Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Mining Two Decades of Repository History to Predict Software Complexity Accumulation: A Leakage-Aware Landmark Survival Analysis

Moatasem M. Draz

moatasem.draz@fci.kfs.edu.eg

Kafrelsheikh University

Method Article

Keywords: Repository Mining, Data-Driven Software Analytics, Survival Analysis, Landmark Design, Software Complexity, Leakage-Aware Modeling

Posted Date: November 13th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10468453/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Mining Two Decades of Repository History to Predict Software Complexity Accumulation: A Leakage-Aware Landmark Survival Analysis

Moatasem M. Draz

Information Systems Department, Faculty of Computers and Information, Kafrelsheikh University, Kafr El-Sheikh, Egypt

(*Corresponding author: Moatasem M. Draz. Email: moatasem.draz@fci.kfs.edu.eg)

Abstract

Context: Software repositories accumulate decades of fine-grained change history, yet the analytic value of that history remains underexploited: complexity metrics are widely used to flag risky code, but nearly all empirical evaluations rely on static snapshots, so it remains unknown whether early measurements mined from a file's first commits predict how much complexity it will subsequently accumulate.

Objective: Complexity accumulation is modelled as a time-to-event process to determine (RQ1) which signals observable in a file's earliest commits predict how soon its complexity grows substantially beyond its initial level, (RQ2) whether richer structural information improves such forecasts beyond early process dynamics, and (RQ3) whether the resulting model generalises to projects unseen during training.

Method: An automated mining pipeline traversed fourteen Python open-source systems in chronological order and extracted 36,076 file-level commit observations spanning 2005–2026, yielding 1,291 per-file trajectories. A relative growth event, defined as a file's maximum cyclomatic complexity rising δ units above its own initial value, is combined with a landmark design in which every predictor is measured only within a file's first three commits. Together these eliminate event tautology and outcome leakage by construction, and two automated guards halt the pipeline if either property is violated. Penalized Cox proportional-hazards models were then evaluated by repeated cross-validation with paired confidence intervals against reimplemented size, static-complexity, coupling, and process baselines.

Results: A concordance of 0.663 (95% CI 0.626 – 0.704) was achieved, which is +0.086 (+0.023 to +0.131) higher than the best baseline and, in eight configurations out of nine, was significantly higher. Initial static complexity barely exceeded chance (0.541). Neither structural covariates (+0.014) nor gradient boosting (+0.021) produced supported gains, and leave-one-project-out concordance fell to 0.599.

Conclusion: Complexity accumulation is governed by early development dynamics rather than by initial code structure. Within-corpus forecasts are reliable, whereas cross-project transfer is weak and calls for per-organisation calibration. A fully automated, guard-checked mining and analysis pipeline is released to support replication.

Keywords: Repository Mining; Data-Driven Software Analytics; Survival Analysis; Landmark Design; Software Complexity; Leakage-Aware Modeling

1. Introduction

Complexity measures have been in software engineering's toolbox for 50 years. For cyclomatic complexity [1] and the software science measures [2], these continue to be part of review gates, quality dashboards, and static analysis tools, and the object-oriented suite [3] extended it to design level attributes. However, the legitimacy of these measures has been questioned as long as they have been around: some popular measures did not obey intuitive rules [4] and cyclomatic complexity was deemed to be largely a measure of size [5] while the statistical foundations of metric-based defect prediction was critiqued in detail [6].

Almost all of this discussion is about the meaning of a metric at a given moment. But modern repositories are living systems: files are created small and simple, and some of them, and only some of them, are "hotspots" of branching logic, nesting, and coupling that dominate the cost of maintenance. This dynamic perspective was also predicted by the laws of software evolution years ago [7] and later software aging [8] and code decay [9] have been described at the system level. But what the literature does not provide is a weekly answer to this question, which is asked at file granularity: When a file is young, is it likely to add a lot of extra complexity, and if so, what kind? To answer the question, it's not the snapshot, it's the trajectory that needs to be modeled.

This paper treats complexity accumulation as a time-to-event problem, and uses survival methods [10, 11] to analyze it. A file has experienced an event if the highest function-level cyclomatic complexity has increased by a fixed amount from the initial complexity of the file; those files that have not experienced an event are right-censored. This is a deliberate choice for the definition of a relative event: When using an absolute threshold (complexity ≥ 10), the event indicator is shown to be correlated with initial complexity at 0.61, which makes “prediction” merely a tautology (what are complex files if not complex files that begin complex?). Under the relative definition the correlation falls to 0.063, so the model must forecast growth rather than merely restate the starting point.

Another methodological issue is information leakage. Bias families that were summed across all of the history of a file integrate information after the event and are documented in the epidemiology literature as immortal time bias [12] and the bias from covariates as rates over the observation window encodes the outcome in the denominator (in the present data, the correlation between observation span and survival time is exactly 1.00). The construction of a landmark design removes both hazards: the size of the window is fixed for each file, and the survival clock is restarted at the time the window closes, at which point the hazard is measured. The pipeline is stopped and a warning message sent if either of these properties is violated by two automated checks: an event-tautology check and a covariate-circularity check.

Fourteen mature open-source Python projects are evaluated (36,076 file-level commit observations from 2005 to 2026; 1,291 analyzable trajectories, 387 events). The Cox proportional-hazards trajectory model is compared to reimplemented baseline models from the main predictor families of previous work: size [13], static complexity models [1, 2], coupling models, and process metrics [14, 15]. The metric used for the models to calculate scores is the same (Harrell's concordance index [16, 17]) and is computed using repeated stratified cross-validation with paired bootstrap confidence intervals.

There are five contributions:

- A study design that is leakage-free and guard-checked, using a relative event definition and covariate measured to the best of current knowledge at the trajectory level, the first use of landmarking to source-code evolution.

- An empirical study, using the same data set and using the same metric, that: early process dynamics predict complexity accumulation (concordance 0.663), which extends the process-over-product finding that other defect prediction studies have discovered so far [14, 15] to another outcome; initial static complexity correlates only slightly above chance (0.541) to complex processes.
- A nonlinear gradient-boosted survival model [18] and twelve additional structural covariates do not provide a statistically supported improvement, only two of which are disciplined.
- An honest generalization assessment: leave one project out validation leads to an assessment value of 0.599, quantifying the amount of the within-corpus signal that is transferred to unseen projects.
- Relevant to the growing body of work on AI-assisted development [19, 20], incidental finding is that the AI-authorship attribution through commit trailers was zero-hit in this corpus (20,014 commits).

The rest of the paper is organized as follows: The related work is summarized and gap is stated in section 2. The study design is described in Section 3, which contains the two design hazards and their solutions. The results of the three research questions are presented in Section 4. Implications, situating the findings against prior studies, threats to validity analysis and conclusions are included in Sections 5, 6, and 7, respectively.

2. Background and Related Work

The recent studies are divided into six streams: validity of complexity measures (2.1); using code and process data to predict quality outcomes (2.2); longitudinal studies of decay, smells and technical debt (2.3); and the growing body of research on AI-generated code (2.5); survival analysis in software engineering (2.4). Section 2.6 summarizes the gap in a tabular format and make the general statements that drive this study.

2.1 Complexity metrics and their contested validity

The classical metrics were suggested as metrics of comprehensibility and testing effort [1, 2] but were criticized early on due to their lack of construct validity: widely used metrics do not satisfy basic principles [4] and size correlates with cyclomatic complexity so highly that it is not certain that the increment adds value [5]. The object-oriented suite [3] was proven to be a fault-proneness indicator (FPI) [21, 22] but it was noted that this validation process should be regarded as a correlation and not a causal test [6]. A second wave re-grounded complexity in human comprehension, the reader scores were used to learn readability metrics [23] and these were later simplified [24]; the understandability was automatically measured by a complex combination of code, documentation and developer features [25]; 40 years of readability studies by human readers were systematically reviewed [26]. Cognitive Complexity [27] had been validated to be statistically significantly [28] related to ten cognitive datasets of comprehenders, and fMRI evidence suggested that the existing metrics account for only a part of the cognitive load for programmers [29]. As a group, this stream confirms that snapshot complexity is at best a weak proxy for the phenomena practitioners are interested in motivating the shift from – pursued here – the level of complexity to its accumulation over time.

2.2 Predicting quality outcomes: product versus process signals

The most obvious precedent for the current study's main result is the work of defect prediction. First, faults were forecast based on properties of the product itself [30, 31] and then on the size and structure of the module [21, 22]. It was shown that process signals are more successful predictors: change history was found to be superior to size and structure [32]; defect density was predicted from relative churn [33]; change metrics were shown to outperform static code attributes [14]; the entropy of changes was quantified as a predictor [34]; the temporal locality of faults was exploited [35]; and a systematic account was given of how, and why, process metrics are better [15]. The just-in-time tradition brought the signal of prediction to commit granularity [36, 37, 38] and benchmarks and reviews focused the evidence base [39, 40]. Whether the process-over-product regularity extends to a different outcome complexity growth itself had not been tested; it is tested here. Taken together, these limitations motivate a leakage-aware, trajectory-based framework that forecasts future complexity accumulation rather than merely describing past observations.

2.3 Longitudinal decay, code smells, and technical debt

The "laws of software evolution" [7] and the "software-aging argument" [8] laid the foundations of the evolutionary view, code decay was operationalized by taking change-management data [9] and super-linear growth has been scientifically documented in open-source [41]. A snapshot-versus-trajectory nuance that directly influences the present design was found in the smell literature, with most smells being introduced at artifact creation time [42] and smell evolution, lifespan, diffuseness and maintenance-effort being studied in [43, 44, 45]. The technical-debt community evolved over a period of time, from the founding metaphor [46] to its consolidation [47, 48] and, as a result, curated longitudinal datasets [49]. These studies report on the accumulation; predicting at file granularity with a censoring aware model was left open.

2.4 Survival analysis in software engineering

Along with a well-developed tool kit for model checking [50, 51] and discrimination assessment [16, 17, 52] there also exists a good tool box of machine learning extensions [18, 53, 54] and open-source code [55, 56] for time-to-event methods. Their use is relatively limited, but expanding in software engineering: Cox models underpinned the theory of relative defect proneness and the size–defect relationship [57, 13]; smell lifespans were measured with survival mining [43]; the time until vulnerable dependencies are remediated was recently modelled with Kaplan–Meier and Cox estimators [58]; and, most recently, the survival of individual source-code lines was predicted with AST-aware mining [59]. None of these efforts combines a relative growth event with landmark predictor measurement [60, 61]. The landmark technique, developed to avoid guarantee-time and immortal-time biases in clinical studies [62, 12], transfers naturally to repository mining, where the same biases arise whenever predictors are aggregated over a file's full history.

