DOI: https://doi.org/10.15276/hait.09.2026.36
UDC 004.41:004.85

A process-grounded framework for submission-time prediction of code review outcomes

Oleksandr S. Aratovskyi[1)]
ORCID: https://orcid.org/0009-0006-0854-8582; alex.aratovskiy@stud.op.edu.ua. Scopus Author ID: 60503533100
Vira V. Liubchenko[1), 2)]
ORCID: https://orcid.org/0000-0002-4611-7832; lvv@op.edu.ua. Scopus Author ID: 56667638800
[1)] Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine
[2)] University of Applied Sciences, Ulmenliet 20, Hamburg, 21033, Germany

ABSTRACT

Relevance. Code review is a central quality assurance practice, yet its outcomes are hard to anticipate at submission: duration varies widely, the number of revision cycles is unknown, and many pull requests quietly lose reviewer attention. Existing approaches address these outcomes in isolation and are predominantly evaluated within a single repository or organization, limiting evidence of cross-project generalization. The **purpose** of the paper is to establish an explicit model of the code review process and to show that characteristics derived from this model are predictable at submission time by models capable of generalizing across projects. **Objectives**: to formulate a process model treating automated participants as actors; to derive measurable characteristics of review from its structural elements; to construct a multi-repository dataset; and to validate the predictability of two characteristics under within-project and cross-project evaluation. **Methods**. The model combines Business Process Model and Notation with a Unified Modeling Language state machine. Gradient boosting classifiers use submission-time attributes of the change, author, and project, computed in an expanding temporal window that prevents data leakage. The empirical study covers 64,820 merged pull requests from 12 GitHub repositories representing six primary programming languages. The evaluation uses stratified cross-validation and leave-one-repository-out validation. **Scientific novelty**. Prediction targets follow from structural elements of the process model – a state sojourn, a loop, and a timer – rather than pragmatic choice. We treat continuous integration and a large language model assistant as process actors. **Results**. Iteration prediction generalizes to repositories unseen in training; staleness prediction transfers less reliably, indicating stronger dependence on project context. **Practical significance**. The proposed models support submission-time risk assessment, review prioritization, early identification of changes likely to require rework, and escalation of pull requests at risk of becoming stale. The findings suggest that iteration prediction can be deployed as a pre-trained cross-project model, whereas staleness prediction benefits from project-specific calibration. **Conclusions**. Characteristics derived from an explicit process model are predictable at submission time, and a substantial part of this predictability carries across projects. The framework provides a foundation for data-driven review prioritization and future integration of AI-assisted review workflows while preserving human control over review and merge decisions.

Keywords: software engineering; code review; pull request; process model; business process modeling; state machine; software repository mining; machine learning; gradient boosting; cross-project prediction

For citation: **Aratovskyi O. S., Liubchenko V. V. "A process-grounded framework for submission-time prediction of code review outcomes". *Herald of Advanced Information Technology*. 2026; Vol.9 No.4: 588–604. DOI: https://doi.org/10.15276/hait.09.2026.36**

INTRODUCTION

Code review is one of the most widely adopted quality assurance practices in contemporary software engineering. In its modern, lightweight, tool-based form, it does more than detect defects; it enables knowledge transfer, shared code ownership, and enforcement of team conventions [1], [2]. Surveys of the field consistently show that nearly all large software organizations and most active open-source projects gate changes through some form of peer review [3]. On collaborative platforms such as GitHub, review is organized around the pull request (PR): a proposed change that must be inspected, possibly revised, and explicitly approved before it is merged into the main branch [4].

Despite its ubiquity, the review process remains operationally unpredictable. The time from PR submission to first reviewer reaction and to final merge varies by orders of magnitude both across and within projects [5]. The number of revision iterations (cycles of reviewer feedback followed by author rework) is unknown when the PR is opened, although it largely determines the total cost of the review. A non-negligible share of PRs silently loses reviewer attention and becomes stale: the change remains open without activity, its potential defects are never examined, and the contributor's effort is wasted. Project maintainers currently lack systematic tools to anticipate these outcomes when a PR is submitted.

The research community has responded with predictive models for individual review outcomes: completion time prediction in large cloud-service

© Aratovskyi O., Liubchenko V., 2026

This is an open access article under the CC BY license (https://creativecommons.org/licenses/by/4.0/deed.uk)

organizations [6], nudging services for overdue PRs [7], and models of review completion time in open-source projects [8]. These results are encouraging, but they share two limitations. First, almost all published models are trained and evaluated within a single repository or a single organization, so it is unclear whether they capture transferable regularities of the review process or merely project-specific conventions. Second, the predicted quantities (time, iterations, and abandonment) are chosen ad hoc, without an explicit model of the underlying process to justify why these characteristics matter and how they relate to one another. The second limitation has become more pressing as the process itself has changed: continuous integration (CI) pipelines now serve as automated gatekeepers, and artificial intelligence (AI) assistants based on large language models increasingly participate in the review process by generating comments and even entire changes [9], [10]. A process description that ignores these actors no longer reflects how review actually happens.

Our previous research addressed a related methodological problem of supporting decision-making through quantitative assessment of data quality [11]. We proposed information-based quality measures and visualization techniques for evaluating survey data quality. While that work focused on survey data, it demonstrated how measurable process characteristics can support analytical decision-making. The present study extends this general methodological approach to software engineering by modeling the code review process and predicting its key characteristics at submission time.

This paper addresses both limitations through an empirical framework that couples a formal model of the code review process with cross-project predictive modeling of its measurable characteristics.

This **paper aims** to present an explicit model of the code review process and shows that two of its three derived characteristics can be predicted at submission time using models that generalize across projects. The proposed framework supports a broader research agenda focused on improving review efficiency by reducing review time, iteration costs, and the silent abandonment of code changes. This paper delivers the predictive foundation for that purpose rather than a measured efficiency gain.

To achieve this aim, four objectives are addressed:

– to formulate a model of the code review process;

– to derive systematically, from the model, the measurable characteristics of the review process worth predicting at submission time;

– to construct a multi-repository dataset;

– to validate the predictability of two derived characteristics (iterations and staleness) empirically using gradient boosting classifiers.

The rest of the paper is organized as follows. Section 2 surveys the literature on code review process models, prediction of review characteristics, and cross-project generalization, and defines the research problem. Section 3 presents the materials and methods: the process model with the derivation of the target characteristics, the construction of the cross-project dataset, and the predictive modeling methodology. Section 4 reports the results. Section 5 discusses cross-project generalization, threats to validity, and directions for future work. The last section concludes.

RELATED WORKS

Software process modeling has a long tradition in software engineering [12], and researchers have developed a broad spectrum of notations to describe development workflows, ranging from informal diagrams to executable process languages [13]. BPMN 2.0 is the de facto standard for business process description and offers constructs (lanes, tasks, gateways, and events) that map naturally onto collaborative engineering workflows; UML state machines complement it by describing the lifecycle of individual artifacts.

The code review process itself, however, has mostly been described informally. Rigby et al. [14] documented the structure and parameters of contemporary peer review across many open-source projects, characterizing it as lightweight and iterative, but they did so through empirical observation rather than a formal model. Subsequent empirical work refined the picture of individual stages: reviewer participation and its determinants [15], the influence of non-technical factors on review outcomes [16], and the expectations of developers at Google [2] and Microsoft [1]. The comprehensive survey by Badampudi et al. [3] catalogs more than a hundred studies on modern code review and notes that research attention focuses on isolated aspects (reviewer recommendations, comment quality, outcome prediction), while integrated descriptions of the process as a whole remain rare.

Two recent developments make the absence of an explicit process model increasingly costly. First, CI services have become a standard gate in PR-

based workflows: automated checks run before or in parallel with human review, and their verdicts condition reviewer behavior [4]. Second, AI assistance has entered the loop. LLM-based tools generate review comments [9], industrial deployments automatically resolve a measurable share of human review comments [10], and bot-authored PRs (dependency updaters, AI coding agents) form a growing fraction of incoming changes. Bots in code review have been characterized empirically [17]; in the process descriptions cited above, and across the studies cataloged by the systematic review of Badampudi et al. [3], these automated participants are nevertheless treated as context rather than as actors of the process. The model proposed in the following section fills this gap.

A substantial body of work predicts individual outcomes of the review process. Maddila et al. [6] predicted pull request completion time in large-scale cloud services at Microsoft using gradient-boosted trees on change, author, and process features; the follow-up Nudge system [7] used these predictions to send timely reminders for overdue PRs and was deployed across thousands of repositories within a single organization. Chouchen et al. [8] modeled code review completion time in open-source projects as a multi-stage problem. First-response latency, an important component of overall duration, has been studied and predicted with tree-ensemble classifiers on 20 large GitHub projects [18]. Fan et al. [19] predicted, at an early stage, whether a code change would eventually be merged and thereby prioritizing review tasks. The number of review rounds, the characteristic closest to the first target of this paper, has been modeled by Huang et al. [20] on three Gerrit projects, and evaluated across projects: under their union-training protocol, the F1 score falls to 0.27-0.35, from which they conclude that transfer between projects is difficult. Gerrit organizes changes around patch sets rather than around a pull request, so their setting is related to but not identical to PR-based review. PR abandonment, the closest analog to what we call staleness, has been studied empirically [21], although mainly through retrospective analysis of why contributors abandon changes rather than through submission-time prediction.

