

QModel: A Time-Aware GitHub Mining Framework for Empirical Software Quality Studies

Dmytro Polishchuk 

Jagiellonian University, Kraków, Poland

Abstract

Quality-oriented mining software repositories studies require more than access to commits, issues, pull requests and CI logs. They require a reproducible way to represent how technical, social, temporal and defect-related evidence is linked across repository history. Existing MSR infrastructures provide important support for repository mining, but they do not directly expose a project-level, time-aware empirical model in which cross-artifact quality evidence can be queried, inspected and reused across alternative study designs.

This paper presents *QModel*, an emerging infrastructure for reproducible GitHub mining in empirical software quality studies. QModel stores Git, GitHub, CI, timeline, reaction, graph, churn and SZZ-style candidate-defect evidence in a project-centric relational schema. Its SQL-based compilation layer allows researchers to define feature-target datasets over arbitrary analysis units, including pull requests, issues, commits, files, time windows and custom cross-artifact objects.

We evaluate QModel on `ansible/ansible` and `facebook/react`, mining 76,475 commits, 70,509 pull requests, 45,908 issues, 2,174,020 timeline events, 315,235 file-change records and 446,161 CI records. The early results show that QModel can materialize analysis-ready datasets with computable target, graph, churn and provenance features, while also making evidence gaps explicit when historical links, CI records, or fixing evidence are incomplete. These results support QModel as an empirical infrastructure for making quality-oriented repository operationalizations explicit, reusable and systematically evaluable.

2012 ACM Subject Classification Software and its engineering → Software creation and management; Software and its engineering → Software notations and tools; Information systems → Data mining

Keywords and phrases Mining software repositories, CI/CD, software quality, reproducibility

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.56

Category Emerging Results, Vision & Reflection Track Paper

Supplementary Material *Dataset (Replication artifact)*: <https://zenodo.org/records/21737012>

1 Introduction

Open-source repositories contain rich traces of software development, including commits, pull requests, issues, review discussions, continuous-integration checks, file changes, timelines and user reactions. These traces have made mining software repositories a central methodology in empirical software engineering. However, raw repository traces are not yet empirical datasets. A quality-oriented study must still decide which artifacts form the unit of analysis, which events define the target variable, which links among artifacts are valid and which derived metrics are available for each row of the dataset.

This operationalization step is a major reproducibility risk in repository-mining research. MSR workflows transform repository data into analytical datasets through retrieval, parsing, cleaning, filtering, parameterization and analysis; these intermediate steps and artifacts are frequently incompletely reported or unavailable [13, 5, 18]. A researcher may need to combine Git history traversal, GitHub API extraction, issue-pull-request linking, pull-request-commit mapping, timeline reconstruction, CI aggregation, churn computation, commit-graph analysis and SZZ-style defect tracing. Git and GitHub introduce additional ambiguity concerning



© Dmytro Polishchuk;

licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 56; pp. 56:1–56:14



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

branch and merge traversal, forks, repository selection and pull-request merge status [9], while Git timestamps are frequently noisy and can affect time-based analyses [4]. Cross-artifact links and defect labels are also heuristic: alternative issue-linking algorithms recover different sets of defect-fixing commits [11] and SZZ variants are often underspecified and can produce substantially different or noisy defect-provenance labels [14, 8, 1]. Consequently, two studies may mine the same repository yet construct different analytical datasets because of different linking rules, filters, timestamp definitions, branch and merge handling, aggregation choices, or defect-provenance heuristics.

The problem is especially important for software-quality research. Concepts such as maintainability, reliability, reviewability, or defect-proneness are not directly stored in GitHub. They must be represented through observable signals. Therefore, the central challenge is not only to collect repository artifacts, but to preserve the relations among them so that quality hypotheses can be expressed, inspected and repeated.

This paper presents *QModel*, an emerging infrastructure for time-aware, quality-oriented GitHub mining. We do not claim that repository mining, pull-request analysis, or SZZ-style defect tracing are new. The contribution is instead the operationalization of these elements in a single project-centric framework. QModel persists Git, GitHub, CI, timeline, reaction, graph, churn, and candidate-defect evidence as linked, queryable relations. Researchers can then define issue-level, pull-request-level, commit-level, or file-level datasets in SQL and reuse the same mined database across multiple empirical studies.

This work is submitted as an *Emerging Results* paper. The new idea under initial evaluation is an *executable operationalization*: a study definition that can materialize its analysis rows and audit whether target and predictor evidence are jointly computable. The initial empirical result is that computability varies sharply by analysis unit and evidence path, even when the underlying repository database is fixed. Thus, the contribution is not only that the pipeline executes, but that it makes the consequences of alternative operationalization choices directly observable. The paper does not claim a universal theory of software quality or a final review-time benchmark.

