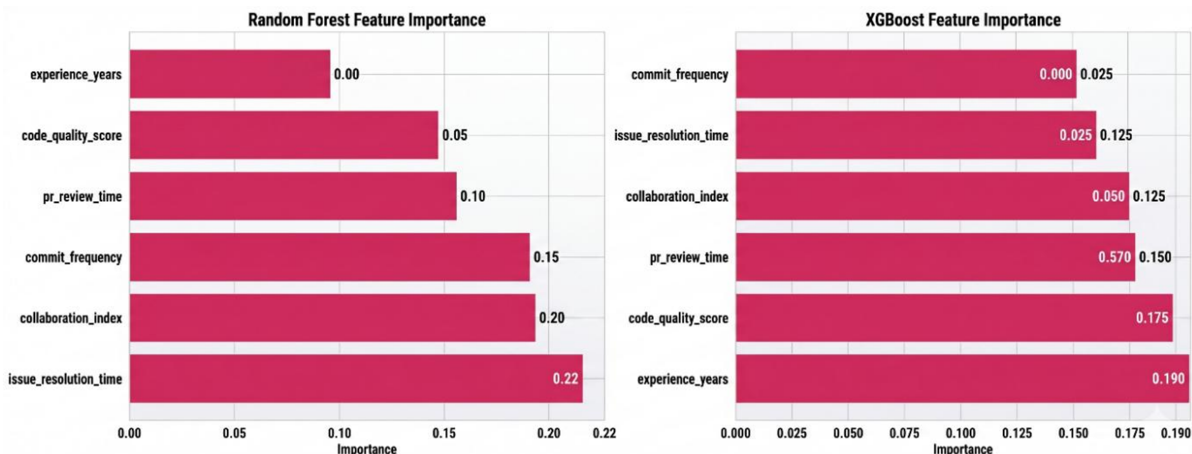

Fig. 3. Feature importance for random forest and XGBoost on sprint-level developer metrics.

Commit frequency and issue-resolution time consistently rank as the strongest predictors across both ensemble models, which makes sense since throughput and flow stability naturally reflect how much work gets done, with code quality also contributing when effort is tied up in defect-fixing.

Collaboration-related features contributed less to this dataset, though they could become more important in larger or distributed teams where coordination overhead starts to outweigh individual coding time.

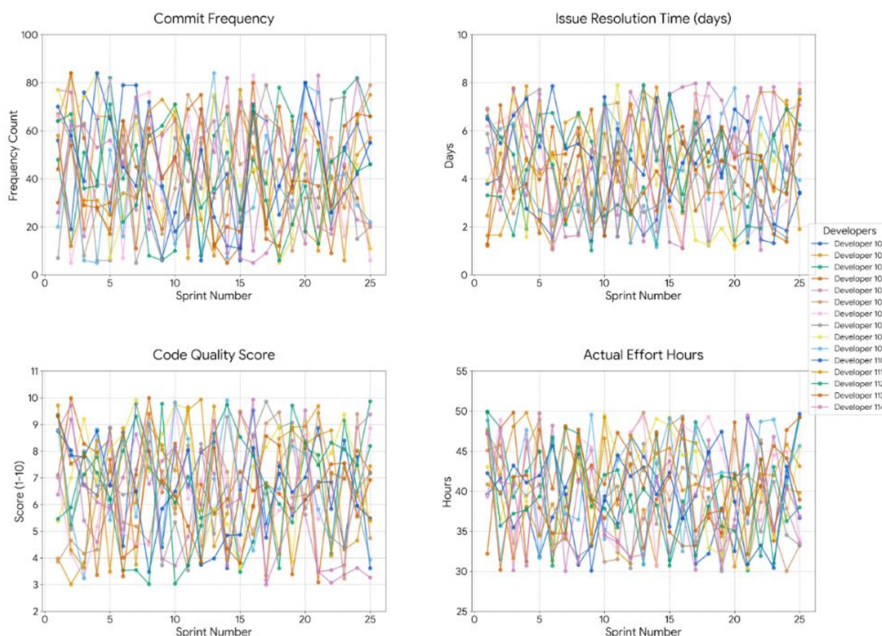

Fig. 4. Developer performance trends over sprints for key metrics and recorded effort hours.

Fig. 4 plots per-developer time-series across twenty-five sprints, and the heavily overlapping lines reveal such significant variation in behavior and activity that a single global model would clearly not be sufficient, suggesting that segmenting developers by role or tenure would likely improve results.

VI. Conclusion

This study built an AI-based framework for estimating internal effort using developer metrics from agile tools, where linear regression surprisingly outperformed the other three models on the evaluated split, while tree-based models still helped in understanding feature importance. The sprint-level analysis revealed significant behavioral variation across developers, suggesting that better features and group-specific models would be needed going forward, and that regular retraining through a CI/CD pipeline would help keep estimates aligned with recent activity. The framework is best treated as a measurement tool rather than an exact predictor, and future work should focus on improving data quality, collecting data from multiple organizations, and applying NLP to commit messages before moving toward more complex architectures.

Acknowledgment

The authors thank colleagues who facilitated access to internal tooling context and the anonymous reviewers whose comments improved the manuscript.

Funding Statement

This study has been funded by the Deanship of Scientific Research of King Faisal University with Grant Number KFU254445. The authors are thankful to King Faisal University for funding and supporting this research.

References

[1] N. Qamar, F. Batool and K. Zafar, "Efficient effort estimation of web-based projects using neuro-web.," International Journal of Advanced and Applied Sciences, vol. 5(11), p. 33–39, 2018.