Three patterns recur across this body of work. Each study predicts a single characteristic, and the choice is justified pragmatically rather than derived from a model of the process. Validation is predominantly within-project, although several studies [18], [19], [20] do report cross-project results; of these, only Khatoonabadi et al. [18] hold out whole projects across a sample larger than a handful. Feature sets also overlap heavily across studies (e.g., change size, author history, project activity), which suggests that a shared, process-grounded feature taxonomy is feasible.

The question of whether predictive models transfer across projects has been studied most thoroughly in the context of defect prediction. Zimmermann et al. [22] found that only a small fraction of cross-project defect prediction attempts succeeded, and a subsequent systematic review [23] confirmed that cross-project transfer remains difficult and strongly depends on feature design and data selection. For review characteristic prediction, the gap is wider still: the works cited above either do not evaluate cross-project performance at all or restrict themselves to a handful of projects. Maddila et al. [6], [7] operate across many repositories but within a single company, where tooling and process conventions are homogeneous. Among the studies cataloged by the systematic review of Badampudi et al. [3] and those surveyed above, we found no submission-time prediction of review iterations and staleness evaluated under a leave-one-repo-out protocol on PR-based review. Khatoonabadi et al. [18] apply exactly this protocol to twenty GitHub projects, but for a different characteristic, the latency of the first response; our contribution is therefore the targets and their derivation from an explicit process model, not the scale of the sample. Closing this gap is the empirical core of this paper.

This review reveals two gaps: existing descriptions of the review process do not account for its automated actors (continuous integration and AI assistance), and submission-time predictions of review characteristics have not been validated across projects. The research problem addressed in this paper is therefore to establish an explicit, automation-aware model of the review process and to determine, under leave-one-repo-out evaluation, which characteristics derived from it can be predicted at submission time in a way that generalizes to previously unseen projects.

The problem is stated as follows. Given a pull request observed at its submission instant, together with the history of the repository and of the author strictly preceding that instant, and given the process model of Section 3, determine which characteristics derived from that model admit a decision rule whose discriminative power on a repository absent from the training set remains materially above the base rate of that repository. The solution is constrained in three ways. Only information available at the submission

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

instant may enter the model, so that no post-submission evidence leaks into the prediction. The evaluation must hold out entire repositories rather than individual pull requests, so that project-specific conventions cannot be learned in training and reused in testing. The characteristics must follow from the structure of the process model rather than from the availability of convenient data. The criterion of success is therefore not a single accuracy figure, but the retention of discriminative power when a model is transferred to a project it has never seen.

RESEARCH METHODOLOGY

The model of the code review process should serve two purposes. Descriptively, it fixes a shared vocabulary of actors, activities, artifacts, and states of the contemporary PR-based review process, including its automated participants. Prescriptively, this study identifies which quantities are measurable from the process trace and therefore can serve as prediction targets. We model PR-based review as practiced on GitHub and similar platforms, since this workflow dominates both open-source and an increasing share of commercial development [4]. Email-based patch workflows (for example, the Linux kernel) are outside the model's scope.

We deliberately chose standard notations over a bespoke formalism: BPMN 2.0 for the process view and a UML state machine for the artifact view. Standard notations keep the model accessible to practitioners, enable tool-supported simulation in future work, and cleanly separate the two concerns: BPMN captures who does what and when, while the state machine captures what happens to the pull request as an artifact.

The model distinguishes four actors, each represented as a BPMN lane:

– Author: the developer (or bot) who prepares the change, opens the PR, and performs rework in response to feedback.

– Reviewer: one or more developers who inspect the change, leave comments, and issue verdicts (approval or change request). Reviewer assignment may be manual, rule-based, or recommendation-driven.

– AI Assistant: an automated participant based on a large language model that produces review suggestions, summaries, or draft fixes. The assistant acts on its own trigger (a PR opened or updated), and its output enters the same comment stream as human feedback, but its suggestions carry no gating authority: only human reviewers issue verdicts. This lane is not hypothetical in the studied corpus: twelve distinct assistants based on large language models act as reviewers in it, and 15.9 % of the merged pull requests carry at least one such review (Table 2). Their participation begins abruptly: the earliest such review event was recorded in February 2025, and it rises from 0.1 % of pull requests opened in 2024 to 14.3 % in 2025 and 35.9 % in 2026. Adoption is, however, almost entirely a property of the project rather than of the change, ranging from 0.1 % to 91.5 % across the twelve repositories, and an assistant reviews a change only after it has been submitted. A repository-level adoption rate computed in the expanding window therefore adds nothing measurable to either target, and we report the lane as empirically observed but not predictively useful at submission time.

– CI: the continuous integration service that runs automated checks (build, tests, static analysis) on every revision and reports a binary or graded verdict. Unlike the AI assistant, CI typically does have gating authority: a failing check blocks the merge in most configurations.

The central artifact is the pull request, which aggregates the proposed diff, its revision history (commits), the discussion (review comments and threads), formal review verdicts, CI check results, and metadata (title, description, labels, draft status). Secondary artifacts include the review comment (with its creation and resolution lifecycle) and the CI report.

The process view (Fig. 1) is organized as follows. The start event is the author's opening of a PR. This event triggers two parallel automated flows: in the CI lane, the task “Run automated checks” executes and terminates at an exclusive gateway, “Checks passed?”. In the AI Assistant lane, the task “Generate review suggestions” produces an intermediate event, “AI suggestion available”, whose output is attached to the PR discussion. A failing CI verdict routes control back to the Author lane (task “Fix the change”), and the loop repeats. In most configurations, human review may proceed in parallel, but the merge is blocked until checks pass.

In the Reviewer lane, the flow begins with the task “Assign reviewer(s)”, followed by “Inspect the change”, where the reviewer examines the diff, discussion, CI results, and any AI suggestions. Inspection terminates in the exclusive gateway “Verdict”, with three outgoing paths: “approve”, “request changes”, and “comment only” (feedback without a formal verdict). The “request changes” path activates the Author lane task “Rework the change”; the author pushes a new revision, which re-triggers CI and, in configured projects, the AI assistant, and returns control to “Inspect the

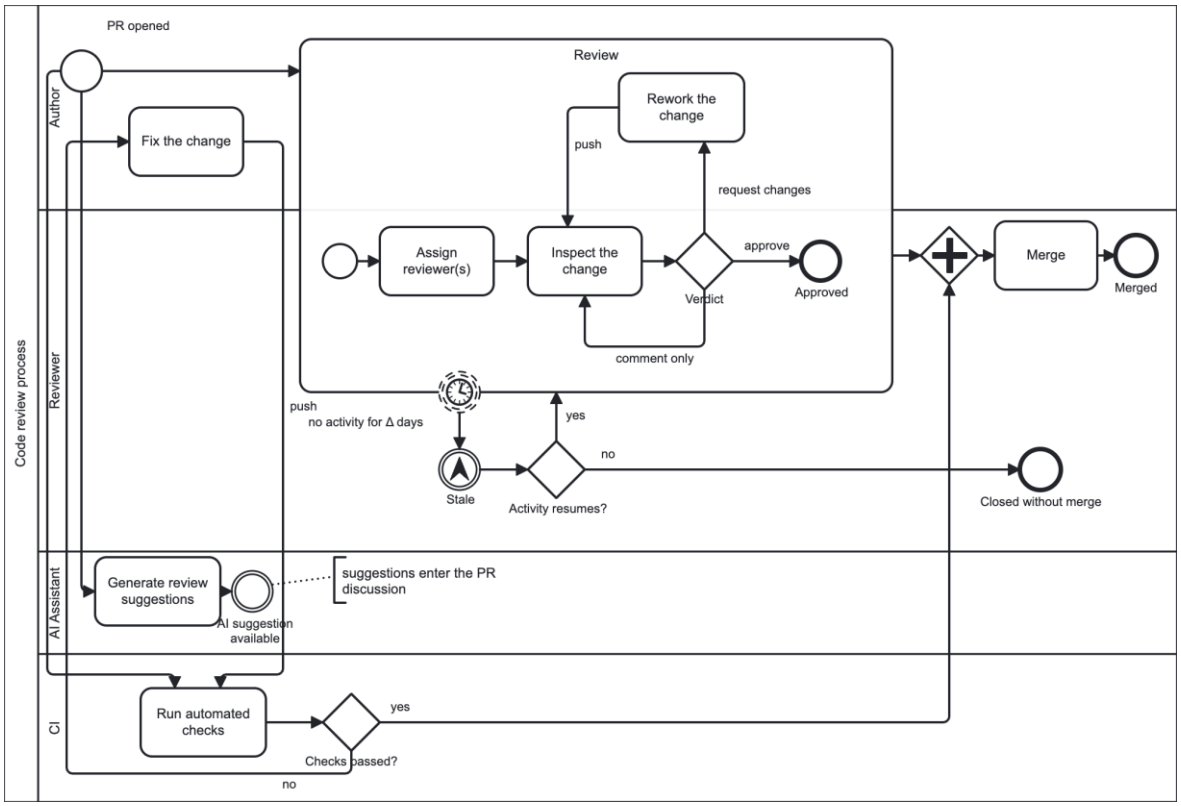


Fig. 1. **BPMN model of the code review process**

Source: **compiled by the authors**

change". Each traversal of this rework loop is one review iteration. The "approve" path leads to the task "Merge" (performed by a maintainer in the Reviewer lane or by the author, depending on project policy), which terminates the process in the end event "Merged". At any point, an explicit decision to abandon the change terminates the process in the end event "Closed without merge".