Study Goal

The pilot evaluates two stages of reproducible study construction: materializing analysis-ready datasets under alternative operationalizations and instantiating an empirical analysis from the same persisted evidence.

Research Questions

RQ1

How complete are the datasets materialized by QModel with respect to the target variable, metrics derived from the directed acyclic graph (DAG) of commits and churn measures?

The commit DAG comprises commits and their parent relationships. Completeness is the proportion of rows containing the target and the required DAG and churn metrics.

RQ2

Can QModel materialize an analysis-ready dataset from persisted repository artifacts, cross-artifact links and derived metrics for a reproducible empirical analysis?

We examine this question using pull-request review time as a worked example. Researchers can define other analysis units, targets and features over the same persisted repository data.

Contributions

QModel makes three contributions. First, its project-centric ingestion pipeline and schema integrate Git, GitHub and CI artifacts, compute churn and commit-DAG metrics, link defect-related issues to fixing commits and identify candidate bug-introducing commits (BICs) using an SZZ-style tracer. Second, its SQL-based compilation layer makes analysis units, targets, features, links, filters and aggregations explicit. Third, the pilot shows that completeness varies across operationalizations and that the same persisted evidence can support different empirical analyses.

2 Background and Research Gap

The mining software repositories community has produced several important infrastructures. GHTorrent mirrors GitHub metadata into a queryable database [6]; Boa provides a language and infrastructure for large-scale repository mining [3]; World of Code indexes a large universe of version-control objects [12]; and PyDriller offers a convenient Python framework for mining Git histories [15]. These infrastructures substantially lower the cost of obtaining repository evidence. However, quality-oriented empirical studies need more than artifact access: they need a reproducible way to turn repository history into analysis objects whose links, timestamps, derived metrics and uncertainties are explicit.

This is the operationalization gap targeted by QModel. In a quality study, the same repository history can support many plausible datasets depending on the chosen unit of analysis, linking rules, event timestamps, feature definitions, and defect-provenance assumptions. If these choices remain embedded in project-specific scripts, it becomes difficult to inspect why two studies over the same repository produce different rows, targets, or predictors. QModel therefore treats the repository not only as a source of artifacts, but as a linked empirical model from which alternative quality-oriented datasets can be defined, inspected and reused.

Table 1 summarizes this distinction. The comparison does not claim that the listed infrastructures cannot be extended. Rather, it asks whether the capability is available as a native, analysis-ready abstraction for linked, quality-oriented GitHub studies.

Prior work on pull requests and code review shows that contribution evaluation and review outcomes depend on both social and technical factors [7, 17]. Prior work on just-in-time quality assurance shows that commit-level history and churn can support defect-oriented analyses [10], while SZZ-style tracing shows that candidate defect links are useful but sensitive to implementation choices [1]. These observations motivate a framework that stores cross-artifact relations explicitly and treats derived evidence as queryable but inspectable empirical constructs.

QModel addresses this operationalization gap. It does not replace broad ecosystem-scale infrastructures. Instead, it targets reproducible, project-level empirical studies in which researchers need linked GitHub artifacts, time-aware process and graph metrics, CI evidence, candidate defect-provenance relations and analysis-ready datasets. The key idea is to separate ingestion from analysis: once the database is mined, multiple issue-level, pull-request-level, commit-level, or file-level datasets can be defined using SQL.

3 QModel Framework

QModel represents a GitHub repository as a linked empirical database rather than as a collection of independent artifacts. Its architecture comprises an ingestion framework and a compilation and analysis framework, connected through a project-centric relational

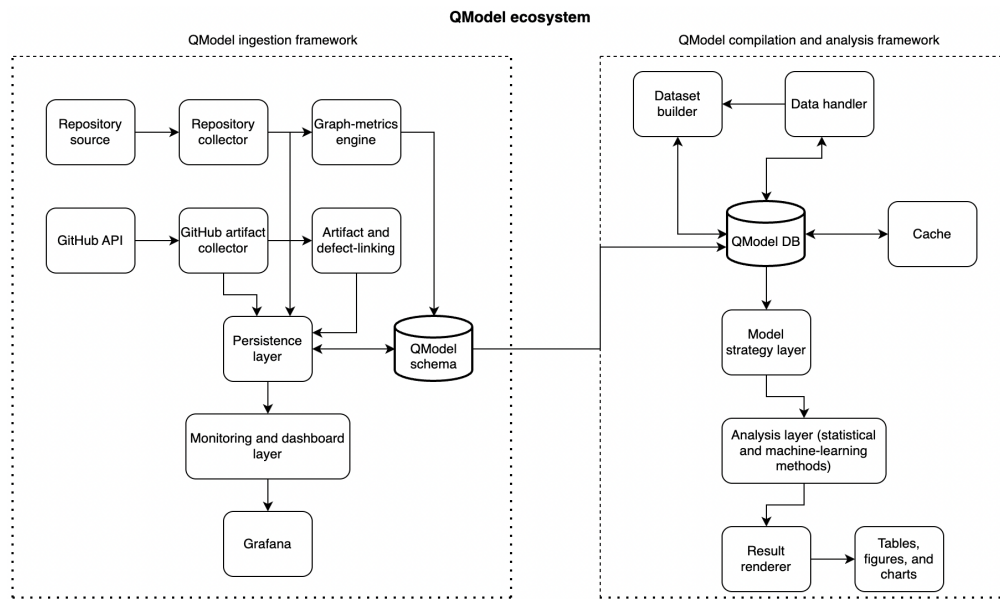
■ **Table 1** Comparison with MSR infrastructures for linked, quality-oriented GitHub analyses.

