AND DATA SCIENCES
BULLETIN OF COMPUTER

Fork Visibility and Language Survivability in Large-Scale Open Source Ecosystems

Yajuan Wang

School of Management, Suzhou University, Suzhou, Anhui, 234000, China

Abstract

Forks on social coding platforms such as GitHub are both a technical mechanism for branching development and a social signal of community interest and reuse. Prior work has proposed "fork visibility performance" as a lens on the interoperability and survivability of programming language stacks in open source projects, but existing studies are limited by small samples and simplistic predictive models. In particular, a recent study using k-nearest neighbours on 38 projects suggested that multi-language interoperability may improve fork visibility, yet could not provide statistically robust evidence or a nuanced view of the relative importance of technical and social factors. This paper revisits fork visibility and language survivability at scale. Using data from a large corpus of GitHub repositories obtained via GHTorrent and the GitHub API, we formulate fork visibility prediction as a supervised learning problem and compare classical baselines to modern tree-based and neural architectures. We construct a rich set of features capturing language composition and interoperability, project scale and activity, social engagement, and language-network centrality. We further complement static prediction with a survival analysis of project activity, modelling the time to repository inactivity as a function of language stacks and social-technical covariates. Our contributions are threefold: (1) an operationalisation of fork visibility and language survivability suitable for large-scale analysis; (2) an empirical comparison of predictive models and feature families for fork visibility prediction; and (3) an investigation of how language interoperability interacts with social and organisational factors in shaping project survival. The paper concludes with implications for project maintainers and platform designers, and outlines how these models can underpin practical recommendations for language stack design in new projects.

Keywords: fork visibility prediction, programming language interoperability, GitHub repositories, social and technical factors, survival analysis of project activity

1. Introduction

Open source software (OSS) underpins a large part of modern computing infrastructure and is increasingly developed on social coding platforms such as GitHub. These platforms couple distributed version control with lightweight social features such as stars, forks, pull requests, and issues that make project activity and community interest highly visible in everyday practice. When a developer evaluates whether to adopt a library for a production service, to rely on a framework for a commercial product, or simply to learn a new technology, they will almost invariably glance at the

Corresponding author: Yajuan Wang (yajuanwang12@proton.me)

BCDS 2025, Vol. 6(1), 81-106. DOI: 10.71448/bcds2561-5

repository's stars and forks, scan recent pull requests, and check how responsive maintainers are in the issue tracker. In this environment, forks play a dual role. Technically, they support independent experimentation, long-running feature branches, and organisational customisation without disturbing the upstream repository. Socially, they act as a coarse-grained indicator of reuse, popularity, and potential influence: high-profile projects such as Linux, Kubernetes, or React accumulate tens of thousands of forks, while niche or abandoned repositories may receive none. Prior empirical work has explored stars and forks as signals of project popularity and growth patterns, as well as their relationship to code quality, contributor behaviour, and project evolution [1]. These studies confirm what many practitioners already suspect from experience: popularity metrics correlate with, but do not fully determine, the health and longevity of a project.

At the same time, mainstream OSS projects increasingly rely on *multi-language* stacks rather than a single dominant language. A modern web application repository might combine JavaScript or TypeScript for the frontend, a mixture of Python, Ruby, Go, or Java for the backend, shell scripts and Dockerfiles for deployment, and YAML or JSON for configuration. A data science toolkit may integrate Python notebooks, C++ or Rust extensions for performance-critical components, and R bindings aimed at a different user community. Even relatively small tools often contain build scripts, test harnesses, and configuration files written in domain-specific languages alongside the main implementation language. These concrete patterns, observed across thousands of GitHub projects, illustrate that multi-language systems are no longer exotic edge cases but the default for many domains. Prior large-scale studies have shown that programming languages differ in their defect profiles and usage patterns, and that most non-trivial projects involve multiple languages and technology stacks rather than a single homogeneous codebase [2]. However, despite the visibility of these trends to practitioners, the relationship between language interoperability, project survival, and visible success signals such as forks remains poorly understood in a rigorous, quantitative sense.

A recent small-scale study by Chua (2018) took a first step in this direction by introducing the concept of *fork visibility performance* as a measure of how well a project's language stack and interoperability characteristics predict its observed number of forks. In practical terms, that study asked whether projects that combine certain languages, or that rely on highly interoperable stacks, tend to achieve higher fork counts than comparable projects with less interoperable stacks. The analysis used a k-nearest neighbours (KNN) classifier on 38 projects, focusing primarily on JavaScript-based stacks, and reported that specific language combinations and an interoperability index correlated with higher fork visibility. While this result is conceptually appealing and resonates with anecdotal experience from practitioners who deliberately choose "popular" languages to lower adoption barriers, the study's limited sample size, coarse features, and reliance on a single simple model restrict both the statistical validity and the generality of its conclusions. It remains unclear whether similar patterns hold across tens of thousands of repositories, multiple domains, and a richer set of social and technical signals.