2.5 AI-generated code and complexity

As code-generation assistants become more common, the issue of how AI-created changes add up in complexity comes into focus. Productivity benefits of GitHub Copilot have been measured by telemetry [19, 63] and controlled studies, industry analysts have cautioned about increased churn and duplication [64], security issues have been reported in the code created by GitHub Copilot in real GitHub projects [20], benchmark studies have compared the static quality of code generated by LLM to the static quality of human code [65, 66] and related performance to the complexity of

the problem [67]. Recent secondary studies consolidate the field: a broad survey of LLMs for code generation [68], and a 2026 review of non-functional quality characteristics of generated code, which concludes that maintainability outcomes depend strongly on task complexity and evaluation method [69]. There are conflicting results on the complexity differences between languages and models, and almost all of the results are based on snapshot comparisons of short benchmark programs. A signal based on the commit trailers was intended to be used in the present study but, as reported in section 4.4, it was not found at all (0 of 20,014 commits) and that is a methodological result of interest for this stream.

2.6 The gap

Table 1 matches representative studies with four properties: outcome predicted, method, explicit control of leakage between the covariates and the outcome, and dynamic (models evolution over time) or static (snapshot). Three common restrictions arise. First, complexity metrics are considered in static snapshots, which neglects the trajectory level prediction. Second, longitudinal studies of decay and debt are descriptive, not predictive, and most of the files in any window of time have not (yet) suffered degradation, but longitudinal studies do not account for the presence of censoring. Third, if studies are conducted to predict evolution-related outcomes, they generally take a time-aggregated approach to covariates across the entire history, introducing immortal-time-style leakage into the estimation of accuracy, and leakage is rarely reported explicitly (Table 1).

Table 1. Comparison of Existing Approaches for Complexity Prediction

Study	Predicts	Method	Leakage-controlled?	Dynamic?
Static gates [1]	current complexity	threshold on snapshot		no
Gyimóthy et al. [22]	fault-proneness	OO metrics + ML	not reported	no
Moser et al. [14]	defects	change metrics + classifiers	not reported	partial
Rahman & Devanbu [15]	defects	process vs. product features	not reported	partial
Kamei et al. [36]	defect-inducing commits	JIT commit features	partial	yes
Koru et al. [13]	defect timing	Cox survival	partial	yes
Tufano et al. [42]	(descriptive)	history mining of smells	n/a	yes
Peters & Zaidman [43]	smell lifespan	survival mining	not reported	yes
Lenarduzzi et al. [49]	(dataset)	SonarQube longitudinal mining	n/a	yes
Mohayeji et al. [58]	vulnerability fix time	KM + Cox	partial	yes
CLSA [59]	line survival	AST-aware survival mining	not reported	yes
The proposed study	**complexity accumulation**	**landmark Cox + automated guards**	**yes (guarded)**	**yes (forecasting)**

The present study addresses the three methodological limitations identified above by introducing a leakage-aware landmark survival framework. Nevertheless, several challenges remain for future work, including multi-language replication, validation on industrial software systems, and improving cross-project transfer through project-specific calibration or transfer learning.

The existing studies mostly focus to either explain the complexity of the current software or predict software defects with snapshot-based metrics. None explicitly avoid event tautology and covariate-circularity and none simultaneously model future complexity accumulation as time-to-event.

Accordingly, this study introduces a leakage-aware landmark survival framework that predicts future complexity accumulation while explicitly preventing event tautology and covariate-circularity.

3. Study Design

3.1 Research questions

RQ1. What are the signals that are observable early in a file's life that can predict when its cyclomatic complexity will increase significantly above its baseline?

Early development dynamics provide stronger predictive signals than initial structural properties for forecasting future complexity accumulation

RQ2. Is the improvement in predicting a trajectory from richer structural information, beyond process dynamics, significant?

The richer structural covariates and more complex survival learners do not significantly improve predictive performance over the proposed landmark trajectory model.

RQ3. Does the model transfer to projects that it was not trained on?

The overview for each of the mining, attrition to the analyzable population, landmark construction, automated guards, and evaluation tasks is summarized in figure 1. The proposed framework is generalized fairly well for unseen projects, but there is still a need for project specific calibration.

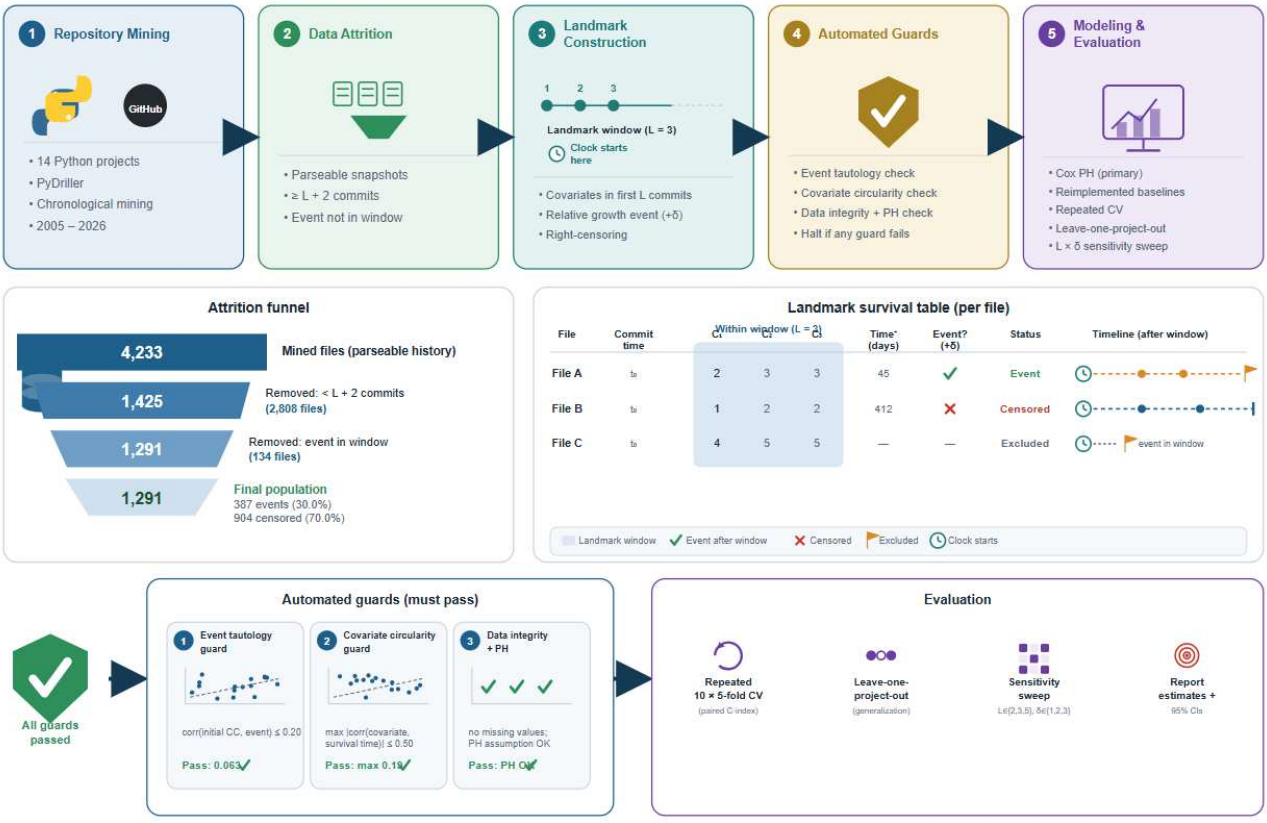


Figure 1. Overview of the landmark survival analysis workflow

Why a survival-analysis framework. The choice of a time-to-event framework is deliberate, and the alternatives were considered and set aside for concrete reasons. Framing the task as binary classification ("will this file ever become complex?") is inadequate because most files are still under observation when the data are collected: seventy per cent never experience the event within the mining window, and a classifier is forced either to mislabel these censored files as negatives, which understates risk, or to discard them, which wastes most of the sample. Plain regression on the time until the event fails for the same reason, since the outcome value is unknown precisely for the censored files, and dropping them reintroduces the bias. Standard time-series forecasting is a poor fit because the quantity of interest is not the future value of a series at a fixed horizon but the timing of a one-off transition. Sequence models such as LSTMs or Transformers were also set aside: beyond the same censoring problem, only 387 events are available across fourteen heterogeneous projects, far too few to fit a high-capacity model without overfitting, and such models forfeit the interpretability that a quality-assurance setting requires. Survival analysis was designed for exactly this situation, namely right-censored time-to-event data with covariates, and the Cox proportional-hazards model [10] yields interpretable, per-covariate hazard ratios while handling censoring by construction [11]. It is therefore the natural framework for forecasting when, not merely whether, a file will accumulate complexity.

3.2 Corpus and language choice

The following criteria were used to select 14 Python open-source projects: (i) they have a multi-year commit history that isn't terminated by a single year of stagnant commits; (ii) they use a single dominant implementation language so that one metric toolchain applies uniformly; (iii) they are not tutorials or generated code, but production software; and (iv) they are spread across a range of governance styles, from small, heavily reviewed libraries (requests, click, flask, httpx, rich, fastapi) to large, faster-moving systems (sqlalchemy, sphinx, airflow, streamlit, celery, scrapy, pipenv, sentry-python). The spread of the governance matters: in a pilot restricted to the seven mature libraries, 83% of files never grew in maximum cyclomatic complexity at all, which bounds the information available to any forecaster; the larger systems were added to raise the base rate of accumulation.

It is important to note that this restriction on Python is not a convenience, but it is a deliberate design decision. To maximize the practical relevance of the population studied, it is the first language of the 2024 Octoverse report [70]. Second, the Radon toolchain offers well established implementations of cyclomatic complexity and the Maintainability Index for Python, instead of using ad-hoc implementations. Third, the language inherently has a stable grammar that has remained the same over the entire Python 3 period, so structural covariates (nesting, interfaces, exception handling) are extracted deterministically as opposed to heuristically through parsing. Fourthly, the reliability of a parsing over twenty years of snapshots can be reported and measured; any snapshots that do not now parse are ignored rather than silently mis-measured. Cyclomatic complexity, furthermore, is not semantically equivalent in different languages: identical thresholds have different meanings for Python, C++ and Java [71, 72] and so a single-language design protects construct validity; multi-language replication (e.g., using the lizard analyzer) is future work. As summarized in Table 2, there are 36,076 file-level commit observations over 4,233 different files and authored by 1,376 developers, in the period from July 2005 to February 2026, with per-project event rates as shown in Figure 2.