One element of the diagram deserves emphasis: a "timer boundary event" attached to the whole review subprocess. If no activity (commit, comment, verdict, or check) occurs within a threshold Δ, the timer fires, and the PR is marked as "stale". The stale condition is not terminal: the process may resume if any actor acts on the PR, or it may eventually drain into "Closed without merge". Modeling staleness as a timer event rather than a normal-flow state reflects its nature: it is the absence of process and is invisible in models that describe only what actors do.

The state machine (Fig. 2) describes the same process from the artifact's perspective. A PR begins in the state "Open" (submitted, awaiting first reviewer attention). The first review event moves it to "Under Review". From there, a "changes requested" verdict moves it to "Needs Changes"; the author's push of a new revision returns it to "Under Review". An approving verdict moves the PR to "Approved", from which the "merge" action leads to the final state "Merged". An explicit close action from any non-final state leads to the final state "Closed". Finally, a timer transition guarded by "no activity for Δ days" leads from any active state ("Open", "Under Review", "Needs Changes") to "Stale"; any new activity returns a stale PR to its previous state, and a close action moves it to "Closed".

The two views are consistent by construction: every BPMN task or event that affects the artifact corresponds to a transition in the state machine, and every state of the machine corresponds to a region of the BPMN flow. The state machine adds what the process view lacks: an explicit notion of time spent in a state, which is the foundation for the measurable characteristics derived next.

Let the trace of a pull request p be the finite timestamped event sequence

$$\sigma(p) = \langle (e_1, \tau_1), (e_2, \tau_2), \ldots, (e_k, \tau_k) \rangle,$$

where each e_i belongs to the event alphabet E = {opened, push, ci_verdict, ai_suggestion, review_approve, review_request_changes, review_comment, merge, close} and τ_i are non-decreasing timestamps. The state machine maps every prefix of $\sigma(p)$ to a state, so the trace induces a sequence of state sojourns. Three families of characteristics are then well-defined and computable from the trace alone.

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

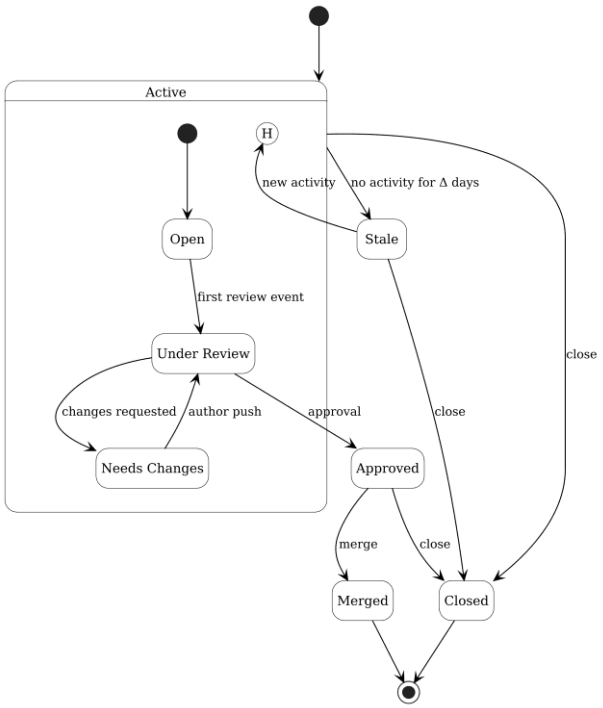


Fig. 2. **UML state machine of the pull request lifecycle**

Source: **compiled by the authors**

Definition 1. Review duration. Let E_{rev} = {review_approve, review_request_changes, review_comment} $\subset E$ denote the review events issued by a participant other than the author, so that the opening of the pull request is excluded by construction.

The time to first review is

$$t_{fr}(p) = \min_{e_i \in E_{rev}} (\tau_i - \tau_1),$$

and the time to merge

$$t_m(p) = \tau_{merge} - \tau_1.$$

In state machine terms, t_{fr} is the sojourn time of the initial "Open" state, and t_m is the total time from entry into "Open" to entry into "Merged".

Definition 2. Number of review iterations $n_{it}(p)$ is the number of cycles of the form "review event by a non-author, followed by one or more pushes by the author, followed by another review event" in $\sigma(p)$. In state machine terms, this counts the number of times the "Under Review" to "Needs Changes" (or comment-driven rework) loop is traversed. A PR approved without intervening rework has $n_{it} = 1$; $n_{it} \geq 2$ indicates that the review surfaced issues requiring rework.

Definition 3. Staleness $stale(p) = 1$ if there exists a time window of length Δ within the open lifetime of p containing no events, that is, if the timer transition fires. We set $\Delta = 30$ days, a common inactivity threshold for identifying stalled pull requests. Further, we discuss the label's sensitivity to this choice. We operationalize this condition directly as the longest interval between consecutive events of $\sigma(p)$ within the open lifetime of p. The condition is decidable for every pull request whose lifetime is fully observed, which is the case for merged pull requests.

Table 1 summarizes these characteristics.

These three characteristics are not arbitrary: each corresponds to a distinct structural element of the model (a sojourn, a loop, a timer), and together they cover the three failure modes of review efficiency that motivated this work: review takes too long, review costs too many iterations, and review silently does not happen. Accordingly, we define review efficiency for the purposes of this framework as the joint minimization of these three quantities (review duration, iteration cost, and staleness) so that the derived characteristics are precisely the measurable facets of efficiency and predicting them at submission time is the prerequisite for improving efficiency. They are also exactly the quantities that are knowable only after the fact, yet most valuable before it, which makes them natural targets for submission-time prediction.

Table 1. **Mapping of process model elements to measurable characteristics**

Characteristic	Defining model element	Trace-level definition	Operationalization in this study
Review duration	State sojourn	$t_{fr}(p)$: time from opening to the first review event; $t_m(p)$: time from opening to merge	Derived from the model; not validated here
Review iterations	Loop traversal	$n_{it}(p)$: number of cycles of reviewer feedback followed by author rework in the trace	Binary target T1: $n_{it} \geq 2$ vs $n_{it} < 2$
Staleness	Timer transition	$stale(p) = 1$ if a window of length Δ within the open lifetime of p contains no events	Binary target T2; $\Delta = 30$ days; longest inter-event interval in the trace

Source**: compiled by the authors**

This notion of efficiency is deliberately multi-objective. Writing $Eff(p) = \alpha \cdot t_m(p) + \beta \cdot n_{it}(p) + \gamma \cdot stale(p)$ with non-negative weights α, β, γ chosen by the project, review efficiency is the minimization of *Eff* over the pull requests of a repository. The weights are not universal: a project releasing on a fixed cadence penalizes duration heavily, whereas a safety-critical project may accept a long review and penalize only silent abandonment. The three terms are moreover not independent, which is precisely why a single scalar target would mislead. Pressing for a faster first response tends to raise the iteration count, because reviewers engage before the change has settled. Suppressing iterations by approving early lowers n_{it} at the cost of the defects the review existed to catch. Reducing staleness by reassigning idle pull requests redistributes reviewer attention rather than creating it, and can lengthen the queue for changes already under review. Predicting the three quantities separately, as this paper does, is what allows a project to choose its own position among these trade-offs rather than inherit one.

The empirical part of this paper validates the predictability of the second and third characteristics, both operationalized as binary classification at submission time: whether the PR will require two or more iterations ($n_{it} \geq 2$), and whether it will become stale. Duration prediction (the first characteristic) involves regression or interval classification with different evaluation machinery, and its detailed treatment is left to a companion study.