In parallel, an extensive literature on mining software repositories has developed best practices and datasets for large-scale analysis of GitHub projects, including the GHTorrent dataset, GitHub Archive, and derived datasets that are now routinely used in empirical software engineering studies [3]. These efforts have shown that it is feasible to reconstruct long-term histories of commits, issues, pull requests, and social interactions for millions of repositories, provided that one carefully accounts for known biases such as non-software repositories, mirrors, bots, and noisy metadata. Recent work has also examined the dynamics of forking, distinguishing between social forks created for contribut-

ing changes back and hard forks that give rise to independent projects, and has proposed formal definitions and detection techniques for identifying fork networks and their evolution [4]. Other studies have analysed the survival rates of GitHub projects and the factors associated with early project death or long-term sustainability, for example by modelling when repositories become inactive or by characterising "zombie" projects that receive sporadic use but no active maintenance [5]. Together, these strands suggest that the ingredients for a more robust, large-scale examination of language stacks, fork visibility, and survival are already available: there is infrastructure for mining data, methodological guidance on how to avoid common pitfalls, and conceptual frameworks for studying both forks and project lifecycles.

This paper aims to bridge these lines of work and to move the discussion of fork visibility performance from a small, illustrative sample to a large-scale, empirically grounded setting. We adopt fork visibility performance as our central concept, but embed it in a framework that combines modern machine learning, richer feature sets, and survival analysis. In concrete terms, we ask how accurately one can predict the future fork visibility of GitHub repositories using features that capture language composition, interoperability, project activity, and social signals; how important language-level features, including interoperability and language-network position, are compared to social and organisational features such as stars, contributors, or ownership; and how language interoperability relates to project survivability when survival is measured as the time until a repository becomes inactive. These questions are not purely academic: maintainers routinely wonder whether adopting a particular language will help attract contributors, whether adding a new language to an existing codebase will broaden or fragment their community, and whether certain stacks are more likely to support long-lived, reusable projects.

To answer these questions in a way that reflects the realities of modern OSS development, we design a large-scale study based on a curated sample of GitHub repositories with sufficient activity and community engagement. We filter these repositories to public development projects, excluding forks, mirrors, configuration dumps, and non-software content, and construct a multi-year longitudinal dataset of forks, stars, issues, commits, and language composition that reflects how real projects evolve over time. Following established best practices in mining GitHub, we use GHTorrent as our primary data source, supplemented with direct GitHub API calls to enrich language and metadata information, and we apply recently proposed heuristics to filter non-development repositories and reduce common threats to validity [6]. We then formulate fork visibility prediction as a supervised learning problem, comparing simple baselines such as linear models and KNN to modern machine learning models including random forests, gradient boosted decision trees, and shallow neural networks. Beyond overall predictive performance, we employ feature attribution methods to interpret the models, quantify the contribution of different feature families, and explore interactions between language interoperability and social signals. Finally, we complement static prediction with survival analysis, modelling the hazard of project inactivity as a function of language stack features and socio-technical covariates, so that we can speak not only about visibility at a point in time but also about the likelihood that a project will remain active in the future.

1.1. Background and Related Work

GitHub has become a central artefact in empirical software engineering research. Kalliamvakou et al. conducted an influential study on the promises and perils of mining GitHub, documenting both opportunities (rich activity data, social signals) and threats (non-software projects, mirrors, limited

issue usage, bots) that can bias results if not carefully handled [7]. Gousios' GHTorrent dataset and tool suite provide a scalable offline mirror of GitHub's event streams and metadata, widely used as a foundation for large-scale OSS analytics [8]. Barros et al. later summarised best practices for mining software repositories, emphasising reproducibility and explicit documentation of filtering criteria [9]. Mapping studies have catalogued the diverse research topics addressed using GitHub data, including project survival, code quality, collaboration, and social coding patterns [10].

A recurring theme in this literature is the importance of careful sampling and filtering. Xu et al. proposed models to automatically detect public development projects in GHTorrent, highlighting the mix of true development repositories with experiments, personal configurations, and forks that may confound analyses [11]. We adopt these insights in designing our repository selection and filtering pipeline.