Table 2. Characteristics of the analyzed open-source projects

Project	File-commit obs.	Landmark files	Events	Event rate
airflow	2,261	70	19	0.271
celery	2,349	104	24	0.231
click	1,996	59	24	0.407
fastapi	3,105	147	16	0.109
flask	1,229	52	17	0.327
httpx	3,088	107	36	0.336
pipenv	2,111	20	8	0.400
requests	1,317	29	13	0.448
rich	3,559	110	49	0.445
scrapy	3,172	199	30	0.151
sentry-python	5,031	154	65	0.422
sphinx	1,399	63	20	0.317
sqlalchemy	2,557	72	34	0.472
streamlit	2,902	105	32	0.305
Total	36,076	1,291	387	0.300

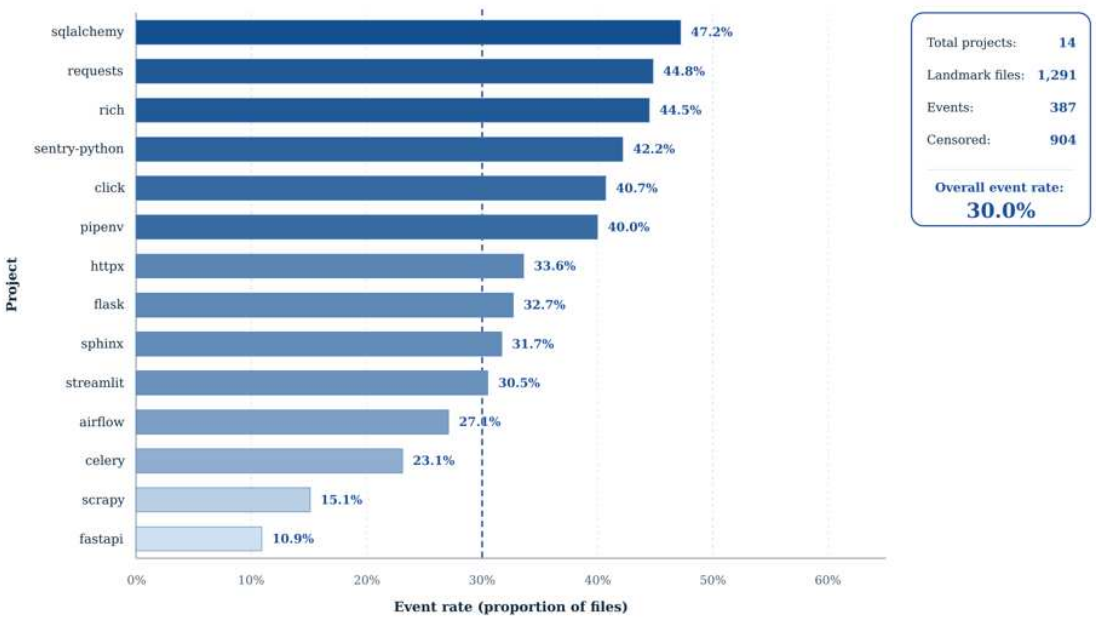


Figure 2. Distribution of Complexity Accumulation Events across Projects

The mining was limited to the earliest 2,500 commits per repository to fit a free Colab runtime but resulted in truncating late lifetimes, which inflates censoring, which was revisited in Section 6.

3.3 Mining pipeline

PyDriller [73] is used to traverse each repository in chronological order. Post change source is recorded for each commit and the following are computed for each modified Python file: lines of code; per-function cyclomatic complexity, aggregated as the file maximum (max_cc); the

Maintainability Index (via Radon); and nine structural attributes extracted from the abstract syntax tree maximum nesting depth, import count, comment ratio, docstring ratio, class count, maximum parameter count, exception-handler count, return count, and complexity density historic snapshots, which no longer parse, are skipped and counted. Coupling is calculated from a per-repository time-ordered import graph, which is updated as commits are received; fan-in (number of other files that import this file's module) and fan-out (number of other files that are exported by this file's module) are read when the commit happens. Coupling covariates can't leak future repository state, since the graph is only reflecting commits that have been processed.

Pipeline scale and reproducibility. The extraction stage is the data-engineering core of the study and is fully automated end to end. Chronological traversal of fourteen repositories produced 36,076 file-level observations across 4,233 distinct files and 1,376 contributors, spanning July 2005 to February 2026, with every observation carrying eighteen derived measures computed at the moment of the commit. Three properties make the pipeline reusable rather than single-use. First, extraction is streaming: repositories are traversed commit by commit and the import graph is updated incrementally, so memory use is independent of repository size and the same code scales to larger corpora without modification. Second, every derived measure is recomputed from the raw snapshot rather than read from a cache, so the dataset can be regenerated from the repository URLs alone. Third, snapshots that fail to parse under the current grammar are skipped and counted rather than silently dropped, so data loss is measurable rather than hidden. The complete pipeline, together with the resulting datasets and every table and figure derived from them, regenerates from a single notebook.

3.4 Event definition, δ choice, and the tautology hazard

The survival framing (Fig. 3) assumes that a file's covariates are fixed at the end of a landmark window, after which the file experiences the event (File A) or comes to the end of the observed history without experiencing the event or is right-censored (File B); if the event in the window already occurred, then covariates postdate it and it is excluded (File C). The key design decision is the event and the first attempt was a teaching moment. A model with quite good discrimination (single-split concordance 0.843) was found under an absolute threshold (max_cc ≥ 10), which is a convention in the thresholds literature [71, 72, 74]. The result came back tautological: only 3.9% of the files actually crossed, the indicator of the event was 0.61 correlated with initial complexity, and 0.825 correlation with only initial complexity was obtained by the univariate ranking.

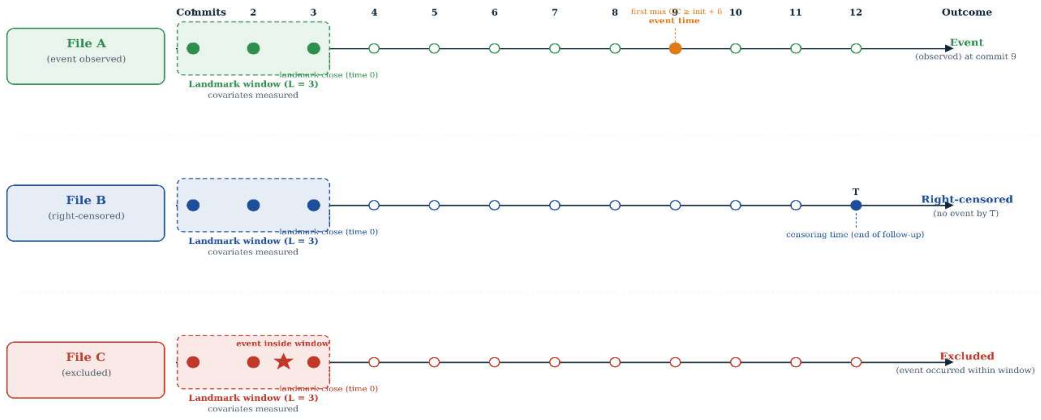


Figure 3. Landmark-Based Prediction Timeline

A relative event is therefore defined: a file will experience the event at the first commit where at least max_cc ≥ max_cc(first commit) + δ. The primary analysis sets δ = 2, which has a theoretical justification based on the definition of the metric: Adding a linearly independent decision point makes the cyclomatic complexity grow by exactly 1, so that at least two new decision points or a compound conditional is required to increase it by 1. δ = 2 is also a 50% increase in structure relative to the population median initial max_cc value of 4.0. Values of δ = 1, 3 are swept in sensitivity analyses. If the relative definition is used, then the initial-complexity/event correlation drops to 0.063 and an automatic guard is programmed to shut down the pipeline if it ever goes above 0.20. No trajectory can be observed in files with fewer than L + 2 commits, these are excluded; Table 3 lists all the exclusions. The final population consists of 1,291 files of which 387 events occurred (rate 0.300), and 904 trajectories were deleted as censored, with the median follow-up time of 74 days and the median censored time of 213 days (Figure 4 shows a concrete example; Figure 5 the distributions of times).

Table 3. Definitions of Landmark-Derived Covariates

Stage	Files remaining	Removed	Reason
Mined files with parseable history	4,233		
Fewer than L + 2 = 5 commits	1,425	2,808	no observable trajectory (55% of files have ≤ 2 commits)
Event inside the landmark window	1,291	134	covariates would postdate the event
Final population	1,291		387 events / 904 censored