Cross-project claims require a dataset that varies the factors a single-project study holds constant: programming language, community structure, review culture, and project scale. We selected 12 open-source GitHub repositories according to the following criteria: (a) more than 1,000 merged PRs with review activity, ensuring sufficient per-repository sample size; (b) native GitHub review workflow, excluding projects that gate merges through external bots or mailing lists (for this reason kubernetes/kubernetes, which uses the Prow bot, and apache/spark, which merges outside GitHub, were excluded after piloting); (c) diversity of primary programming languages; and (d) public availability of rich PR metadata, including review events and CI status.

The resulting set comprises 12 repositories covering six primary languages (Table 2). The repositories span different domains (developer tooling, web frameworks, e-commerce, communication platforms) and different community structures, from company-driven projects with employed maintainers to volunteer-driven ones.

Table 2. **Repository statistics of the dataset**

Repository	Primary language	Merged PRs collected	Open PRs collected	Share of multi-iteration PRs (%)	Median time to first review (hours)	Median time to merge (hours)	Merged PRs with an AI review (%)
Woocommerce/ woocommerce	PHP	7.647	369	52.5	10.2	47.5	20.8
Facebook/react	JavaScript	7.127	401	27.4	2.1	12.6	0.1
Microsoft/ vscode	TypeScript	6.212	1.643	38.6	0.1	1.1	91.5
Vercel/next.js	TypeScript	8.392	1.604	34.9	2.9	16.6	15.9
Angular/angular	TypeScript	3.080	145	17.4	2.9	24.2	0.3
Supabase/supabase	TypeScript	5.869	755	39.9	2.0	17.5	14.8
Strapi/strapi	TypeScript	2.418	239	45.7	5.7	44.2	2.3
Grafana/grafana	Go	6.420	786	37.3	2.1	16.4	5.0
Mattermost/ mattermost	Go	5.081	329	40.6	1.8	33.5	6.7
Nodejs/node	JavaScript/ C++	1.872	300	23.6	3.4	67.2	0.6
Discourse/ discourse	Ruby	9.319	157	20.1	1.0	3.1	0.5
Django/django	Python	1.383	428	16.2	6.5	78.3	0.1
Pooled		64.820	7.156	34.4	1.7	17.0	15.9

***Source*: compiled by the authors**

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

The dataset exhibits substantial cross-project variability, which is precisely what a cross-project study needs: the share of PRs requiring two or more review iterations ranges from 16.2 % (django) to 52.5 % (woocommerce), the median time to first review ranges from 0.1 hours (vscode) to 10.2 hours (woocommerce), and the median time to merge ranges from 1.1 to 78.3 hours across repositories. The vscode repository is an outlier in review latency (median 0.1 hours to first review); it uses GitHub's native review workflow and was retained under criterion (b), but its near-instant first-review times inflate the speed of the pooled medians and should be read in that light.

Data were collected through the GitHub GraphQL API using a Python pipeline with checkpointing and rate-limit handling, building on the data-construction methodology of our earlier work [24]. For each PR, we retrieved metadata (title, description, labels, draft status, timestamps), diff metrics (additions, deletions, changed files), the commit history with timestamps, all review events with their verdicts and timestamps, review threads, and requested reviewers.

Merged PRs were added to the dataset if they had at least one review event, valid creation and merge timestamps, and targeted the main branch, the branch condition being a name match that also admits release and canary branches. We excluded PRs merged in under one minute (automated merges without review), PRs open for more than 365 days (abandoned and unrepresentative), and PRs without code changes; reverts and duplicates were not filtered out. After filtering, the dataset contains 64.820 merged PRs. The staleness target is computed on the same merged pull requests, from their event traces: a PR is labeled stale if the longest interval between consecutive events within its lifetime exceeds the threshold. We first operationalized staleness from a single collection snapshot of the 7.156 open pull requests, labeling a PR stale after the threshold had elapsed since its last recorded activity. Diagnostics led us to discard that design. The snapshot label is close to a restatement of pull request age: the stale share rises monotonically from zero below 30 days to 0.97 beyond one year; age alone attains an AUC-ROC of 0.938 against the label, which exceeds what the full feature set attains, and 32.7 % of the sample is structurally censored because a pull request younger than the threshold cannot satisfy the condition. The trace-derived label is free of these artifacts because the lifetime of a merged pull request is fully observed. Under it, 1.498 of the 64.820 merged PRs (2.3 %) are stale.

Review iterations were counted according to the trace definition. Two details matter for replication. First, all review verdict types (approve, request changes, comment) count as review events, since piloting showed that counting only formal “changes requested” verdicts yields a degenerate label distribution (over 90 % of PRs with zero counted rounds). The sensitivity of the iteration label to this counting rule warrants further study. Second, self-reviews are excluded, commits predating the PR are ignored, and multiple review events without an intervening push fall within one iteration. Under this definition, 34.4 % of merged PRs in the dataset required two or more iterations.

For every PR, we computed a set of submission-time features organized into groups that mirror the actors and artifacts of the process model.

The predictive models in this study use 24 of these features, drawn from the first three groups:

– PR-level features (artifact properties): lines added and deleted, files changed, number of commits, presence of tests, documentation, and CI configuration changes, description and title length, presence of screenshots and code blocks, issue references, draft status, label count, and the day of the week and hour of submission. 16 features.

– Developer history features (Author lane): the author's historical merge rate, the average number of review iterations on the author's previous PRs, commit frequency over a trailing 90-day window, tenure in the repository, and the author's prior PR count. 5 features.

– Project context features (process load): the number of open PRs at submission time and merge velocity over the preceding 7 and 30 days. These features quantify review-pipeline congestion. 3 features.

– Reviewer-level features (Reviewer lane): reviewer experience, average response time, current open review requests, file-path expertise, and familiarity with the author. This group is computed where a requested reviewer is known; because reviewer identity is generally not fixed at submission time, the prediction models in this study rely on the first three groups, that is, on 24 features in total, while the reviewer-level group supports reviewer-oriented extensions of the framework.

All historical features are computed under an expanding temporal window: for each PR, only data from PRs created strictly earlier contribute to its feature values. This protocol prevents temporal data leakage, a known pitfall in repository mining, in

which features inadvertently encode information from the future relative to the prediction point. Feature completeness across the 12 repositories ranges from 93.7 % to 100 %, with all PR-level features extractable with no missing values.

Developer-level features are aggregated metrics; the dataset and all analyses identify no individual contributors, and all collected data are publicly available through the GitHub API.

The model derives three characteristics; two of them are binary outcomes known at submission time, which we validate here, whereas review duration is a continuous quantity that calls for regression or interval estimation and is left to a companion study. We operationalize these two as submission-time binary classification:

– T1 (iterations). Will the PR require two or more review iterations ($n_{it} \geq 2$) before merging? The positive class covers 34.4 % of merged PRs. A positive prediction indicates that the change is likely to require rework, so reviewers can prioritize it, or the author can be prompted to improve the submission before review starts.

– T2 (staleness). Will the review of the PR stall at least once, that is, will the timer of the process model fire before the change is merged? A positive prediction enables early escalation before the change silently leaves the reviewers' attention. The positive class covers 2.3 % of merged PRs, so this is a rare-event problem, and the ranking metrics are the informative ones.

Both predictions use only features known at the time of submission.

The primary models are gradient-boosted decision-tree ensembles: CatBoost [25] and LightGBM [26]. Both handle heterogeneous tabular features, are robust to monotone transformations, capture non-linear interactions, and have shown strong performance on similar repository-mining tasks [6], [18]. Class imbalance is handled with automatic class weighting.

Three baselines bound the contribution of the full models: a majority-class dummy classifier (the floor), logistic regression (a linear reference), and a CatBoost model restricted to code-size features only (lines added and deleted, files changed, number of commits). The size-only baseline addresses a recurring confound in repository studies, where change size dominates predictions; comparing it with the full model isolates the added value of developer history and project context features.

We evaluated the models under two complementary protocols:

– Within-distribution: stratified 5-fold cross-validation over the pooled multi-repository dataset. This estimates performance when the deployment population resembles the training population.

– Cross-project: leave-one-repo-out (LORO). Models are trained on 11 repositories and tested on the held-out 12th repository, with training and testing rotating through all 12 repositories. LORO directly answers the question existing studies leave open: does a model trained on other projects work on a project it has never seen? This is the relevant protocol for the most useful deployment scenario, namely adopting the tool on a repository without historical data.

We reported F1, AUC-ROC, and AUC-PR; the latter is informative in the presence of class imbalance. Model interpretation uses SHAP (SHapley Additive exPlanations) values [27], which attribute each prediction to feature contributions and allow a global view of which process factors drive each characteristic.

RESULTS OF STUDY

Table 3 compares models for review iteration prediction.