Tool	Native abstraction	Native SZZ evidence	Native linked quality-analysis object
PyDriller [15]	Git history, commits, file changes and evolution metrics.	No.	No native GitHub issue/PR/timeline/reaction/CI/SZZ linking; external integration is needed.
GHTorrent [6]	GitHub metadata for projects, users, issues, pull requests and comments.	No.	Partial GitHub metadata, but no analysis-ready fixing commits, CI evidence, or candidate-BIC relations.
Boa [3]	Large-scale repository mining through a domain-specific language.	No.	No native GitHub-centered quality-linking schema for PR-issue-CI-defect provenance objects.
World of Code [12]	Large-scale VCS object graph for commits, projects, files, authors and blobs.	No.	No native GitHub social-artifact layer for PRs, issues, reactions, timelines, or CI evidence.
GrimoireLab	Multi-forge retrieval, enrichment, identity management, indexes and dashboards [2].	No.	Cross-source enriched indexes, but no feature-target SQL compiler.
SmartSHARK	VCS, issue, PR/review, CI, code metrics and validated bug evidence [16].	Yes.	No dedicated feature-target SQL compiler.
QModel	Project-centric database integrating Git, GitHub, CI, timelines, reactions, graph/churn metrics and defect evidence.	Yes.	Yes; linked artifacts and candidate-BIC relations are stored as explicit queryable relations.

schema (Figure 1). The ingestion framework obtains version-control data from the repository and development artifacts through the GitHub API. It computes commit-graph metrics, reconstructs cross-artifact and candidate-defect links and stores both collected and derived data in the schema. The compilation framework uses the persisted data to materialize SQL-defined datasets and manage reusable intermediate data. The analysis layer selects an implementation from a registry keyed by the configured strategy name and executes it on the materialized dataset. The registered strategies include correlation analysis, PCA, ANOVA, linear regression, random forest and elastic net regression. The resulting outputs are then passed to the result renderer.

3.1 Design Principles

QModel is designed around three principles. First, cross-artifact evidence is stored explicitly: pull requests, commits, issues, timelines, reactions, CI runs, file changes, graph metrics and candidate-defect links can be joined rather than reconstructed. Second, repository history is time-aware: QModel stores the commit DAG and derives graph and branch/merge metrics that describe where a change occurs in history. Third, empirical operationalization is SQL-defined: researchers can define analysis rows, targets, joins, filters and feature aggregations over the schema without changing the mining pipeline.



■ **Figure 1** QModel architecture and principal data flows.

3.2 Ingestion Pipeline

The ingestion pipeline collects both Git and GitHub evidence.

- The repository collector clones a target repository and extracts commits, parent relations, file changes, changed lines, patches, author metadata and commit timestamps.
- The GitHub artifact collector retrieves issues, pull requests, labels, assignees, reviewers, timeline events, reactions and check runs.
- The graph-metrics engine reconstructs the commit directed acyclic graph and computes time-aware graph and branch/merge metrics.
- The linking layer connects issues, pull requests, commits, timelines, reactions and CI evidence using explicit association schema.
- The defect-linking component links issues to fixing pull requests and applies SZZ-style tracing to identify candidate bug-introducing commits.
- The persistence layer stores raw artifacts, derived metrics and cross-artifact relations in a normalized relational schema.

The result is not only a raw mirror of repository data. It is a database of analysis-ready empirical evidence. For example, a pull request can be connected to its commits, commit-level churn, CI outcomes, reviewer activity, linked issues, issue timelines and candidate defect-provenance relations without re-running the mining process.

3.3 Relational Schema

The central schema entity is a project, identified by repository owner and name. Around this entity, QModel organizes repository evidence into relational families for version-control history, GitHub collaboration artifacts, continuous-integration records, derived history metrics and defect-provenance relations. The full entity-relationship diagram and SQL schema definition are included in the supplementary artifact; the main paper summarizes the schema at the level needed to explain the empirical workflow.