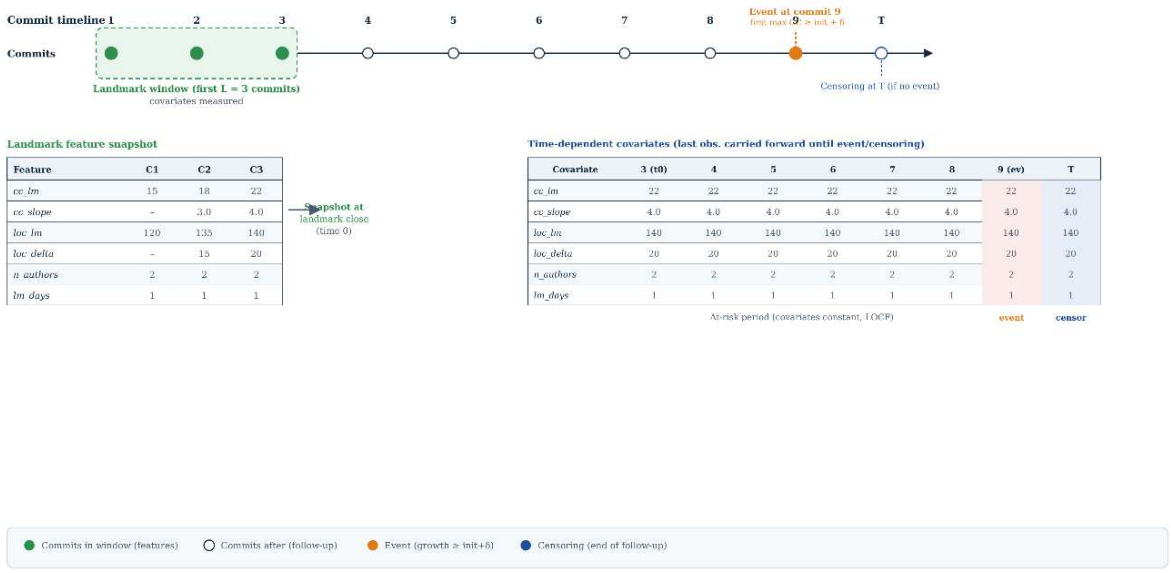


Figure 4. Landmark Feature Extraction and Time-Dependent Covariate Construction

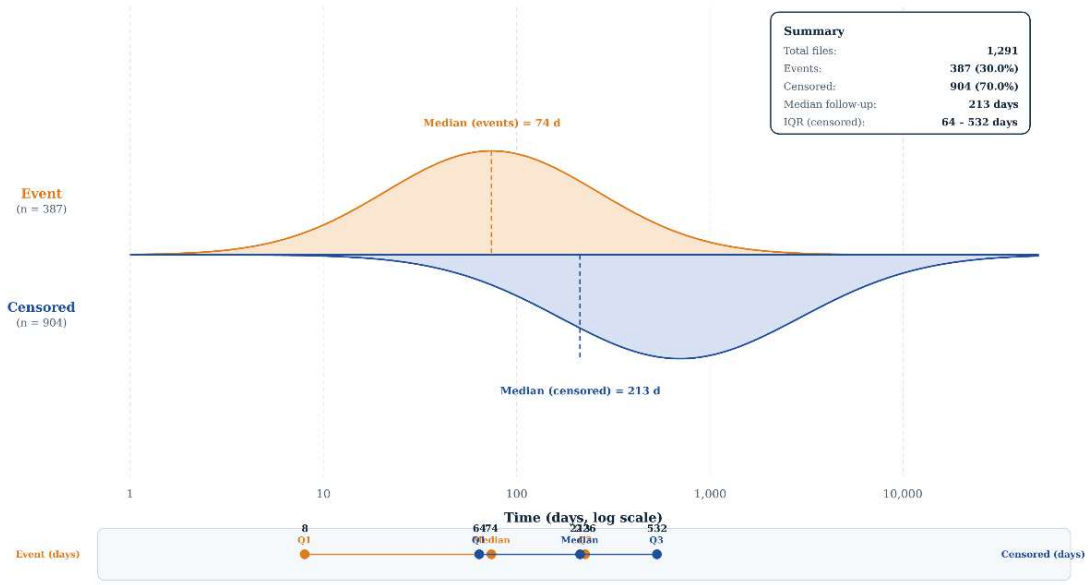


Figure 5. Distribution of Time-to-Complexity Accumulation Events

3.5 Landmark covariates, L choice, and the circularity hazard

The covariate design was motivated by an initial failed formulation that exposed a severe information-leakage issue. A rate of commits, which is seemingly natural covariate, achieved a spectacular univariate concordance of 0.910, commits/observation span. The reason for this is that the denominator is the result: for event files, the span ends at the event, meaning that corr (span, survival duration) = 1.00 exactly; and any per-time rate represents survival time in reverse, the repository-mining analogue of immortal-time bias [12]. The landmark design [60, 61] eliminates the hazard by design: All the covariates are computed in a fixed window of the first L commits, common to all files, and the survival clock begins at the time of the closure of that window. A second automated guard checks if any covariate has a correlation with the duration of survival larger than $|r| = 0.5$ and stops otherwise.

The choice $L = 3$ is practical and data-driven. At least two observations are required for any within-window slope, so $L \geq 2$; the population median of two commits per file means each increment of L sharply shrinks the analyzable population (1,588 files at $L = 2$, 1,291 at $L = 3$, 918 at $L = 5$); and $L = 3$ adds one extra observation for slope stability while retaining 81% of the $L = 2$ population. With a median landmark span of 15 days, predictions become available roughly two weeks into a file's life, preserving early-warning utility. Values $L \in \{2, 5\}$ are swept in sensitivity analyses. Fourteen covariates enter the primary model (Table 4), all measured at or within the landmark. Seven further covariates from the mining stage (import count, Halstead volume, complexity density, function count, nesting slope, class count, return count) were removed after collinearity analysis: fan-out and import count correlate at $r = 0.79$ and split one signal into opposite-signed coefficients, and the full 21-covariate set reaches a maximum variance-inflation factor of 8.6 versus 2.9 after pruning (Figure 6). For this reason, individual coefficient signs are not interpreted anywhere in the paper; all inference is predictive.

Table 4. Covariates of the primary model

Covariate	Family	Definition (at landmark)
cc_lm	static	maximum function-level cyclomatic complexity
cc_slope	trajectory	change in max CC across the window
loc_lm	size	lines of code
loc_delta	trajectory	change in LOC across the window
mi_lm	static	Maintainability Index
n_authors	process	distinct commit authors in the window
lm_days	process	calendar days spanned by the window (inverse tempo)
nesting_lm	structural	maximum AST nesting depth
comment_ratio_lm	documentation	comment lines / non-blank lines
docstring_ratio_lm	documentation	documented defs / (defs + classes + module)
max_params_lm	interface	maximum parameter count over functions
n_except_lm	structural	exception handlers
fan_in_lm	coupling	files importing this module (time-ordered graph)
fan_out_lm	coupling	modules imported by this file

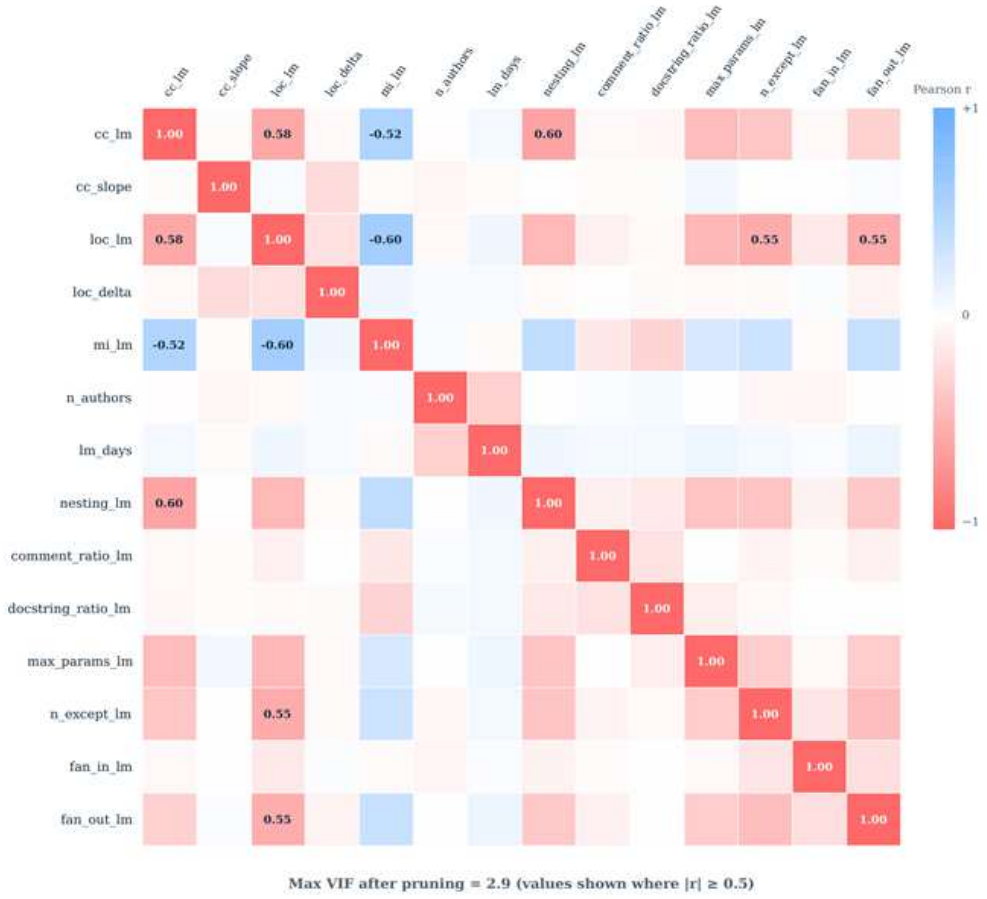


Figure 6. Dependency Structure among Model Covariates

3.6 Models and baselines

The primary model is a Cox proportional-hazards regression [10] with standardized covariates and L2 penalty ($\lambda = 0.5$), with a separate baseline hazard for each project [51] and using scaled Schoenfeld residuals [50] as a check for proportionality. In the model class of tree-based survival learners [53] a gradient-boosted Cox model [18] is fitted shallow regression trees (depth 3, subsampling 0.8, learning rate 0.05, 150 rounds) to the gradient of the Breslow partial likelihood to answer the model-class question RQ2. The same data is used to implement each of the five baselines, which correspond to the 5 different predictor families in Section 2: size (LOC at landmark) [13]; static complexity (max_cc at landmark) [1]; the Maintainability Index at landmark; coupling (fan-in at landmark); and process (author count) [14, 75]. Every baseline sorts the files according to their raw value, and each gets its optimal orientation on the test fold, which is intended to be conservative for the trajectory model's claim.

3.7 Evaluation protocol

Harrell's concordance index [16, 17] is used to measure discrimination: 0.5 is chance. All headline numbers are based on ten repeated splits (with 5-fold cross-validation per split) of a single test/train split, with per-fold differences in the trajectory model and the top-performing baseline, and with percentile 95% confidence intervals reported. Only when there is no zero in the paired interval is there a claim of superiority. In both contrasts (structural vs. process-only covariates and gradient-boosted vs. linear Cox) in RQ2, the same paired-fold procedure was used. Leave-one-project-out (LORO) validation is used for RQ3. The complete protocol is repeated for the sensitivity analysis for $L = 2, 3, 5$ and for the sensitivity analysis for $\delta = 1, 2, 3$. The re-generation of all experiments comes from a single notebook, all the components were unit-tested before deployment, according to methodology for repository mining [76] and empirical software engineering [77, 78].

4. Results

4.1 RQ1: What predicts complexity accumulation?

The repeated cross-validation concordance data for all models is presented in Table 5, and shown graphically in a forest plot in Figure 8. The Cox trajectory model attains 0.663 (95% CI 0.626–0.704); All baseline models achieved concordance values no higher than 0.563. The paired per-fold gain compared to the best baseline is +0.086 (95% CI +0.023 to +0.131). The proposed model outperformed the strongest baseline in every fold of the primary evaluation protocol.