Table 3. **Model comparison for iteration prediction (T1), 5-fold stratified cross-validation, mean ± standard deviation over folds**

Model	Accuracy	F1	Precision	Recall	AUC-ROC	AUC-PR
Dummy (majority)	0.656	0.000	0.000	0.000	0.500	0.344
Logistic regression	0.766 ± 0.005	0.566 ± 0.012	0.783 ± 0.010	0.443 ± 0.013	0.834 ± 0.006	0.719 ± 0.009
CatBoost (size-only)	0.815 ± 0.002	0.776 ± 0.002	0.664 ± 0.003	0.935 ± 0.002	0.903 ± 0.002	0.797 ± 0.004
CatBoost (all features)	0.839 ± 0.003	0.798 ± 0.004	0.702 ± 0.004	0.925 ± 0.003	0.928 ± 0.003	0.852 ± 0.006
LightGBM (all features)	0.842 ± 0.004	0.800 ± 0.004	0.707 ± 0.005	0.919 ± 0.004	0.928 ± 0.002	0.852 ± 0.004

***Source*: compiled by the authors**

Under 5-fold cross-validation, the full-feature gradient boosting models achieve F1 = 0.80 and AUC-ROC = 0.93 (LightGBM: F1 = 0.800, AUC-ROC = 0.928; CatBoost F1 = 0.798, AUC-ROC = 0.928). Over 25 paired folds (five repeats of stratified five-fold cross-validation, fold seeds 42 to 46; five folds alone cannot resolve such a difference, the smallest attainable two-sided signed-rank p being 0.0625) the AUC-ROC difference between the two learners is indistinguishable from zero (CatBoost minus LightGBM −0.0002, 95 % confidence interval [−0.0006, +0.0001], paired t-test p = 0.116, signed-rank p = 0.182), and the F1 difference, though detectable, is −0.0007 (95 % confidence interval [−0.0013, −0.0000], p = 0.040) against the 2.4 percentage point margin either learner holds over the size-only baseline. We therefore treat the two as practically equivalent and report CatBoost throughout.

The margin over logistic regression (F1 = 0.57) confirms that the signal is substantially non-linear. The size-only baseline reaches F1 = 0.78 and AUC-ROC = 0.90: change size alone carries much of the signal for iteration count, which is consistent with the intuition that larger changes attract more rework, but the full feature set still adds a consistent improvement, in particular in ranking quality (AUC-PR 0.852 versus 0.797). This headline F1 sits on a markedly asymmetric operating point: at the default 0.5 threshold, recall is about 0.92 but precision only about 0.70 (LightGBM 0.919/0.707; CatBoost 0.925/0.702). The errors are thus dominated by false positives (Fig. 3a) – the model flags almost every rework-heavy PR (few false negatives) but also over-flags some single-iteration PRs. This follows from the class weighting that offsets the 34.4 % positive base rate and deliberately favors recall. Since ranking quality is high (AUC-ROC = 0.93, AUC-PR = 0.85), the threshold is a free parameter that can be raised to trade recall for precision according to the cost a project places on a false alarm relative to a missed rework-heavy PR – a calibration that the cross-project results below make concrete.

Under LORO validation, the models retain AUC-ROC = 0.89 (CatBoost: F1 = 0.736, AUC-ROC = 0.887; LightGBM: F1 = 0.732, AUC-ROC = 0.881). Across the held-out repositories, the per-repository F1 is right-skewed (Fig. 3b), so we report its median as a reference: median F1 = 0.78 (mean 0.74, range 0.47-0.88). The mean is pulled down by the two lowest-base-rate projects, angular and django (17.4 % and 16.2 % multi-iteration PRs versus 34.4 % pooled), where the fixed 0.5 decision threshold over-predicts the positive class; on both, AUC-ROC remains 0.80-0.84, so the model still ranks pull requests well, and the low F1 reflects threshold miscalibration rather than weak discrimination. The modest drop from 0.93 to 0.89 indicates that most of what the model learns about iteration-inducing changes transfers to repositories it has never seen. We consider this the central empirical finding of the study: the propensity of a change to require rework is, to a large extent, a cross-project regularity rather than a project convention.

The SHAP summary plot (Fig. 4) identifies the number of commits in the PR at submission as the dominant predictor (47.9 % of total feature

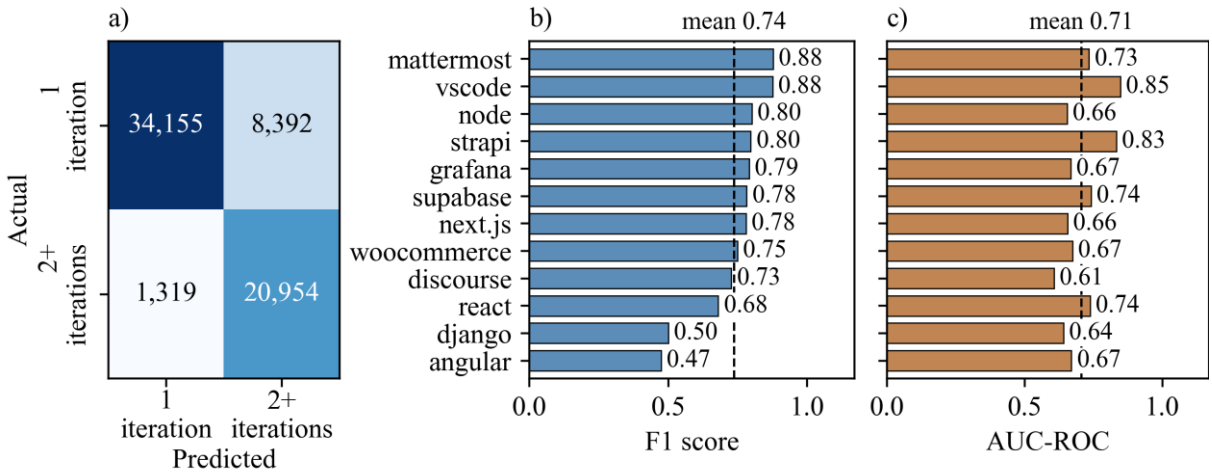


Fig. 3. **Model performance under LORO validation**
a – confusion matrix of the iteration model; b – per-repository F1 scores of the iteration model; c – per-repository AUC-ROC scores of the staleness model
Source: **compiled by the authors**

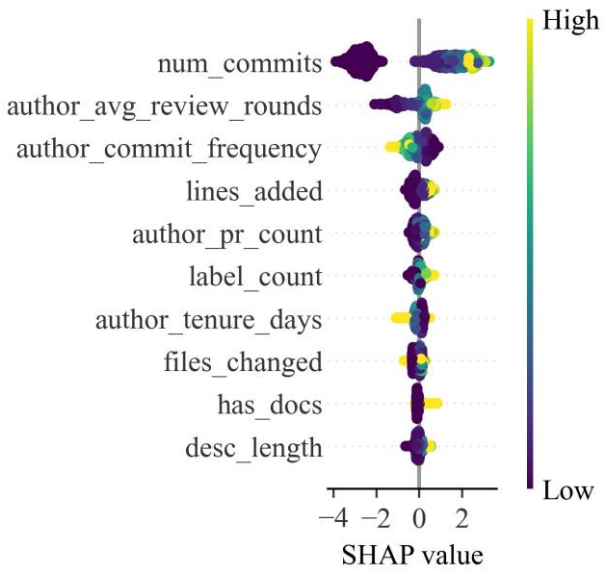


Fig. 4. **SHAP summary plot for the iteration model (T1); color encodes the feature value**

Source: **compiled by the authors**

importance in the CatBoost model), followed by the author's historical average iteration count (7.9 %), the author's recent commit frequency (5.4 %), the author's PR count (4.4 %), and 30-day merge velocity of the project (4.2 %). In terms of the process model, the loop is driven primarily by the artifact's properties (a multi-commit PR often reflects an unsettled change) and the Author lane history, with project congestion playing a secondary role. The commit count is measured at PR opening; commits added during review are excluded, so this feature carries no post-submission information. This dominance reflects the commit count specifically rather than change size: it correlates only moderately with the size features (Spearman 0.30-0.44 against lines added, lines deleted, and files changed) yet aligns far more strongly with the target (Spearman 0.60 with the iteration label, against at most 0.29 for any size feature), so a many-commit PR signals an unsettled change beyond what its size alone conveys. A fuller analysis of feature interactions is left to future work.

Table 4 reports the same comparison for staleness prediction.