■ **Listing 1** Abridged SQL definition of the issue-level candidate-BIC dataset.

```
SELECT pi.project_owner, pi.project_name,
pi.id AS analysis_id,
TIMESTAMPDIFF(HOUR, pi.created_at, pi.closed_at)
AS target_hours,
AVG(c.distance_to_branch_start) AS avg_fp_distance,
SUM(cc.total_changes) AS total_changes
FROM project_issue pi
JOIN project_issue_bug_introducing_commits pibic
ON pibic.project_issue_id = pi.id
AND pibic.project_issue_project_name = pi.project_name
AND pibic.project_issue_project_owner = pi.project_owner
JOIN commit c
ON c.sha = pibic.bug_introducing_commits_sha
LEFT JOIN commit_churn cc ON cc.sha = c.sha
WHERE pi.state = 'closed'
AND pi.created_at IS NOT NULL
AND pi.closed_at IS NOT NULL
GROUP BY pi.project_owner, pi.project_name, pi.id,
pi.created_at, pi.closed_at;
```

The schema is designed around analysis joins rather than around isolated raw tables. Version-control relations preserve commits, parent structure, file changes, changed lines and graph metrics. GitHub relations preserve issue and pull-request metadata together with labels, assignees, reviewers, timelines, reactions and cross-links among artifacts. Defect-provenance relations connect reported issues to fixing evidence and to SZZ-style candidate bug-introducing commits. As a result, technical evidence, process evidence, social activity, and candidate defect provenance can be assembled in one SQL-defined analysis row.

The unit of analysis is intentionally not fixed by the schema. Depending on the study, a row can represent a pull request, issue, commit, file, developer, time window, release interval, or custom cross-artifact object. This design lets researchers reuse the same mined database while changing the empirical operationalization: for example, one query can construct pull-request rows with review, CI, churn and history context, while another can construct issue rows focused on resolution time, fixing evidence and candidate defect provenance.

3.4 SQL-to-Dataset Compilation

QModel Compilation separates dataset definition from statistical analysis. A researcher defines a dataset using SQL over the `qmodel` schema. The SQL query specifies the unit of analysis, target variable, predictors, filters and aggregation logic. The resulting table can then be passed to descriptive statistics, correlation analysis, regression, random forests, or other analysis strategies.

Listing 1 illustrates how these decisions are encoded in SQL. The excerpt is taken from the definition of the issue-level candidate-BIC dataset used in the pilot evaluation.

The query makes the operationalization directly inspectable: `project_issue` establishes the issue as the analysis unit, `target_hours` defines the resolution-time target and the candidate-BIC relation specifies the path from issues to candidate bug-introducing commits. The `WHERE` clause defines the eligible population, while `AVG` and `SUM` aggregate commit-graph and churn measures into one row per issue. In the complete query, `commit_churn` is defined

as a common table expression (CTE), which is a temporary named result available within the query. It aggregates file-change measures by commit and is joined to the main query through the alias `cc`. Its definition is omitted from the listing for brevity. The complete executable query is provided in the replication artifact.

4 Time-Aware Metrics and Defect Linking

QModel computes derived variables that describe not only what changed, but also where and when the change occurred in the repository history. This section summarizes the main derived evidence used later in the evaluation: commit-history metrics, churn and CI summaries, process metrics and SZZ-style candidate defect links.

4.1 Commit-History Metrics

QModel therefore stores the full commit DAG and derives a deterministic auxiliary structure called a continuity line. This structure is used only to compute path-based metrics such as distance to line start, upstream integration, and days since last merge; the original commit DAG is still preserved.

For a commit with one parent, the parent is used as the continuation parent. For a merge commit with multiple parents, QModel scores candidate parents and selects the parent that continues the longest visible non-branching line. Ties are broken deterministically using commit timestamp and SHA. The resulting metric `distance_to_branch_start` should therefore be interpreted as distance to the start of the selected continuity line, not as a verified distance to the creation point of a developer-intended branch.

QModel also stores minimum and maximum depth in the commit history, degree statistics, branch visibility from references available at mining time and days since the previous merge on the selected continuity line. These metrics are not intended to replace standard churn or process measures. They provide additional history context that can be joined with commits, pull requests, issues, files, or any other SQL-defined analysis object.

4.2 SZZ-Style Candidate Defect Links

QModel implements an SZZ-style defect-linking procedure based on prior work [1]. We do not introduce a new SZZ algorithm. Instead, QModel uses SZZ-style tracing as an infrastructure component and stores the resulting links as explicit relations in the database.

The stored links are interpreted as *candidate* bug-introducing commits, not as ground truth. This distinction is important because SZZ-style links can be affected by incomplete issue-fix links, refactorings, code movement, squash/rebase workflows and limitations of line-based blame. QModel therefore keeps these relations queryable and inspectable rather than treating them as validated defect labels.