Table 5. Predictive Performance of the Proposed and Baseline Models

Model	Concordance	95% CI low	95% CI high
Cox trajectory (14 covariates)	0.663	0.626	0.704
Process (author count)	0.563	0.514	0.604
Coupling (fan-in)	0.559	0.520	0.608
Static complexity (McCabe)	0.541	0.509	0.584
Size (LOC)	0.528	0.502	0.558
Maintainability Index	0.527	0.503	0.556

The sequencing of the baselines within the text is a discovery of its own. Initial static complexity (0.541) is just slightly above chance, while size and Maintainability Index are actually worse (0.528, 0.527). Process-oriented variables (such as author count, with a value of 0.563), and fan-in coupling (with a value of 0.559) are the strongest single signals. The absolute risk predicted is also well separated (Figure 7), with event rates steadily increasing from 0.19 (low tercile) to 0.32 and 0.39 (high tercile); the survival curves start to separate from the first week. The pattern extends the defect prediction [32, 14, 15] approach of regularity in process to a new outcome – prediction of complexity growth in the first few days of a file's development.

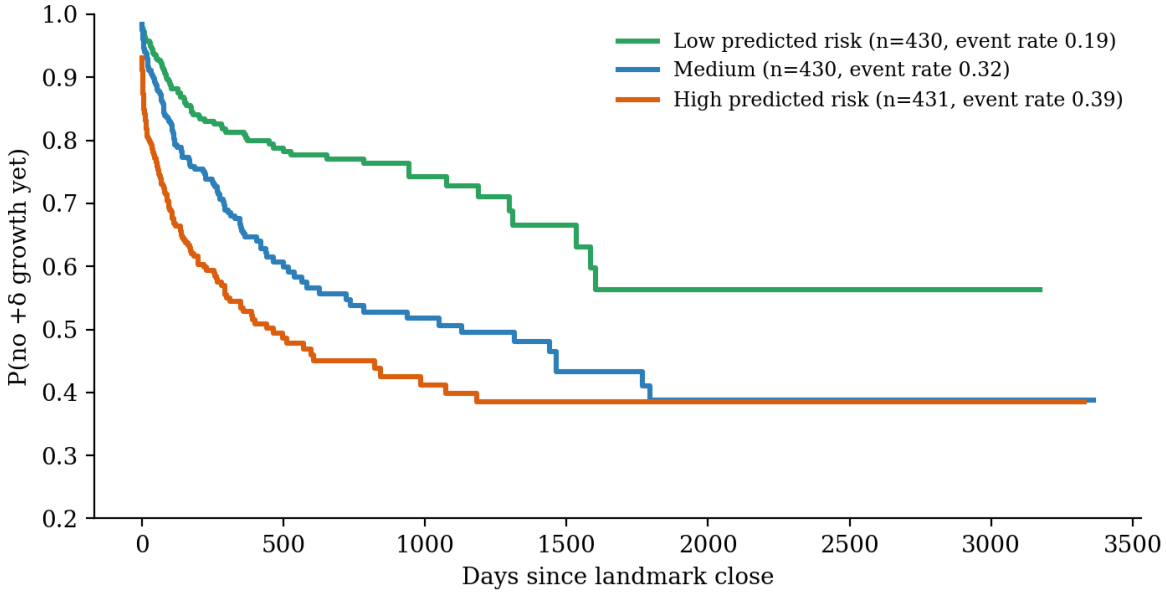


Figure 7. Risk-Stratified Survival Curves for Complexity Accumulation

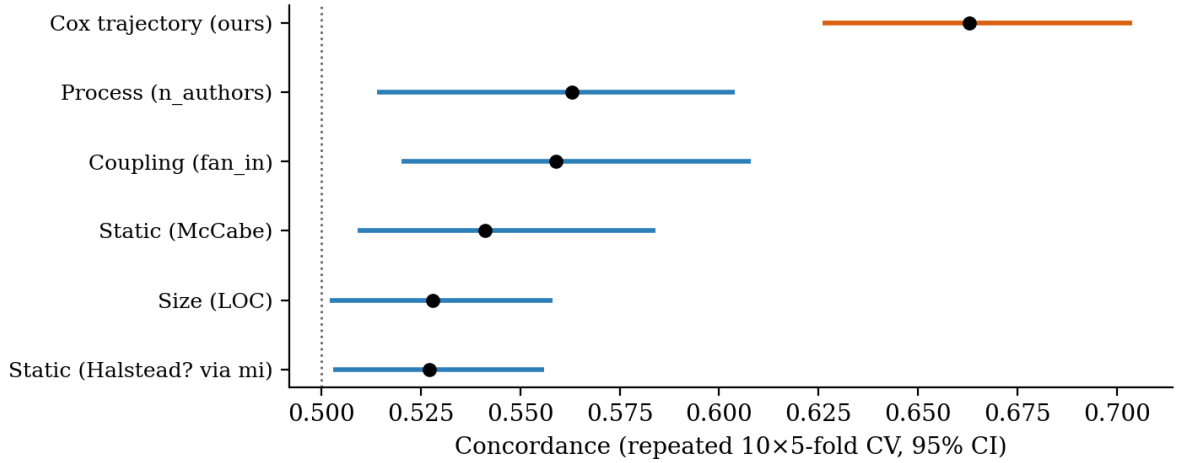


Figure 8. Multivariable Hazard Ratios for Complexity Accumulation

4.2 RQ2: Does richer structural information improve prediction?

The progression is monotone but modest when moving from one set of covariate parameters to another (Table 6, Figure 10): the set of nine process/trajectory covariates alone achieves a simple progression of 0.652 (95% CI 0.597–0.694); the set of fourteen covariate parameters after pruning 100 times progresses to 0.661 (95% CI 0.608–0.701); and the full set of 21 covariate parameters advances to 0.665 (95% CI 0.594–0.721). The difference in the paired data is +0.014, and the 95% CI for the difference

is [−0.026, +0.050] which contains 0. At covariate granularity, the same point is made via permutation importance (Figure 9): most structural covariates found in the nesting, parameters, exception handling, contribute near zero compared to landmark tempo (lm_days), documentation ratio, coupling fan-out, or author count. The difference between the model-class contrast is also null. The gradient-boosted Cox wins a majority of folds but by small margins: +0.021 (95% CI −0.017 to +0.078) on the expanded covariate set, and +0.028 (95% CI −0.022 to +0.055) in a pilot on the nine-covariate set. It is concluded that at this population size and event rate, the model class is not the limiting factor, as the linear Cox model takes out nearly all the signal the covariates convey. For both null contrasts, the objections of reviewers, which are the most natural, are addressed, namely, “add richer features” and “use a stronger learner,” and both are reported prominently (Table 7) because they are not rhetorical, but answered empirically.

Table 6. Ablation Analysis across Covariate Sets

Covariate set	k	Concordance	95% CI
Process/trajectory only	9	0.652	0.597–0.694
Pruned (primary model)	14	0.661	0.608–0.701
Full (pre-pruning)	21	0.665	0.594–0.721