For staleness, the full models attain AUC-ROC = 0.818 (CatBoost) and 0.811 (LightGBM) under 5-fold cross-validation. With a positive class of 2.3 %, the F1 score at the default threshold is low (0.176 and 0.187), reflecting the operating point rather than ranking quality: AUC-PR reaches 0.135 and 0.139 against a base rate of 0.023, a lift of 5.8 and 6.0 times. The contrast with the size-only baseline is again striking: restricted to code-size features, CatBoost falls to AUC-ROC = 0.594 and AUC-PR = 0.039, barely above the base rate. Staleness is therefore not a property of the change itself but of its context, which the SHAP summary plot (Fig. 5) makes explicit: the dominant predictors are the author's recent commit frequency (17.9 % of total importance), the author's prior PR count (11.0 %), the number of commits in the submitted change (9.6 %), the project's merge velocity over the preceding week (9.4 %) and month (8.1 %), and the length of the description (9.3 %). In model terms, the timer fires when the author is weakly coupled to the project and the project merges slowly, rather than because of the content of the change.

Under LORO validation, staleness prediction degrades more sharply, to a mean AUC-ROC of 0.706 (median 0.672, range 0.607-0.848 across the held-out repositories; Fig. 3c). The per-repository F1 stays low throughout (median 0.120, mean 0.132, range 0.043-0.252), which is what a fixed decision threshold yields against a 2.3 % base rate. The model remains usefully above chance on unseen repositories, but the gap between 0.818 and 0.706 indicates that a sizable component of stalling dynamics is project-specific.

Table 4. **Model comparison for staleness prediction (T2) on the trace-derived label, 5-fold stratified cross-validation, mean ± standard deviation over folds**

Model	Accuracy	F1	Precision	Recall	AUC-ROC	AUC-PR
Dummy (majority)	0.977	0.000	0.000	0.000	0.500	0.023
Logistic regression	0.977	0.000	0.000	0.000	0.757 ± 0.012	0.082 ± 0.009
CatBoost (size-only)	0.768 ± 0.004	0.073 ± 0.004	0.040 ± 0.002	0.393 ± 0.018	0.594 ± 0.022	0.039 ± 0.004
CatBoost (all features)	0.880 ± 0.003	0.176 ± 0.009	0.105 ± 0.005	0.553 ± 0.035	0.818 ± 0.013	0.135 ± 0.007
LightGBM (all features)	0.905 ± 0.002	0.187 ± 0.015	0.117 ± 0.009	0.472 ± 0.040	0.811 ± 0.011	0.139 ± 0.007

***Source*: compiled by the authors**

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

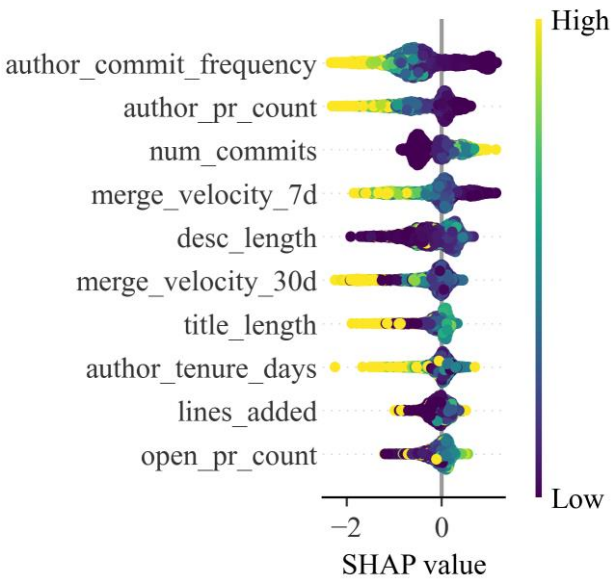


Fig. 5. **SHAP summary plot for the staleness model (T2); color encodes the feature value**

Source: **compiled by the authors**

DISCUSSION

The two targets behave differently under cross-project validation, and the difference is interpretable through the process model. Iteration count is driven by properties of the artifact and the author (commit structure, change size, the author's track record), which means roughly the same thing across repositories: a 40-file PR with 15 commits from an author who historically needs rework is risky everywhere. Accordingly, performance degrades only mildly under LORO (AUC-ROC 0.93 to 0.89).

Staleness, in contrast, is dominated by the author's relationship with the project and by the project's own cadence, and both have project-specific scales. A merge velocity that signals a healthy pipeline in one project is slow in another; what counts as an established contributor differs with the size of the community. A model trained on eleven repositories learns thresholds calibrated to those repositories, and these thresholds transfer imperfectly (AUC-ROC 0.818 to 0.706). This asymmetry is consistent with the structure of the model: the iteration loop is an interaction between the Author and Reviewer lanes mediated by the artifact, while the staleness timer is a property of the whole process instance embedded in its environmex31nt.

The practical implication is a two-tier deployment guidance. Iteration prediction can be shipped as a pre-trained cross-project model and applied to new repositories with modest degradation. Staleness prediction benefits from per-project calibration (for example, threshold tuning or fine-tuning on a small local history) before its predictions are acted upon [28].

Concretely, the predictions support three decisions at the moment a pull request arrives. The first is queue ordering: a change predicted to need rework can be routed to a reviewer with capacity for a longer exchange, while changes predicted to pass in a single round can be batched into short review slots. The second is pre-review feedback to the author: because the dominant predictor of rework is the commit structure of the submitted change rather than any property of the reviewer, a high predicted iteration count can be returned to the author before review starts, as a prompt to split or consolidate the change. The third is escalation: a high predicted staleness risk identifies the changes that would otherwise leave the pipeline unnoticed, so that a maintainer can assign a reviewer explicitly instead of waiting for the timer to fire. In each case, the prediction changes who looks at the change and when, not whether the change is accepted; the merge decision remains with the human reviewer.

Direct comparison with prior work is complicated by differing targets and protocols, so the following comparison is indicative rather than strict. Fan et al. [19] report an average AUC of 0.73 for early-merge prediction on three Gerrit projects, and Huang et al. [20] report within-project accuracies of 0.72-0.80 for review rounds on three further Gerrit projects, degrading to an F1 of 0.27-0.35 when the model is transferred between projects. Our iteration model attains a median F1 of 0.78 across the twelve held-out repositories, which suggests better transfer, although the platform and the wider project sample differ, so the difference should not be read as evidence that the method itself is superior. The Microsoft line of work [6], [7] demonstrated organizational-scale deployment within a single company's homogeneous tooling; our results suggest that cross-organizational transfer is feasible for artifact-driven characteristics but harder for context-driven ones, a distinction that prior work was not positioned to observe.

Concerning construct validity, review iterations are computed from review event sequences using Definition 2. Alternative definitions (for example, counting only formal change requests) yield different label distributions, and piloting showed that our definition is the most informative, but the choice remains a modeling decision. The staleness threshold $\Delta = 30$ days follows prior practice, and we treat it as a parameter. Repeating the experiment for

Δ between 15 and 90 days leaves discrimination essentially unchanged (AUC-ROC 0.818 to 0.839), so the conclusions do not depend on the particular threshold, although the positive rate and hence the F1 score do. The more consequential decision is the population: because the label requires a fully observed lifetime, it is computed on merged pull requests, and it therefore measures stalls that were eventually overcome rather than permanent abandonment. Pull requests that are never resolved lie outside this population. Measuring permanent abandonment at submission time requires longitudinal observation of open pull requests rather than a single snapshot, and we leave it to future work.

Concerning internal validity, the expanding temporal window eliminates the most common leakage path, and all features are restricted to submission-time information. Residual risks include subtle correlations between pooled-CV folds (the same author appearing in train and test); the LORO protocol, which separates entire repositories, is immune to this particular risk and is the basis of our cross-project claims.

External validity is the most important limitation, and we state it plainly: the dataset comprises 12 repositories, all open source, all on GitHub, all using the native PR workflow, in six languages. Twelve repositories are sufficient to demonstrate cross-project signal under LORO, but each LORO fold generalizes from only 11 source projects, and the diversity of the sample is bounded. Our findings should not be extrapolated to commercial closed-source development, to email-based review workflows, or to languages and ecosystems not represented in the sample. Expanding the repository pool is the first item of future work.

Concerning conclusion validity, results are averaged over folds. Per-repository LORO variance is non-trivial (Fig. 3b, c), so performance on any individual new repository may deviate from the mean in either direction.

Three directions follow directly from this study. First, scaling the dataset to 30 or more repositories, which will tighten the LORO estimates, enable per-language analysis (does a model trained on PHP projects transfer to Python projects?), and support an open-source versus commercial comparison if suitable industrial data can be obtained; all three analyses bear on the boundary of the framework's applicability. Second, the process model explicitly includes an AI Assistant lane. Building on it, we are developing an AI-assisted review methodology in which model predictions determine where and how LLM-based assistance enters the loop, with human feedback on assistant output accumulated to improve the assistant. Third, the Reviewer lane and the reviewer-level feature group support a reviewer ranking method that combines technical expertise, current workload, and predicted responsiveness [28]; the predictive models in this paper provide the responsiveness component. The latter two directions are parts of the broader research program, of which this framework is the foundation.