In the issue-level datasets, a candidate BIC is an earlier commit linked to the fixing evidence for that issue by the SZZ-style procedure. In the pull-request-level datasets, a candidate-BIC indicator means that at least one commit contained in the pull request later appears in an issue-level candidate-BIC relation. Thus, the pull-request-level variable does not label the pull request as definitively defective; it records later defect-provenance evidence associated with commits contained in that PR.

5 Early Evaluation Design

The early evaluation assesses QModel as research infrastructure. The objective is not to establish universal laws about software quality. Instead, the study asks whether QModel can extract and link heterogeneous GitHub artifacts, compute derived metrics and support SQL-defined analysis contexts over the same persisted repository data.

5.1 Subject Systems

Repositories were selected in two steps. First, they had to provide the raw artifacts required by the evaluation: an extended commit history, sustained pull-request and issue activity, metadata from which cross-artifact links could be constructed and CI records. We favored long-running projects because their histories span changes in repository-management practices and platform support, including the adoption of pull-request workflows, CI services and GitHub Checks. This temporal variation allows QModel to be exercised on both recent, data-rich periods and older periods in which some artifact types may be unavailable.

The eligibility criteria did not require the datasets produced by QModel to be complete. In particular, repositories were not selected according to the availability of derived commit-DAG or churn measures, because dataset completeness is the outcome evaluated by RQ1.

Second, we selected repositories from different development contexts. `ansible/ansible` is a Python-based infrastructure-automation project, whereas `facebook/react` is a JavaScript-based user-interface library. This contrast allows the pilot to examine QModel across different application domains, programming ecosystems and repository practices.

The two repositories are pilot cases and are not intended to represent the GitHub population. A broader evaluation should define an eligible repository population and sample projects across relevant characteristics, including ecosystem, project size and age, merge strategy, CI configuration and issue-linking practice.

Across the two projects, QModel extracted 76,475 commits, 70,509 pull requests, 45,908 issues, 2,174,020 timeline events, 315,235 file-change records and 446,161 CI records. The number of CI records should not be interpreted as the number of CI-covered commits. A commit may generate several check runs, whereas older commits may predate GitHub Checks or rely on external CI systems.

5.2 Operationalization

For RQ1, we evaluate whether QModel can transform persisted repository evidence into analysis-ready datasets. The evaluation focuses on candidate-BIC datasets because they require several linked evidence paths at once: issues or pull requests, fixing evidence, candidate bug-introducing commits, graph-history metrics and churn summaries. We measure whether each analysis row has a computable target variable, graph summary, churn summary and combined graph+churn evidence. Issue-level candidate-BIC rows test whether QModel can construct defect-provenance datasets from issues and their associated candidate introducing commits. Pull-request-level candidate-BIC rows test whether QModel can identify PRs containing commits that later appear in issue-level candidate-BIC relations.

For RQ2, we instantiate a pull-request-level dataset as a worked empirical analysis. The target is $\log(1 + \text{review_hours})$, where review time is derived from pull-request creation and merge timestamps. Predictors include timeline-event counts, reviewer counts, assignee counts, reaction counts, graph summaries over PR commits, churn summaries, CI summaries and

■ **Table 2** RQ1 derived-metric computability for SQL-defined analysis datasets. Each metric cell reports rows with computable evidence and the corresponding percentage of analysis rows.

Project	Dataset	Rows		Target	Graph	All graph	Churn	Graph+Churn
ansible	Issue-level candidate-BIC	2,001	2,001 (100.00%)	1,995 (99.70%)	1,704 (85.16%)	1,995 (99.70%)	1,995 (99.70%)	1,995 (99.70%)
ansible	Issue-level fixing-commit	6,323	6,323 (100.00%)	1,115 (17.63%)	373 (5.90%)	1,093 (17.29%)	1,093 (17.29%)	1,093 (17.29%)
ansible	PR-level candidate-BIC	3,680	3,680 (100.00%)	3,680 (100.00%)	3,680 (100.00%)	3,680 (100.00%)	3,680 (100.00%)	3,680 (100.00%)
react	Issue-level candidate-BIC	724	724 (100.00%)	724 (100.00%)	724 (100.00%)	724 (100.00%)	724 (100.00%)	724 (100.00%)
react	Issue-level fixing-commit	730	730 (100.00%)	728 (99.73%)	728 (99.73%)	244 (33.42%)	244 (33.42%)	244 (33.42%)
react	PR-level candidate-BIC	4,720	4,720 (100.00%)	4,720 (100.00%)	4,720 (100.00%)	4,720 (100.00%)	4,720 (100.00%)	4,720 (100.00%)

indicators or counts related to candidate defect provenance. The model is used only as a feasibility demonstration of SQL-defined analysis construction, not as a claim of state-of-the-art prediction performance.

6 Early Results

This section reports results for two questions: whether QModel materializes datasets with computable derived metrics and whether they support empirical analysis.