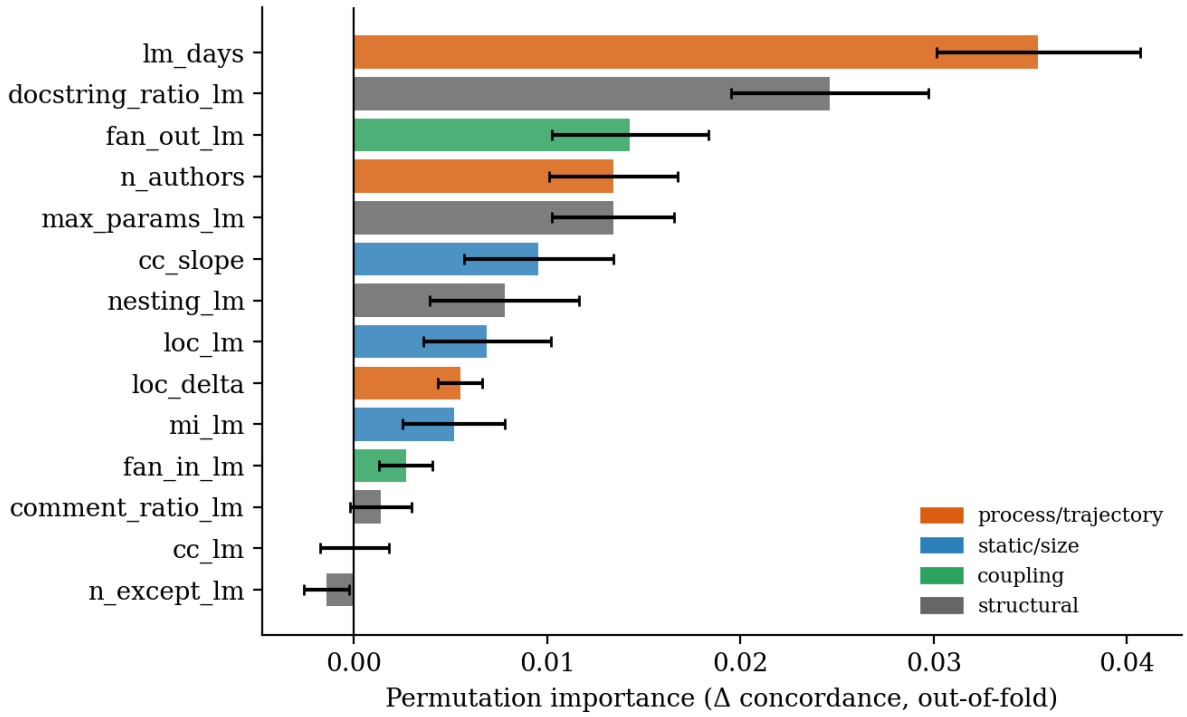


Figure 9. Relative Importance of Landmark-Derived Predictors

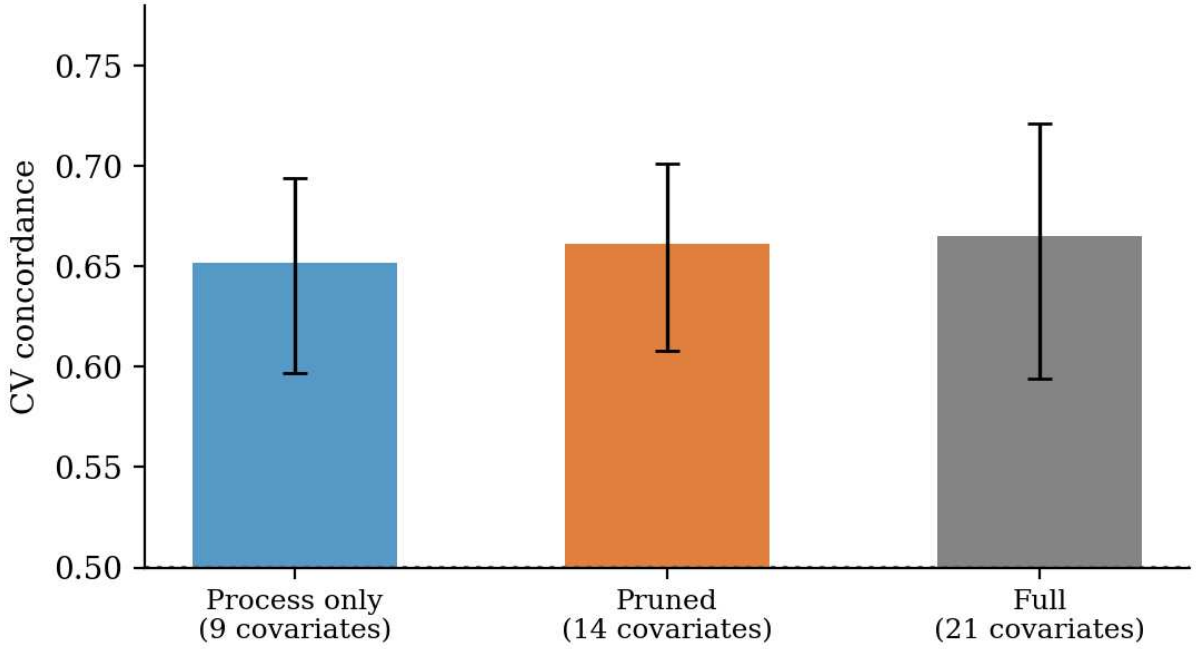


Figure 10. Incremental Contribution of Covariate Groups

Table 7. Summary of Paired Statistical Comparisons

Contrast	Estimate	95% CI	Verdict
Trajectory model − best baseline (primary)	+0.086	+0.023 to +0.131	supported
Structural (21) − process-only (9)	+0.014	−0.026 to +0.050	not supported
GB-Cox − linear Cox (expanded set)	+0.021	−0.017 to +0.078	not supported
GB-Cox − linear Cox (9-covariate pilot)	+0.028	−0.022 to +0.055	not supported

4.3 RQ3: Generalization to unseen projects

Leave-one-project-out results are reported in Table 8 and Figure 11. The trajectory model outperforms McCabe on ten of 14 projects, with a mean holdout concordance of 0.599 compared to 0.588 for the McCabe baseline, and a holdout concordance below 0.5 on projects such as scrapy (0.482), sphinx (0.486), pipenv (0.480), and sqlalchemy (0.497). The difference between pooled cross-validation (0.663) and LORO (0.599) is the honest measure of cross-project transfer; about half of the within-corpus advantage remains on an unseen project. Project heterogeneity is substantial; per-project event rates range from 0.109 to 0.472, and repository stratification inside the training folds cannot help a project whose baseline hazard was never observed. This trained model can be practically used as an in-house tool: calibration done on a small number of projects, application to siblings of projects under the same governance is supported, and shipping universal coefficients is not.

Table 8. Cross-Project Validation Results (Leave-One-Project-Out)

Held-out project	Files	Events	Trajectory model	McCabe
airflow	70	19	0.725	0.656
celery	104	24	0.558	0.530
click	59	24	0.548	0.533
fastapi	147	16	0.744	0.654
flask	52	17	0.577	0.552
httpx	107	36	0.685	0.549
pipenv	20	8	0.480	0.713
requests	29	13	0.675	0.613
rich	110	49	0.681	0.671
scrapy	199	30	0.482	0.538
sentry-python	154	65	0.575	0.526
sphinx	63	20	0.486	0.559
sqlalchemy	72	34	0.497	0.626
streamlit	105	32	0.674	0.515
Mean			0.599	0.588

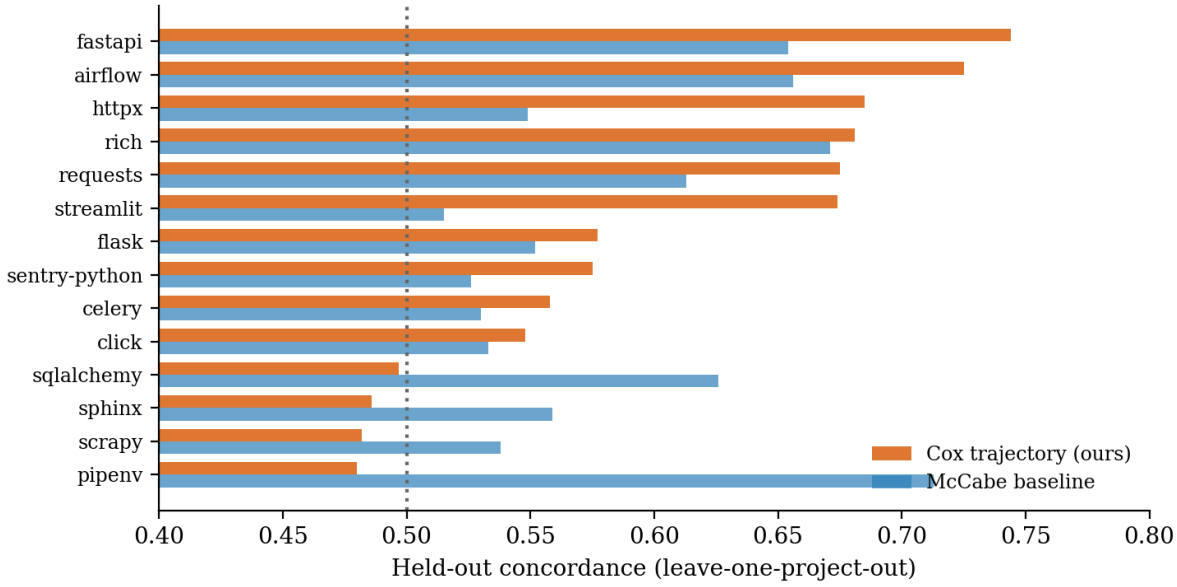


Figure 11. Cross-Project Generalization Performance

4.4 Sensitivity and robustness

Table 9 and Figure 12 display the entire cross valid comparison for nine combinations of landmark size and growth threshold. The paired gain over the best baseline is positive in all nine and in eight it is statistically supported, the only case where it is not, L = 5, δ = 1, having an interval whose lower bound is just touching 0.000. Event populations are still okay (252-606 events) and the tautology guard is well below bound in all configurations. When the primary configuration was used, proportional-hazards diagnostics indicated no violation of proportionality. Lastly, a planned AI-authorship covariate was not constructed – a conservative pattern of a trailer (Co-authored-by with assistant identifiers, assisted-by, generated-with) matched zero of 20,014 commits in the initial

seven-project corpus. Even though this may sound like an observation, it is actually a bad sign that commit trailers aren't a viable attribution signal in mainstream Python open-source yet, even though there is a lot of promise in them for provenance.

Table 9. Sensitivity Analysis across Landmark Configurations

L	δ	Files	Events	Gain	CI low	CI high	Significant
2	1	1,492	606	+0.072	+0.025	+0.117	yes
2	2	1,588	467	+0.075	+0.043	+0.115	yes
2	3	1,623	364	+0.057	+0.025	+0.087	yes
3	1	1,152	459	+0.084	+0.045	+0.117	yes
3	2	1,291	387	+0.082	+0.043	+0.126	yes
3	3	1,346	316	+0.090	+0.047	+0.122	yes
5	1	764	302	+0.076	−0.000	+0.128	no
5	2	918	283	+0.084	+0.031	+0.138	yes
5	3	995	252	+0.084	+0.023	+0.143	yes

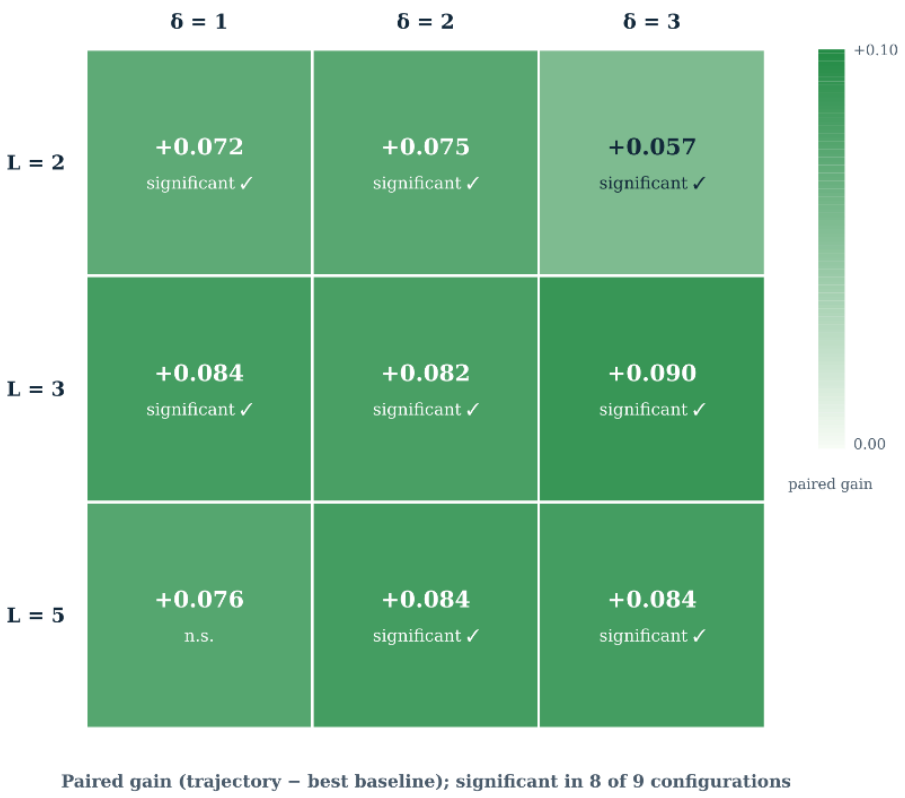


Figure 12. Model Robustness across Landmark Configurations

5. Discussion

5.1 Why initial structure fails, and early process succeeds

There are two mechanisms that are consistent with the near chance performance of snapshot complexity as a growth predictor. First, growth is demand-driven: files grow in complexity because features that they add invite change, and the demand is evident in processes signals such as tempo, growth, and inflow of contributors before it is evident in the structures. Second, mature open-source governance actively separates complexity of the current state from future expansion: a review gate or linter limits complexity a file can have, a file that is complex today is just as likely to be refactored as it is to expand.

What early activity encodes. These mechanisms can be read more concretely through the information carried by early activity. The tempo and volume of a file's first commits act as a proxy for the development style applied to it: files edited rapidly and by several hands early on are typically those under active feature pressure, and that pressure is what later drives branching and nesting. Early activity also reflects the amount of change demand a file attracts, since a file touched repeatedly in its first days is one the surrounding system keeps pulling on, and sustained demand accumulates complexity in a way that a one-off complex file does not. A file whose early history is quiet and whose structure settles quickly instead signals design stability, and such files tend to remain simple regardless of how large they were at birth. Initial static complexity captures none of this: it records what a file is at a single instant, not the developmental forces acting on it, which is why it carries so little forward-looking signal.