CONCLUSIONS

The contribution of this paper is twofold. The first part is an explicit model of the code review process, expressed in BPMN 2.0 with author, reviewer, AI assistant, and CI lanes and complemented by a UML state machine of the pull request lifecycle, which treats the two automated participants as first-class actors of the process rather than as tooling arranged around it; the process descriptions surveyed in the related work do not. The second is the way the prediction targets are obtained: duration, iterations, and staleness are not chosen for convenience but derived from structural elements of that model, namely state sojourns, loop traversals, and timer events. The empirical part of the paper asks whether targets obtained in this way are predictable at submission time, and whether that predictability survives a change of project.

Two of these three targets, review iterations and staleness, are validated empirically in this paper. Duration prediction is a continuous quantity that calls for regression or interval estimation with different evaluation machinery, and it is the subject of a separate study; the claims made here are accordingly confined to the two validated targets.

The central empirical finding is an asymmetry in transfer. Under leave-one-repository-out validation, the iteration model retains an AUC-ROC of 0.89, whereas the staleness model falls to 0.71. The process model accounts for the difference: iterations are driven by the artifact and its author, which means much the same thing from one project to the next, while stalling is a property of the whole process instance embedded in a particular project. The practical consequence is a two-tier deployment guidance. Iteration prediction can be shipped as a pre-trained cross-project model; staleness prediction requires per-project calibration before its output is acted upon. Within the limits of a twelve-repository open-source sample, this supports the central claim

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

of the framework: characteristics derived from an explicit process model are predictable at submission time, and a useful share of that predictability carries across projects.

Two directions extend this work. Because the staleness label is computed on merged pull requests, every stall it records was eventually overcome. Whether a stalled change is ever merged at all is a separate question, and it requires following open pull requests over time, treating those still unresolved when observation ends as censored, which is survival analysis rather than a single snapshot. Enlarging the sample beyond twelve open-source repositories would tighten the cross-project estimates and delimit the boundary of the framework's applicability.

Conflicts of Interest: The author Oleksandr Aratovskyi declares that they have no conflicts of interest regarding this study, including financial, personal, authorship, or other interests that could influence the research and the results presented in this article. Author Vira Liubchenko is a member of the Editorial Board of this journal. This role had no influence on the peer-review process or the editorial decision regarding this manuscript

Financing: Research was carried out without financial support

Data availability: The analysis is based on data collected from the public GitHub API. The source code of the data collection and modeling pipeline is published in a public GitHub repository. The derived dataset is being prepared for release under the FAIR principles (findable, accessible, interoperable, reusable) with a persistent identifier; until that release, it is available from the authors upon reasonable request.

Using artificial intelligence tools: During manuscript preparation, the artificial intelligence (AI) tool Claude was used solely for grammatical and stylistic proofreading. The authors subsequently reviewed and edited all text fragments processed by the AI tool. AI was not used to generate text, formulas, or tables, to formulate scientific statements, to obtain or interpret results, or to draw conclusions. All scientific statements, results, and conclusions are the original intellectual contributions of the authors.

REFERENCES

1. Bacchelli, A. & Bird, C. “Expectations, outcomes, and challenges of modern code review”. *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA. 2013. p. 712–721, https://www.scopus.com/pages/publications/84886394450. DOI: https://doi.org/10.1109/ICSE.2013.6606617.

2. Sadowski, C., Söderberg, E., Church, L., Sipko, M. & Bacchelli, A. “Modern code review: a case study at Google”. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Gothenburg, Sweden. 2018. p. 181–190, https://www.scopus.com/pages/publications/85049691666. DOI: https://doi.org/10.1145/3183519.3183525.

3. Badampudi, D., Unterkalmsteiner, M. & Britto, R. “Modern code reviews: survey of literature and practice”. *ACM Transactions on Software Engineering and Methodology*. 2023; 32 (4): 1–61, https://www.scopus.com/pages/publications/85163852501. DOI: https://doi.org/10.1145/3585004.

4. Gousios, G., Pinzger, M. & van Deursen, A. “An exploratory study of the pull-based software development model”. *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. Hyderabad, India. 2014. p. 345–355, https://www.scopus.com/pages/publications/84994130040. DOI: https://doi.org/10.1145/2568225.2568260.

5. Yu, Y., Wang, H., Filkov, V., Devanbu, P. & Vasilescu, B. “Wait for it: determinants of pull request evaluation latency on GitHub”. *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*. Florence, Italy. 2015. p. 367–371, https://www.scopus.com/pages/publications/84957085916. DOI: https://doi.org/10.1109/MSR.2015.42.

6. Maddila, C., Bansal, C. & Nagappan, N. “Predicting pull request completion time: a case study on large scale cloud services”. *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Tallinn, Estonia. 2019. p. 874–882, https://www.scopus.com/pages/publications/85071919194. DOI: https://doi.org/10.1145/3338906.3340457.

7. Maddila, C. S., Upadrasta, S. S., Bansal, C., Nagappan, N., Gousios, G. & van Deursen, A. “Nudge: accelerating overdue pull requests toward completion”. *ACM Transactions on Software Engineering and Methodology*. 2023; 32 (2): 1–30, https://www.scopus.com/pages/publications/85143061311. DOI: https://doi.org/10.1145/3544791.

8. Chouchen, M., Ouni, A., Olongo, J. & Mkaouer, M. W. “Learning to predict code review completion time in modern code review”. *Empirical Software Engineering*. 2023; 28 (4): 82, https://www.scopus.com/pages/publications/85160020919. DOI: https://doi.org/10.1007/s10664-023-10300-3.

9. Rasheed, Z., Sami, M. A., Waseem, M., Kemell, K.-K., Wang, X., Nguyen, A., Systä, K. & Abrahamsson, P. “AI-powered code review with LLMs: early results”. *arXiv*. 2024. DOI: https://doi.org/10.48550/arXiv.2404.18496.

10. Frömmgen, A., Austin, J., Choy, P., Ghelani, N., Kharatyan, L., Surita, G., Khrapko, E., Lamblin, P., Manzagol, P.-A., Revaj, M., Tabachnyk, M., Tarlow, D., Villela, K., Zheng, D., Chandra, S. & Maniatis, P. “Resolving code review comments with machine learning”. *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Lisbon, Portugal. 2024. p. 204–215, https://www.scopus.com/pages/publications/105011043499. DOI: https://doi.org/10.1145/3639477.3639746.

11. Komleva, N., Liubchenko, V. & Zinovatna, S. “Evaluation of the quality of survey data and its visualization using dashboards”. *Proceedings of the 2020 IEEE 15th International Conference on Computer Sciences and Information Technologies (CSIT)*. Vol. 2. Zbarazh, Ukraine. 2020. p. 234–237, https://www.scopus.com/pages/publications/85100509851.
DOI: https://doi.org/10.1109/CSIT49958.2020.9321970.

12. Fuggetta, A. “Software process: a roadmap”. *Proceedings of the Conference on the Future of Software Engineering (ICSE)*. Limerick, Ireland. 2000. p. 25–34, https://www.scopus.com/pages/publications/85023607484. DOI: https://doi.org/10.1145/336512.336521.

13. García-Borgoñón, L., Barcelona, M. A., García-García, J. A., Alba, M. & Escalona, M. J. “Software process modeling languages: a systematic literature review”. *Information and Software Technology*. 2014; 56 (2): 103–116, https://www.scopus.com/pages/publications/84889885711. DOI: https://doi.org/10.1016/j.infsof.2013.10.001.

14. Rigby, P. C., German, D. M., Cowen, L. & Storey, M.-A. “Peer review on open-source software projects: parameters, statistical models, and theory”. *ACM Transactions on Software Engineering and Methodology*. 2014; 23 (4): 1–33, https://www.scopus.com/pages/publications/84907033452. DOI: https://doi.org/10.1145/2594458.

15. Thongtanunam, P., McIntosh, S., Hassan, A. E. & Iida, H. “Review participation in modern code review: an empirical study of the Android, Qt, and OpenStack projects”. *Empirical Software Engineering*. 2017; 22 (2): 768–817, https://www.scopus.com/pages/publications/84991786769. DOI: https://doi.org/10.1007/s10664-016-9452-6.

16. Baysal, O., Kononenko, O., Holmes, R. & Godfrey, M. W. “Investigating technical and non-technical factors influencing modern code review”. *Empirical Software Engineering*. 2016; 21 (3): 932–959, https://www.scopus.com/pages/publications/84926052272. DOI: https://doi.org/10.1007/s10664-015-9366-8.

17. Wessel, M., de Souza, B. M., Steinmacher, I., Wiese, I. S., Polato, I., Chaves, A. P. & Gerosa, M. A. “The power of bots: characterizing and understanding bots in OSS projects”. *Proceedings of the ACM on Human-Computer Interaction*. 2018; 2 (CSCW): 1–19, https://www.scopus.com/pages/publications/85064157845. DOI: https://doi.org/10.1145/3274451.