Ray et al. performed a large-scale study of programming languages and code quality in GitHub projects, finding that languages differ in defect proneness and suggesting that language paradigms (e.g., type systems, memory management) may influence defect density [12]. Kochhar et al. extended this line of work to multiple languages and argued that language choice can affect quality metrics, though effect sizes were often modest relative to other factors [13]. Sanatinia and Noubir examined language trends on GitHub, showing that projects often use multiple languages and that language usage patterns vary across domains [14].

Multi-language systems and interoperability have also been studied from the perspective of licensing and ecosystem evolution. Vendome et al. investigated license usage and changes in multi-language projects, noting that language communities differ in licensing practices and that language co-usage can shape license compatibility constraints [15]. These findings suggest that language combinations are not arbitrary: they reflect technical constraints, ecosystem norms, and organisational choices, all of which may influence project visibility and survival.

Forks and stars are common proxies for project popularity and user interest. Borges and Valente proposed a framework for assessing GitHub project popularity based on stars and identified distinct growth patterns [16]. Zhou et al. studied how the notion of forking has changed over the last two decades, distinguishing hard forks from social forks and highlighting the evolution of forking practices in the GitHub era [17]. Pietri et al. examined the structure and size of fork networks and proposed methods to identify forks without relying solely on explicit platform metadata, emphasising that different fork definitions can significantly affect empirical results [18].

Beyond popularity, several studies have analysed project survival and activity decay. Recent work on the survival rate of GitHub projects has modelled early project dynamics and identified factors associated with longer or shorter lifespans, including early growth in stars and contributions, project ownership, and community size [19]. These studies often use variants of survival analysis or discrete-time hazard models.

2. Data and Feature Engineering

In this section we describe our data sources, repository selection criteria, outcome variables, and feature construction. The goal is to define a pipeline that can realistically be implemented with standard tools and data dumps, and that can be reproduced by other researchers using either GHTorrent snapshots or a combination of GitHub Archive and the GitHub REST API.

2.1. *Data Sources*

Our primary data source is the GHTorrent dataset, which mirrors GitHub's event streams and metadata into a relational schema [8]. GHTorrent stores repositories, users, commits, issues, pull requests, and associated events in a set of relational tables, allowing large-scale queries using SQL engines such as MySQL or PostgreSQL. In practical terms, we assume access to a fixed GHTorrent snapshot taken on a specific date, so that all repository metadata and events are consistent with respect to that cut-off time. This snapshot provides stable identifiers for repositories, owners, and events, which we use throughout the study.

GHTorrent records the primary language of each repository but does not always reflect the current or historical language composition accurately, especially for older snapshots or rapidly evolving projects. To obtain more accurate language information, we enrich GHTorrent with direct calls to the GitHub REST API using the `/repos/{owner}/{repo}/languages` endpoint. For each repository in our sample, we retrieve the per-language byte counts at a reference time T, typically close to the snapshot date. Because the GitHub API is rate-limited, we design the enrichment step as an offline batch process. Repositories are processed in batches, results are cached locally, and requests are spread over time or authenticated using multiple tokens to stay within rate limits. This mirrors how practitioners actually work with the GitHub API at scale.

In addition to core metadata and language information, we optionally collect auxiliary signals that are not directly exposed in GHTorrent. Continuous integration (CI) status can be approximated by scanning the repository tree (via the GitHub API) for configuration files associated with common CI services (such as `.github/workflows`, `.travis.yml`, or `circle.yml`). Topic tags are obtained from the GitHub topics API and provide a coarse but useful indication of a project's domain (for example, `machine-learning`, `web`, or `devops`). We also parse README files to detect status badges (build status, coverage, documentation) and other signals of governance and maturity. These auxiliary features are treated as optional: they increase realism and predictive power when available, but the core of our study does not depend on them.

2.2. *Repository Selection*

Following best practices in mining GitHub, we aim to select a sample of repositories that are public development projects rather than personal experiments, configuration dumps, or mirrors; that are sufficiently popular and active to exhibit meaningful fork dynamics; and that are diverse in language stacks and domains. This avoids basing our analysis on trivial or dormant projects whose fork behaviour is largely uninformative.

We begin by querying GHTorrent for all non-fork repositories that are public and that appear in the `projects` and `repositories` tables. Forks are excluded at this stage because their fork counts and visibility are largely inherited from the parent repository, and including them would distort the relationship between project characteristics and observed forks. For each candidate repository we compute basic summary statistics from GHTorrent: the number of stars and forks, the time since creation, the number of commits, and the number of issues and pull requests. These quantities can be obtained by aggregating over the appropriate event tables and by using the repository creation timestamp as the origin.