6.1 RQ1: Derived Metric Computability

RQ1 asks whether QModel can transform mined repository evidence into analysis rows where the target variable and derived metrics are available together. We evaluate three SQL-defined datasets: issue-level candidate-BIC rows, issue-level fixing-commit rows and PR-level candidate-BIC rows. These datasets exercise different evidence paths. Candidate-BIC datasets require defect- provenance links, graph-history metrics and churn summaries. Fixing-commit datasets test the fixing side of issue evidence. PR-level candidate-BIC rows test whether pull requests containing later candidate-BIC commits can be materialized as analysis objects.

Table 2 reports the computability results. The reported columns are row-level availability measures. *Target* means that the row has a non-null, non-negative duration: issue creation-to-closure time for issue-level rows and pull-request creation-to-merge time for PR-level rows. *Graph* means that at least one commit in the row has the complete graph metric vector used by the query, including degree, history-depth, branch-distance, upstream-head, merge-recency, branch-count and average-degree fields. *All graph* is stricter: every commit associated with the row must have that complete graph metric vector. *Churn* means that at least one commit in the row has a file-change summary with non-null total changes and at least one changed file. *Graph+Churn* means that the same analysis row has at least one graph-ready commit and at least one churn-ready commit.

The target duration is complete in all six project-dataset combinations. Candidate-BIC datasets are highly computable. In **ansible/ansible**, 1,995 of 2,001 issue-level candidate-BIC rows have both graph and churn summaries. In **facebook/react**, all 724 issue-level candidate-BIC rows have both graph and churn summaries. At PR level, both projects are fully computable: all 3,680 Ansible rows and all 4,720 React rows have graph, churn, and combined graph+churn evidence.

The fixing-commit datasets reveal a different evidence boundary. Although their target durations are complete, graph and churn computability is lower for **ansible/ansible**: 1,093 of 6,323 rows have both graph and churn summaries. The stricter all-graph measure is lower still, at 373 of 6,323 rows (5.90%). This value is expected because the all-graph measure requires every fixing commit associated with an issue to have the complete graph-metric

vector. For historical GitHub data, this can fail when commits are associated with branches or integration paths that are no longer visible at mining time, when branch references have been deleted, or when some graph-derived quantities cannot be reconstructed consistently.

For `facebook/react`, graph metrics are available for nearly all fixing-commit rows, but churn summaries are available for 244 of 730 rows. This shows that fixing evidence, graph evidence and churn evidence are distinct forms of empirical availability: an issue may have fixing commits, while the corresponding local history or file-change evidence needed for metric computation is incomplete.

6.2 RQ2: Worked Example Analysis Instantiation

RQ2 asks whether the same mined database can instantiate a concrete empirical analysis. We use pull-request review time as a worked example because it requires multiple linked evidence sources: PR metadata, commits, graph-history metrics, changed files, CI outcomes, review activity, timeline events, reactions and candidate-defect evidence. The goal is methodological rather than predictive: review time is not a direct quality label and the model is not intended as a universal benchmark for review latency.

Dataset Construction

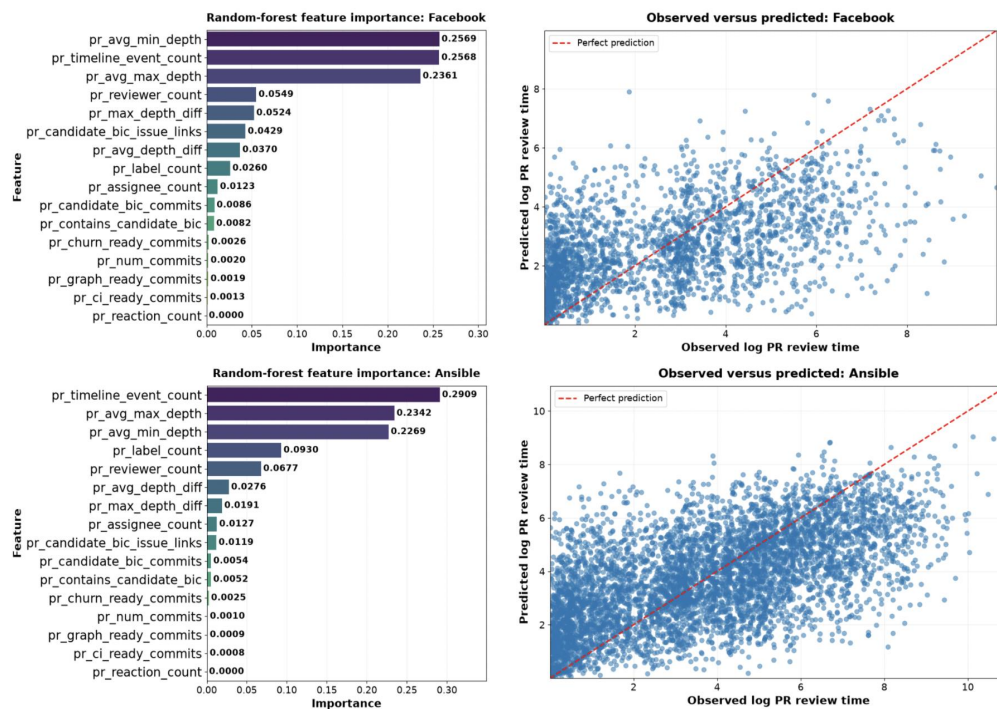
The unit of analysis is a merged pull request with at least one associated commit. Review time is computed as `TIMESTAMPDIFF(SECOND, created_at, merged_at)/3600` and the modeled target is `log(1 + pr_review_hours)`. The transformation limits the influence of extremely long-lived pull requests but does not make review time a quality label. The SQL workflow aggregates commit-level evidence into six PR-level feature groups: process and social activity, commit counts, graph-history metrics, churn, CI summaries and candidate-defect indicators. Each row is therefore a linked empirical object assembled from multiple repository artifacts rather than raw pull-request metadata.