18. Khatoonabadi, S., Abdellatif, A., Costa, D. E. & Shihab, E. “Predicting the first response latency of maintainers and contributors in pull requests”. *IEEE Transactions on Software Engineering*. 2024; 50 (10): 2529–2543, https://www.scopus.com/pages/publications/85201309320. D
OI: https://doi.org/10.1109/TSE.2024.3443741.

19. Fan, Y., Xia, X., Lo, D. & Li, S. “Early prediction of merged code changes to prioritize reviewing tasks”. *Empirical Software Engineering*. 2018; 23 (6): 3346–3393, https://www.scopus.com/pages/publications/85044233675. DOI: https://doi.org/10.1007/s10664-018-9602-0.

20. Huang, Y., Liang, X., Chen, Z., Jia, N., Luo, X., Chen, X., Zheng, Z. & Zhou, X. “Reviewing rounds prediction for code patches”. *Empirical Software Engineering*. 2022; 27 (1): 7, https://www.scopus.com/pages/publications/85117702514. DOI: https://doi.org/10.1007/s10664-021-10035-z.

21. Li, Z., Yu, Y., Wang, T., Yin, G., Li, S. & Wang, H. “Are you still working on this? An empirical study on pull request abandonment”. *IEEE Transactions on Software Engineering*. 2022; 48 (6): 2173–2188, https://www.scopus.com/pages/publications/85100455950. DOI: https://doi.org/10.1109/TSE.2021.3053403.

22. Zimmermann, T., Nagappan, N., Gall, H., Giger, E. & Murphy, B. “Cross-project defect prediction: a large scale experiment on data vs. domain vs. process”. *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Amsterdam, The Netherlands. 2009. p. 91–100, https://www.scopus.com/pages/publications/77949378886. DOI: https://doi.org/10.1145/1595696.1595713.

23. Hosseini, S., Turhan, B. & Gunarathna, D. “A systematic literature review and meta-analysis on cross project defect prediction”. *IEEE Transactions on Software Engineering*. 2019; 45 (2): 111–147, https://www.scopus.com/pages/publications/85034218670. DOI: https://doi.org/10.1109/TSE.2017.2770124.

24. Aratovskyi, O. S. & Liubchenko, V. V. “Constructing a GitHub-based dataset for machine learning driven software effort estimation”. *Proceedings of the 2025 IEEE 13th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. 2025. p. 1–7, https://www.scopus.com/pages/publications/105032752133. DOI: https://doi.org/10.1109/IDAACS68557.2025.11322078.

25. Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V. & Gulin, A. “CatBoost: unbiased boosting with categorical features”. *Advances in Neural Information Processing Systems 31 (NeurIPS*). Montreal, Canada. 2018. p. 6638–6648.

26. Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q. & Liu, T.-Y. “LightGBM: a highly efficient gradient boosting decision tree”. *Advances in Neural Information Processing Systems 30 (NIPS)*. Long Beach, CA, USA. 2017. p. 3146–3154.

27. Lundberg, S. M. & Lee, S.-I. “A unified approach to interpreting model predictions”. *Advances in Neural Information Processing Systems 30 (NIPS)*. Long Beach, CA, USA. 2017. p. 4765–4774.

28. Komleva, N., Liubchenko, V. & Zinovatna, S. “Improvement of teaching quality in the view of a resource-based approach”. *Proceedings of the 16th International Conference on ICT in Education, Research and Industrial Applications (ICTERI 2020), Vol. I: Main Conference*. 2020. CEUR Workshop Proceedings, vol. 2740, p. 262–277, https://www.scopus.com/pages/publications/85096415514. DOI: https://doi.org/10.55056/ceur-ws.org/Vol-2740/20200262.pdf.

29. Thongtanunam, P., Tantithamthavorn, C., Kula, R. G., Yoshida, N., Iida, H. & Matsumoto, K. “Who should review my code? A file location-based code-reviewer recommendation approach for modern code review”. *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Montreal, Canada. 2015. p. 141–150, https://www.scopus.com/pages/publications/84928683559. DOI: https://doi.org/10.1109/SANER.2015.7081824.

Received 26.08.2026
Received after revision 21.10.2026
Accepted 27.10.2026

DOI: https://doi.org/10.15276/hait.09.2026.36
УДК 004.41:004.85

Фреймворк прогнозування результатів перевірки програмного коду на основі формальної моделі процесу

Аратовський Олександр Сергійович[1)]
ORCID: https://orcid.org/0009-0006-0854-8582; alex.aratovskiy@stud.op.edu.ua. Scopus Author ID: 60503533100
Любченко Віра Вікторівна[1), 2)]
ORCID: https://orcid.org/0000-0002-4611-7832; lvv@op.edu.ua. Scopus Author ID: 56667638800
[1)] Національний університет «Одеська політехніка», пр. Шевченка, 1. Одеса, 65044, Україна
[2)] Університет прикладних наук, Ульменліт 20. Гамбург, 21033, Німеччина

АНОТАЦІЯ

Актуальність. Перевірка програмного коду є основною практикою забезпечення якості в сучасній розробці програмного забезпечення, проте її результати важко передбачити на момент подання змін: тривалість значно варіюється, кількість циклів доопрацювання невідома, а значна частина запитів на злиття непомітно втрачає увагу рецензентів. Наявні підходи розв’язують ці проблеми окремо і перевіряються в межах одного репозиторію. **Метою статті** є побудова явної

моделі процесу перевірки коду та доведення того, що виведені з неї характеристики можна прогнозувати на момент подання за допомогою моделей, узагальнених для різних проєктів. **Завдання:** сформулювати модель процесу, що трактує автоматизованих учасників як дійових осіб; вивести з неї вимірювані характеристики перевірки; побудувати набір даних із багатьох репозиторіїв; перевірити прогнозованість двох із цих характеристик. **Методи.** Модель поєднує нотацію моделювання бізнес-процесів із машиною станів уніфікованої мови моделювання. Класифікатори градієнтного бустингу використовують відомі на момент подання ознаки зміни, історії автора та контексту проєкту, обчислені в розширюваному часовому вікні, що запобігає витоку даних; оцінювання виконано стратифікованою перехресною перевіркою та перевіркою з виключенням одного репозиторію. **Наукова новизна.** Цілі прогнозування виведено зі структурних елементів моделі, а саме перебування у стані, циклу та таймера, а не обрано прагматично; безперервну інтеграцію та асистента на основі великої мовної моделі трактовано як дійових осіб процесу. **Практична значимість.** Моделі підтримують оцінювання ризику вхідних змін на момент подання та пріоритизацію перевірки. **Результати.** Прогнозування кількості ітерацій узагальнюється на репозиторії, не бачені під час навчання; прогнозування застарілості переноситься менш надійно, що вказує на сильнішу залежність від контексту проєкту. **Висновки.** Характеристики, виведені з явної моделі процесу, є прогнозованими на момент подання, і значна частка цієї прогнозованості переноситься між проєктами.

Ключові слова: інженерія програмного забезпечення; перевірка програмного коду; запит на злиття; модель процесу; моделювання бізнес-процесів; машина станів; аналіз репозиторіїв програмного забезпечення; машинне навчання; градієнтний бустинг; крос-проєктне прогнозування

ABOUT THE AUTHORS



Oleksandr S. Aratovskyi – PhD student, Department of Software Engineering, Odessa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine.
ORCID: https://orcid.org/0009-0006-0854-8582; alex.aratovskiy@stud.op.edu.ua. Scopus Author ID: 60503533100
Research field: machine learning for software engineering; code review analytics; mining software repositories

Аратовський Олександр Сергійович – аспірант кафедри Інженерії програмного забезпечення. Національний університет «Одеська політехніка», пр. Шевченка, 1. Одеса, 65044, Україна
Галузь досліджень: машинне навчання для розробки програмного забезпечення; аналітика перевірки коду; репозиторії програмного забезпечення для майнінгу



Vira V. Liubchenko – Doctor of Engineering Sciences, Professor, Department of Software Engineering, Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine;
Lecturer, Fakultät Life Sciences, Hochschule für Angewandte Wissenschaften Hamburg, Ulmenliet 20, Hamburg, 21033, Germany
ORCID: https://orcid.org/0000-0002-4611-7832; lvv@op.edu.ua. Scopus Author ID: 56667638800
Scientific field: data science, software engineering, project management

Любченко Віра Вікторівна – доктор технічних наук, професор кафедри Інженерії програмного забезпечення Національного університету «Одеська політехніка», пр. Шевченка, 1. Одеса, 65044, Україна;
факультет наук про життя, Гамбурзький університет прикладних наук, Ульменліт 20. Гамбург, 21033, Німеччина
Галузь досліджень: Наука про дані, програмна інженерія, управління проектами