5.2 Agreement and disagreement with prior work

The results are consistent with those found by Moser et al. [14] and Rahman and Devanbu [15] who extended defect prediction to complexity accumulation in this case, using a metric that is aware of the censoring used. They also support the greater sensitivity of change history over size and structure, as concluded by Graves et al. [32] and survival framing of Koru et al. [57, 13] who predicted that risk is more likely to be expressed in relative than absolute terms and found this to be the case here. The outcomes confirm Shepperd's critique [5] that size and cyclomatic complexity did not distinguish and both were around chance (0.528 vs 0.541). They challenge the practice of complexity-gated dashboards that rely on absolute thresholds [71, 72] of “hotspots” based on the current complexity: here, initial structure conveys little forward-looking information. Finally, the results do not contradict, but rather nuance, the results of Tufano et al. [42]: smells can be created with the artifact, but ‘complexity growth’ after creation is a function of post-creation dynamics—initial state is not destiny, and these two observations are complementary descriptions of different points in a file's life.

5.3 Practical Recommendations

Immediately after a file is at the landmark window, the proposed framework can be used in continuous integration and software quality monitoring procedures. Instead of measuring the cyclomatic complexity of files only in terms of the complexity at build time, development teams can also measure the cyclomatic complexity of the file in terms of expected future complexity, allowing for more timely decisions to refactor code, more targeted code reviews, and better utilization of maintenance resources. The framework is applicable for proactive quality management before complexity becomes difficult and costly to manage as only information available during the initial development period is required for the framework.

A concrete deployment scenario. Suppose that, thirty days into a file's life, the fitted model assigns it a hazard in the top predicted tercile. The signal is not an instruction to rewrite the file; it is a prompt to act while intervention is still cheap. Three graded responses are appropriate. If the file is small and central, a lightweight design review can be scheduled to confirm that its responsibilities are not drifting before any structure hardens. If the elevated risk is accompanied by rapid early growth, targeted refactoring, such as extracting helpers or flattening nesting, can be prioritised for the next maintenance window rather than deferred until the file is already a hotspot. If the file sits on a critical path but cannot yet be restructured, additional tests can be added to raise the safety margin for the changes the model predicts are coming. In each case the value of the forecast lies in timing: the same actions taken reactively, after complexity has accumulated, cost substantially more, so a signal available two weeks into a file's life converts expensive corrective work into cheaper preventive work.

5.4 Implications and methodological lessons

The proposed framework provides practitioners with an actionable early-warning signal, a triage signal, after a few commits of a file's early tempo, early size growth, and early influx of contributors identify the files worth watching, while a dashboard sorted by current McCabe values will systematically fail to identify tomorrow's hotspots. As the cross project transfer is weak, it is recommended to calibrate per organization, and the pipeline is fully automatic, so it is not expensive. The two nulls give researchers a boundary for where improvement is not likely to be found: at this scale, neither changes to structural features nor model nonlinearity is important. Two failure modes identified during design and now automatically protected against are suspected to exist in other studies of repository mining (Table 10): absolute-threshold events on skewed metrics are created because of the starting value, and whole-history covariates, particularly per-time rates, include the outcome in the predictors. Based on the analogies with previous recommendations [76, 78] it is recommended that event-tautology and covariate-circularity bound reporting is considered the standard practice in trajectory-level mining studies.

Table 10. Methodological Hazards and Their Mitigation

Hazard	Diagnostic	Before	After	Remedy
Event tautology	corr(initial CC, event)	0.61	0.063	relative (+δ) event; guard < 0.20
Covariate-circularity	corr(span, duration)	1.00	n/a	landmark window; guard |r| < 0.5
Coefficient instability	max VIF	8.6	2.9	prune 21 → 14 covariates; no sign interpretation
Split instability	single-split concordance	0.701 vs 0.624		repeated 10×5 CV, paired CIs

6. Threats to Validity

Threats are summarized by category in Table 11, and the mitigation applied and the residual risk acknowledged; the most material items are then discussed.

Table 11. Summary of Threats to Validity and Mitigation Strategies

Category	Threat	Mitigation	Residual risk
Construct	event defined on max function-level CC only	δ swept {1,2,3}; MI and cognitive metrics discussed	other operationalizations may shift rankings

Construct	fan-in from mined commits, top-level module names	documented as a lower bound	under-counting; name collisions
Construct	renames restart trajectories	stated; PyDriller default identity	fragmented lifetimes
Internal	confounding by file purpose (core vs. examples)	repo stratification; predictive-only claims	purpose not modeled
Internal	residual collinearity	pruning to VIF 2.9; coefficient signs never interpreted	importance ranks approximate
Internal	commit cap truncates late lifetimes	censoring-aware estimator; cap reported	reduced power at long horizons
External	Python-only, OSS-only, complexity-inert population	governance-diverse corpus; language choice justified (§3.2)	industry/other languages may differ
External	weak cross-project transfer	LORO reported as headline result	per-organization calibration required
Conclusion	concordance sensitive to censoring distribution	repeated CV + LORO convergence; replication package	alternative estimators [52] advisable
Conclusion	single-split instability	repeated 10×5 CV with paired percentile CIs	

Three points deserve emphasis. First, the population is complexity-inert by construction 83% of files in the pilot never grew, so the achievable concordance is bounded by base-rate scarcity, and results may differ in less-reviewed codebases. Second, historic snapshots that no longer parse are skipped and counted, marginally biasing early-2000s observations. Third, the AI-attribution null is a statement about commit trailers in this corpus and period, not about the prevalence of AI assistance, which trailer conventions demonstrably under-record.

Dependence on the measurement toolchain. All complexity values in this study are produced by the Radon implementation of cyclomatic complexity and the Maintainability Index, and cyclomatic complexity as operationalised here follows the counting conventions used by widely deployed tools such as SonarSource. Different tools do not always agree: boolean operators, comprehensions, and exception branches may be counted differently, so the absolute complexity of a given file can shift with the analyser, and with it the exact point at which the relative growth event fires. Two considerations limit the resulting risk. The event is defined relative to each file's own initial value rather than to an absolute cross-tool threshold, so consistent per-tool offsets cancel; and the sensitivity analysis over the growth threshold shows the model's advantage to be stable across a range of event definitions. Replication with an independent complexity analyser, or with a semantically richer measure such as cognitive complexity, would nonetheless strengthen confidence that the findings reflect complexity growth itself rather than an artefact of one measurement tool, and is left to future work.

7. Conclusion and Future Work

Software complexity was reframed from a snapshot property to a forecastable trajectory, and 1,291 file lifetimes across fourteen Python projects were analyzed with a leakage-free landmark survival design. Three findings emerge. First, complexity accumulation is predictable within a corpus concordance of 0.663, a +0.086 paired gain over the best prior-work baseline, robust in eight of nine design

configurations and the predictive signal originates primarily in early development dynamics rather than initial code structure, whose predictive power is barely above chance. Second, neither richer structural covariates nor gradient-boosted survival models produced statistically significant improvements: structural covariates and gradient-boosted survival modeling both have confidence intervals for the contrasts that include zero. Third, cross-project transfer is honest but modest: leave-one-project-out validation loses about half of the within-corpus advantage. These findings also support a process-oriented perspective on complexity risk management, one that focuses on individual organizations, rather than a single project; and the use of tautology and circularity guards as standard practices in repository mining at the trajectory level, when dealing with complexity risks.

In addition to the empirical results presented, this work shows the feasibility and usefulness of leakage-aware longitudinal modeling in software evolution research. In general, the proposed framework shows that, when the quality of the software changes over time, trajectory-aware prediction can provide more reliable evidence than the snapshot-based analysis. Hopefully, this methodology will stimulate future empirical software engineering research to consider modeling software quality over time, taking into account leakage.

Future work builds on stated limitations: multi-language replication using language-agnostic analyzers, industrial & less-governed codebases, where base rates of accumulation are expected to be higher, random effects for projects or transfer learning to close the LORO gap, and revisiting the AI-authorship covariate, whose trailer-based signal proved absent here, as attribution conventions mature (as conventions for attributing code develop). All the pipeline mining, the guards, models, and all the tables and figures contained in this paper regenerate from a single notebook, and all the artifacts are released for replication.

Funding: No funding was received for this research.

Conflicts of Interest Declaration: The author declares there are no conflicts of interest in the submission.

Author Contributions: Moatasem M. Draz is responsible for the study conception and design, the implementation of the mining and analysis, the statistical analysis and validation, the interpretation of results, and the writing and revision of the manuscript.

Ethic Issues Declaration: The author declares there are no ethical issues in the submission. The study analyses publicly available source-code repositories and does not involve human participants or personal data; author identifiers extracted from commit metadata were used only in aggregate to count distinct contributors.

Data Availability: The datasets generated and analyzed during the current study, together with the complete dataset-construction, feature-extraction code, and final evaluation results necessary to reproduce the findings, are open-source and publicly available in the GitHub repository: https://github.com/moatasem2005/complexity-trajectories-repo. The repository contains the end-to-end mining and analysis notebook, all result tables reported in the paper, along with a data dictionary describing every field. The fourteen analyzed projects are public open-source repositories, and their complete commit histories are retrievable directly from GitHub using the provided mining scripts.

References

[1] T. J. McCabe, “A complexity measure,” IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308–320, 1976.

[2] M. H. Halstead, Elements of Software Science. New York: Elsevier North-Holland, 1977.

[3] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476–493, 1994.

[4] E. J. Weyuker, “Evaluating software complexity measures,” IEEE Transactions on Software Engineering, vol. 14, no. 9, pp. 1357–1365, 1988.

[5] M. Shepperd, “A critique of cyclomatic complexity as a software metric,” Software Engineering Journal, vol. 3, no. 2, pp. 30–36, 1988.

[6] N. E. Fenton and M. Neil, “A critique of software defect prediction models,” IEEE Transactions on Software Engineering, vol. 25, no. 5, pp. 675–689, 1999.

[7] M. M. Lehman, “Programs, life cycles, and laws of software evolution,” Proceedings of the IEEE, vol. 68, no. 9, pp. 1060–1076, 1980.

[8] D. L. Parnas, “Software aging,” in Proc. ICSE, 1994, pp. 279–287.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, “Does code decay? Assessing the evidence from change management data,” IEEE Transactions on Software Engineering, vol. 27, no. 1, pp. 1–12, 2001.