Exploratory modeling setup and baseline comparison

We evaluate whether the SQL-materialized PR-level dataset supports a standard modeling workflow. The target is `log_pr_review_hours`. The SQL query assigns pull requests to a stable deterministic hash-based hold-out split: rows labeled `train` are used for fitting and rows labeled `validation` are used only for evaluation. The retrospective covariates cover process and social activity, evidence counts, graph-history metrics, churn, CI summaries and candidate-defect indicators.

We compare a `RandomForestRegressor` with an ordinary least-squares linear-regression baseline using the same SQL-materialized features and complete-case training and validation rows. The random forest uses 200 trees, random seed 42 and no maximum-depth restriction. The linear baseline includes an intercept, with features standardized using training-set statistics only. Both models are evaluated on the held-out validation rows using R^2 and MSE.

The random forest performs modestly better than the linear baseline. For `ansible/ansible`, it obtains $R^2 = 0.3521$ and $\text{MSE} = 3.7919$, compared with $R^2 = 0.3061$ and $\text{MSE} = 4.0607$ for linear regression. For `facebook/react`, the corresponding results are $R^2 = 0.2265$ and $\text{MSE} = 3.5550$, compared with $R^2 = 0.1977$ and $\text{MSE} = 3.6874$. Thus, the random forest increases held-out R^2 by 0.0460 and 0.0288 and reduces MSE by approximately 6.6% and 3.6%, respectively. These modest improvements suggest some nonlinear signal but do not establish a competitive review-time predictor. Instead, the comparison shows that QModel can instantiate a reproducible analysis and evaluate it against a simpler baseline.



■ **Figure 2** Exploratory random-forest diagnostics for PR-level review time.

Figure 2 presents the random-forest diagnostics. In both projects, process and graph-history covariates are more influential than CI/action covariates. The actual-versus-predicted plots show positive trends with substantial dispersion, indicating useful structure but limited pull-request-level precision.

Roadmap

We will evaluate QModel across repositories with different sizes, branching practices, linking conventions and CI histories. Planned extensions include evidence-quality indicators for link confidence, graph/churn completeness and workflows such as squash/rebase, deleted branches, refactorings, renames and formatting-only changes. Reusable SQL templates will support alternative analysis units, targets, filters and features, enabling studies of whether conclusions remain stable across projects and operationalization choices.

7 Threats to Validity

The early evaluation has several validity threats. For construct validity, QModel does not claim that review time, CI outcomes, or SZZ-style links are direct measures of software quality. Review time is a retrospective process measure and candidate bug-introducing commits are noisy evidence rather than ground-truth defect labels. Similarly, CI/action records indicate available repository evidence, not complete CI coverage for every historical commit.

For internal validity, the computed datasets depend on issue-pull-request links, fixing-commit identification, commit-to-PR mappings, line-based blame, and GitHub metadata available at mining time. Squash and rebase workflows, deleted branches, refactorings, code

movement, formatting-only changes, missing check runs and incomplete issue references can affect both defect-provenance relations and derived commit-DAG or churn summaries. QModel mitigates this threat by storing such evidence as explicit, inspectable relations and by reporting computability gaps instead of silently discarding them.

For external validity, we purposefully selected `ansible/ansible` and `facebook/react` because they provide long commit histories, sustained pull-request and issue activity, linkable cross-artifact metadata and CI records. They also represent different application domains and programming ecosystems, but two projects are not a representative sample. Results may differ for smaller or non-GitHub projects and repositories with different branching, issue-linking, or CI practices.