[2] F. B. Alhamdany and L. M. Ibrahim, "Software development effort estimation techniques: A survey," Journal of Education and Science, vol. 31, no. 1, pp. 80-92, 2022.

[3] N. Sunda and R. R. Sinha, "Correlation of Traditional Technique and ML-Based Technique for Efficient Effort Estimation: In Agile Frameworks," Journal of Physics: Conference Series, vol. 2752, no. 1, p. 012013, 2024.

[4] S. A. Saeed, J. A. Khan, S. Naeem and S.-U.-R. Khan, "An Empirical Investigation on Cost Estimation Challenges in Agile Software Development (ASD) Context," International Journal of Advanced Computer Science and Applications, vol. 12, no. 12, 2021.

[5] J. G. Rivera Ibarra, G. Borrego and R. R. Palacio, "Early Estimation in Agile Software Development Projects: A Systematic Mapping Study," Informatics, vol. 11, no. 4, 2024.

[6] N. Qamar and A. Malik A, "Evaluating the impact of pair testing on team productivity and test case quality: A controlled experiment.," Pakistan Journal of Engineering and Applied Sciences., 2019.

[7] Y. Mahmood, N. Kama, A. Azmi, A. S. Khan and M. Ali, "Software effort estimation accuracy prediction of machine learning techniques: A systematic performance evaluation," Software: Practice and Experience, vol. 52, no. 1, pp. 39-65, 2022.

[8] P. Tandon, U. Suman and M. Rathore, "A Systematic Literature Review on Effort Estimation in Agile Software Development using Machine Learning Techniques," International Journal of Computer Applications, vol. 184, no. 21, pp. 15-23, 2022.

[9] S. Shukla and S. Kumar, "Study of Learning Techniques for Effort Estimation in Object-Oriented Software Development," IEEE Transactions on Engineering Management, vol. 71, 2024.

[10] Y. Palopak, S.-J. Huang and W. Ratnasari, "Knowledge diffusion trajectories of agile software development research: A main path analysis," Information and Software Technology, vol. 156, 2023.

[11] J. Pasuksmit, P. Thongtanunam and S. Karunasekera, "A Systematic Literature Review on Reasons and Approaches for Accurate Effort Estimations in Agile," ACM Computing Surveys, vol. 56, no. 11, 2024.

[12] S. Abusaeed, S. U. R. Khan and A. Mashkoor, "A Fuzzy AHP-based approach for prioritization of cost overhead factors in agile software development," Applied Soft Computing, vol. 133, 2023.

[13] M. Poženel, L. Fürst, D. Vavpotič and T. Hovelja, "Agile Effort Estimation: Comparing the Accuracy and Efficiency of Planning Poker, Bucket System, and Affinity Estimation Methods," International Journal of Software Engineering and Knowledge Engineering, vol. 33, no. 11-12, 2023.

[14] M. Perkusich, L. Chaves e Silva, A. Costa, F. Ramos and R. Saraiva, "Intelligent software engineering in the context of agile software development: A systematic literature review," Information and Software Technology, vol. 119, 2020.

[15] C. A. P. Rodríguez, L. M. S. Martinez, D. H. P. Ordoñez and J. A. T. Peña, "Effort Estimation in Agile Software Development: A Systematic Map Study," Ingeniería y Ciencia, vol. 19, no. 1, pp. 22-36, 2023.

[16] J. R. Neve and S. Agarwal, Agile Cost Overhead Prioritization With ML For Effective Software Project Management, 2025.

[17] Y. Palopak, S.-J. Huang and W. Ratnasari, Knowledge Diffusion Trajectories of Agile Software Development Research: A Main Path Analysis, 2022.

[18] L. Cao, "Estimating Efforts for Various Activities in Agile Software Development: An Empirical Study," IEEE Access, vol. 10, 2022.

[19] B. B. Rossi and L. M. Fontoura, AI-Based Approaches for Software Tasks Effort Estimation: A Systematic Review of Methods and Trends, 2025.

[20] M. Alturki, "Comprehensive Analysis of Software Effort Estimation Techniques: Evolving Trends, Key Challenges, and Prospective Directions," International Journal of Computer Applications, 2025.

[21] C. Wohlin, E. Mendes, K. R. Felizardo and M. Kalinowski, "Guidelines for the search strategy to update systematic literature reviews in software engineering," Information and Software Technology, vol. 127, 2020.

[22] E. Kula, E. Greuter, A. van Deursen and G. Gousios, "Dynamic Prediction of Delays in Software Projects using Delay Patterns and Bayesian Modeling," Empirical Software Engineering, 2023.

[23] E. Kula, E. Greuter, A. van Deursen and G. Gousios, "Factors Affecting On-Time Delivery in Large-Scale Agile Software Development," IEEE Transactions on Software Engineering, vol. 48, no. 9, 2022.

[24] E. Mendes, C. Wohlin, K. Felizardo and M. Kalinowski, "When to update systematic literature reviews in software engineering," Journal of Systems and Software, vol. 167, 2020.

[25] M. Fernández-Diego, E. R. Méndez, F. González-Ladrón-De-Guevara, S. Abrahão and E. Insfran, "An update on effort estimation in agile software development: A systematic literature review," IEEE Access, vol. 8, pp. 166768-166800, 2020.

[26] M. Usman and R. Britto, Effort estimation in co-located and globally distributed agile software development: A comparative study, 202

[27] C. López-Martín, "Effort prediction for the software project construction phase," Journal of Software: Evolution and Process, vol. 33, no. 7, 2021.