[10] D. R. Cox, “Regression models and life-tables,” Journal of the Royal Statistical Society: Series B, vol. 34, no. 2, pp. 187–220, 1972.

[11] E. L. Kaplan and P. Meier, “Nonparametric estimation from incomplete observations,” Journal of the American Statistical Association, vol. 53, no. 282, pp. 457–481, 1958.

[12] S. Suissa, “Immortal time bias in pharmacoepidemiology,” American Journal of Epidemiology, vol. 167, no. 4, pp. 492–499, 2008.

[13] A. G. Koru, D. Zhang, K. El Emam, and H. Liu, “An investigation into the functional form of the size-defect relationship for software modules,” IEEE Transactions on Software Engineering, vol. 35, no. 2, pp. 293–304, 2009.

[14] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in Proc. ICSE, 2008, pp. 181–190.

[15] F. Rahman and P. Devanbu, “How, and why, process metrics are better,” in Proc. ICSE, 2013, pp. 432–441.

[16] F. E. Harrell, R. M. Califf, D. B. Pryor, K. L. Lee, and R. A. Rosati, “Evaluating the yield of medical tests,” JAMA, vol. 247, no. 18, pp. 2543–2546, 1982.

[17] F. E. Harrell, K. L. Lee, and D. B. Mark, “Multivariable prognostic models: issues in developing models, evaluating assumptions and adequacy, and measuring and reducing errors,” Statistics in Medicine, vol. 15, no. 4, pp. 361–387, 1996.

[18] J. H. Friedman, “Greedy function approximation: a gradient boosting machine,” Annals of Statistics, vol. 29, no. 5, pp. 1189–1232, 2001.

[19] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, “Measuring GitHub Copilot’s impact on productivity,” Communications of the ACM, vol. 67, no. 3, pp. 54–63, 2024.

[20] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, “Security weaknesses of Copilot-generated code in GitHub projects: an empirical study,” ACM Transactions on Software Engineering and Methodology, 2025, doi:10.1145/3716848.

[21] V. R. Basili, L. C. Briand, and W. L. Melo, “A validation of object-oriented design metrics as quality indicators,” IEEE Transactions on Software Engineering, vol. 22, no. 10, pp. 751–761, 1996.

[22] T. Gyimóthy, R. Ferenc, and I. Siket, “Empirical validation of object-oriented metrics on open-source software for fault prediction,” IEEE Transactions on Software Engineering, vol. 31, no. 10, pp. 897–910, 2005.

[23] R. P. L. Buse and W. R. Weimer, “Learning a metric for code readability,” IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 546–558, 2010.

[24] D. Posnett, A. Hindle, and P. Devanbu, “A simpler model of software readability,” in Proc. MSR, 2011, pp. 73–82.

[25] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," IEEE Transactions on Software Engineering, vol. 47, no. 3, pp. 595–613, 2021.

[26] D. Oliveira, R. Bruno, F. Madeiral, and F. Castor, "Evaluating code readability and legibility: an examination of human-centric studies," in Proc. ICSME, 2020, pp. 348–359.

[27] G. A. Campbell, "Cognitive complexity: an overview and evaluation," in Proc. Int. Conf. on Technical Debt (TechDebt), 2018, pp. 57–58.

[28] M. Muñoz Barón, M. Wyrich, and S. Wagner, "An empirical validation of cognitive complexity as a measure of source code understandability," in Proc. ESEM, 2020.

[29] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund, "Program comprehension and code complexity metrics: an fMRI study," in Proc. ICSE, 2021, pp. 524–536.

[30] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE Transactions on Software Engineering, vol. 33, no. 1, pp. 2–13, 2007.

[31] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in Proc. PROMISE, 2007.

[32] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," IEEE Transactions on Software Engineering, vol. 26, no. 7, pp. 653–661, 2000.

[33] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in Proc. ICSE, 2005, pp. 284–292.

[34] A. E. Hassan, "Predicting faults using the complexity of code changes," in Proc. ICSE, 2009, pp. 78–88.

[35] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in Proc. ICSE, 2007, pp. 489–498.

[36] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," IEEE Transactions on Software Engineering, vol. 39, no. 6, pp. 757–773, 2013.

[37] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," Journal of Systems and Software, vol. 150, pp. 22–36, 2019.

[38] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction," IEEE Transactions on Software Engineering, vol. 44, no. 5, pp. 412–428, 2018.

[39] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," Empirical Software Engineering, vol. 17, pp. 531–577, 2012.

[40] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1276–1304, 2012.

[41] M. W. Godfrey and Q. Tu, "Evolution in open-source software: a case study," in Proc. ICSM, 2000, pp. 131–142.

[42] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," IEEE Transactions on Software Engineering, vol. 43, no. 11, pp. 1063–1088, 2017.

[43] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in Proc. CSMR, 2012, pp. 411–416.

[44] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells," Empirical Software Engineering, vol. 23, pp. 1188–1221, 2018.

[45] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," IEEE Transactions on Software Engineering, vol. 39, no. 8, pp. 1144–1156, 2013.

[46] W. Cunningham, "The WyCash portfolio management system," in Addendum to Proc. OOPSLA, 1992, pp. 29–30.

[47] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: from metaphor to theory and practice," IEEE Software, vol. 29, no. 6, pp. 18–21, 2012.

[48] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," Journal of Systems and Software, vol. 101, pp. 193–220, 2015.

[49] V. Lenarduzzi, N. Saarimäki, and D. Taibi, “The Technical Debt Dataset,” in Proc. PROMISE, 2019, pp. 2–11.

[50] P. M. Grambsch and T. M. Therneau, “Proportional hazards tests and diagnostics based on weighted residuals,” Biometrika, vol. 81, no. 3, pp. 515–526, 1994.

[51] T. M. Therneau and P. M. Grambsch, Modeling Survival Data: Extending the Cox Model. New York: Springer, 2000.

[52] H. Uno, T. Cai, M. J. Pencina, R. B. D’Agostino, and L. J. Wei, “On the C-statistics for evaluating overall adequacy of risk prediction procedures with censored survival data,” Statistics in Medicine, vol. 30, no. 10, pp. 1105–1117, 2011.

[53] H. Ishwaran, U. B. Kogalur, E. H. Blackstone, and M. S. Lauer, “Random survival forests,” Annals of Applied Statistics, vol. 2, no. 3, pp. 841–860, 2008.

[54] J. L. Katzman, U. Shaham, A. Cloninger, J. Bates, T. Jiang, and Y. Kluger, “DeepSurv: personalized treatment recommender system using a Cox proportional hazards deep neural network,” BMC Medical Research Methodology, vol. 18, art. 24, 2018.

[55] C. Davidson-Pilon, “lifelines: survival analysis in Python,” Journal of Open-source Software, vol. 4, no. 40, p. 1317, 2019.

[56] S. Pölsterl, “scikit-survival: a library for time-to-event analysis built on top of scikit-learn,” Journal of Machine Learning Research, vol. 21, no. 212, pp. 1–6, 2020.

[57] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew, “Theory of relative defect proneness,” Empirical Software Engineering, vol. 13, pp. 473–498, 2008.

[58] H. Mohayeji, A. Agaronian, E. Constantinou, N. Zannone, and A. Serebrenik, “Securing dependencies: a comprehensive study of Dependabot’s impact on vulnerability mitigation,” Empirical Software Engineering, vol. 30, no. 3, 2025.

[59] P. Gurov, “Code Lifespan Survival Analysis (CLSA): predicting the survival of source code lines using AST-aware mining,” arXiv preprint arXiv:2606.04993, 2026.

[60] J. R. Anderson, K. C. Cain, and R. D. Gelber, “Analysis of survival by tumor response,” Journal of Clinical Oncology, vol. 1, no. 11, pp. 710–719, 1983.

[61] H. C. van Houwelingen, “Dynamic prediction by landmarking in event history analysis,” Scandinavian Journal of Statistics, vol. 34, no. 1, pp. 70–85, 2007.

[62] U. Dafni, “Landmark analysis at the 25-year landmark point,” Circulation: Cardiovascular Quality and Outcomes, vol. 4, no. 3, pp. 363–371, 2011.

[63] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, “The impact of AI on developer productivity: evidence from GitHub Copilot,” arXiv preprint arXiv:2302.06590, 2023.

[64] W. Harding and M. Kloster, “Coding on Copilot: 2023 data suggests downward pressure on code quality,” GitClear Technical Report, 2024.

[65] H. Jamil, M. Abid, and S. Shamail, “A comparative quality analysis of GPT-3.5-Turbo and GPT-4 generated code against human baselines,” in Proc. MSR, 2025.

[66] D. Molison et al., “Is LLM-generated code more maintainable and reliable than human-written code?,” preprint, 2025.

[67] E. Guimaraes, N. Nascimento, C. Shivalingaiah, and A. Nelapati, “Analyzing prominent LLMs: an empirical study of performance and complexity in solving LeetCode problems,” arXiv preprint arXiv:2508.03931, 2025.

[68] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” ACM Transactions on Software Engineering and Methodology, 2025, doi:10.1145/3747588.

[69] X. Sun, D. Ståhl, K. Sandahl, and C. Kessler, “Quality assurance of LLM-generated code: addressing non-functional quality characteristics,” Journal of Systems and Software, art. 112885, 2026, doi:10.1016/j.jss.2026.112885.

[70] GitHub, “Octoverse 2024: Python overtakes JavaScript as the most used language on GitHub,” GitHub Octoverse Report, 2024.

[71] T. L. Alves, C. Ypma, and J. Visser, “Deriving metric thresholds from benchmark data,” in Proc. ICSM, 2010.

[72] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida, “Identifying thresholds for object-oriented software metrics,” Journal of Systems and Software, vol. 85, no. 2, pp. 244–257, 2012.

[73] D. Spadini, M. Aniche, and A. Bacchelli, “PyDriller: Python framework for mining software repositories,” in Proc. ESEC/FSE (Demonstrations), 2018, pp. 908–911.

[74] P. Oliveira, M. T. Valente, and F. P. Lima, “Extracting relative thresholds for source code metrics,” in Proc. CSMR-WCRE, 2014, pp. 254–263.

[75] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, “Don’t touch my code! Examining the effects of ownership on software quality,” in Proc. ESEC/FSE, 2011, pp. 4–14.

[76] M. Vidoni, “A systematic process for mining software repositories: results from a systematic literature review,” Information and Software Technology, vol. 144, 106791, 2022.

[77] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” Empirical Software Engineering, vol. 14, pp. 131–164, 2009.

[78] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in Software Engineering. Berlin: Springer, 2012.