8 Conclusion

This paper presented QModel, an emerging infrastructure for reproducible, time-aware GitHub mining in empirical software quality studies. QModel persists commits, issues, pull requests, file changes, reactions, timelines, CI check runs, graph metrics and SZZ-style candidate defect links in a unified relational schema. Its SQL-to-dataset layer allows researchers to define units of analysis, target variables, feature aggregations and filtering criteria explicitly.

The early evaluation on `ansible/ansible` and `facebook/react` shows that QModel can support reusable pull-request-level datasets and expose coverage limitations for issue-level and CI-based analyses. The worked review-time analysis demonstrates feasibility rather than final predictive performance. The main contribution is therefore infrastructural: QModel makes quality-oriented MSR operationalizations inspectable, reusable and suitable for systematic empirical evaluation. By separating repository ingestion from SQL-defined dataset construction, QModel provides a foundation for comparing alternative quality operationalizations over the same persisted evidence.

Data Availability Statement

The replication artifact is available on Zenodo at <https://zenodo.org/records/21737012>. It contains snapshots of the QModel mining framework, which collects Git and GitHub evidence and persists the relational database and the QModel Compilation repository, which provides the SQL queries, analysis scripts, derived result tables, generated figures and README files needed to reproduce the reported analyses.

For RQ1, `queries/RQ1.sql` constructs the issue-level fixing-commit, issue-level candidate-BIC and pull-request-level candidate-BIC computability datasets in Table 2. For RQ2, the main entry point is `clients/proven/rq2/RQ2_models.py`, together with the SQL query that materializes the pull-request-level review-time dataset. Derived outputs are in the `results/` directory of the `qmodel_compilation` repository. The artifact also documents the expected database schema and data-processing steps used to construct the SQL-defined datasets.

The artifact prioritizes executable and derived materials rather than the full mined database, which is approximately 500 GB before compression and contains GitHub metadata, Git history, patches, CI/action records, timelines and raw text/JSON payloads. The database can be regenerated using the mining framework and documented configuration; a compressed dump may be provided separately when storage and review logistics permit.

References

- 1 Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. SZZ unleashed: An open implementation of the SZZ algorithm. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, pages 7–12, 2019. doi:10.1145/3340482.3342742.
- 2 Santiago Dueñas, Valerio Cosentino, Jesus Gonzalez-Barahona, Alvaro Felix, Daniel Izquierdo-Cortazar, Luis Cañas-Díaz, and Alberto Pérez García-Plaza. Grimoirelab: A toolset for software development analytics. *PeerJ Computer Science*, 7:e601, July 2021. doi:10.7717/peerj-cs.601.
- 3 Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering*, pages 422–431, 2013. doi:10.1109/ICSE.2013.6606588.
- 4 Samuel W. Flint, Jigyasa Chauhan, and Robert Dyer. Pitfalls and guidelines for using time-based git data, 2022. doi:10.48550/arXiv.2209.04511.
- 5 Jesus Gonzalez-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17:75–89, February 2012. doi:10.1007/s10664-011-9181-9.
- 6 Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 233–236, 2013. doi:10.1109/MSR.2013.6624034.
- 7 Georgios Gousios and Andy Zaidman. A dataset for pull-based development research. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 368–371, 2014. doi:10.1145/2597073.2597122.
- 8 Steffen Herbold, Alexander Trautsch, Fabian Trautsch, and Benjamin Ledel. Problems with szz and features: An empirical study of the state of practice of defect prediction data collection. *Empirical Software Engineering*, 27(2):42, 2022. doi:10.1007/s10664-021-10092-4.
- 9 Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, 2016. doi:10.1007/s10664-015-9393-5.
- 10 Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi:10.1109/TSE.2012.70.
- 11 Masanari Kondo, Yutaro Kashiwa, Yasutaka Kamei, and Osamu Mizuno. An empirical study of issue-link algorithms: which issue-link algorithms should we use? *Empirical Software Engineering*, 27(6):136, 2022. doi:10.1007/s10664-022-10120-x.
- 12 Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretski, and Audris Mockus. World of code: An infrastructure for mining the universe of open source vcs data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 143–154, 2019. doi:10.1109/MSR.2019.00031.
- 13 Gregorio Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 171–180, 2010. doi:10.1109/MSR.2010.5463348.
- 14 Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M. González-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology*, 99:164–176, 2018. doi:10.1016/j.infsof.2018.03.009.
- 15 Davide Spadini, Mauricio Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911, 2018. doi:10.1145/3236024.3264598.

- 16 Alexander Trautsch, Fabian Trautsch, Steffen Herbold, Benjamin Ledel, and Jens Grabowski. The smartshark ecosystem for software repository mining, 2020. [arXiv:2001.01606](#).
- 17 Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering*, pages 356–366, 2014. doi:10.1145/2568225.2568315.
- 18 M. Vidoni. A systematic process for mining software repositories: Results from a systematic literature review. *Information and Software Technology*, 144:106791, 2022. doi:10.1016/j.infsof.2021.106791.