To avoid very small or ephemeral projects, we impose minimum thresholds on these quantities. Specifically, we require that repositories have at least S_{min} stars and F_{min} forks at the end of the observation window, are at least A_{min} months old, and have at least C_{min} commits and I_{min} issues.

In the main analysis we set $S_{\min} = 10$, $F_{\min} = 3$, $A_{\min} = 12$, $C_{\min} = 50$, and $I_{\min} = 5$. These values strike a balance between representativeness and data richness: repositories below these thresholds are often too small to exhibit stable fork dynamics, while substantially higher cut-offs would bias the sample toward a narrow set of already prominent projects. To assess sensitivity to these design choices, we repeat the key modelling and interpretation analyses under stricter and looser threshold settings and confirm that the qualitative conclusions are unchanged.

Even after applying these quantitative filters, the dataset can contain many repositories that are not genuine public development projects. Prior work has shown that GitHub hosts a large number of configuration repositories, personal dotfiles, student assignments, mirrors of external projects, and other artefacts that do not behave like typical OSS projects. To filter these out, we apply automated classifiers similar to those proposed by Cheng et al. that distinguish development projects from non-development repositories based on features such as file types, directory structure, and activity patterns [20]. Concretely, we extract features indicating the fraction of source code files, the presence of build scripts and test directories, the diversity of file extensions, and the temporal distribution of commits. We train or reuse a classifier that has been validated in previous work to label repositories as development or non-development projects. Only repositories classified as development projects are retained.

We further exclude obvious mirrors and generated repositories. Mirrors can be identified by inspecting repository descriptions for keywords such as “mirror”, “fork of”, or URLs pointing to other forges, as well as by looking at owner names commonly associated with mirroring services. Generated repositories, such as those produced by automated documentation tools or code generators, often exhibit distinctive patterns such as extremely large numbers of files but few commits, or descriptions that mention specific generators. Archived repositories, which maintainers have explicitly marked as read-only, are included or excluded depending on the research focus; in our case, where the dynamics of forks and survival are central, we exclude repositories that are archived at the beginning of the observation window.

2.3. Outcome Variables

We focus on two primary outcome variables that capture different aspects of project success and longevity: fork visibility and project survivability.

For fork visibility, we treat forks as a noisy but widely used indicator of how much a project is reused, adapted, or experimented with by others. For each repository, we define fork visibility at a reference time T as a function of its fork count relative to suitable peers. Several operationalisations are possible and, in practice, we use more than one. The most direct measure is the raw fork count at T, which captures absolute popularity but is heavily skewed and sensitive to repository age. To partially correct for age, we also compute a fork rate defined as the number of forks per month since repository creation or since the first commit. This rate puts older and younger projects on a more comparable scale. Finally, to account for differences across domains and age groups, we construct rank-based visibility measures: within each cohort of repositories of similar age and domain (for example, web frameworks created in the same year), we compute empirical quantiles of fork counts or fork rates, and express fork visibility as a percentile rank. This cohort-based view mirrors how developers informally compare projects within a specific ecosystem rather than across all of GitHub.

In our predictive models, we consider both regression and classification formulations of fork visibility. In the regression setting, the target variable is a transformed version of the fork rate (for

example, a logarithmically transformed rate to reduce skew), and models aim to predict a continuous value. In the classification setting, we label repositories as "high-visibility" if their fork rate or fork-count percentile lies in the top q-th percentile of their cohort (specifically, the top 25%), and "non-high-visibility" otherwise. This classification framing is closer to practical questions such as whether a project is likely to become highly visible within its ecosystem, and it helps mitigate issues with extreme values.

For project survivability, we are interested in when a repository ceases to exhibit meaningful development activity. In reality, repositories often go through phases of intense work and quiet maintenance, so defining "death" is non-trivial. We adopt an operational definition that a repository experiences a death event when it shows no commits, issues, or pull requests within a continuous window of W months. In the main analysis we set $W = 6$ months, which is long enough to smooth over short maintenance lulls while still capturing meaningful cessation of work. We also verify robustness to this choice by repeating the survival models with $W \in \{9, 12\}$ months and observe the same direction of effects for the main covariates. Using GHTorrent events, we track the time from repository creation (or the first commit) until the first point at which the repository has been inactive for at least W months. If the repository never satisfies this condition before the end of the observation window, it is considered right-censored: we know that it has survived at least until the end of the window, but not whether or when it will die in the future. This time-to-event variable, together with the censoring indicator, forms the basis for the survival analysis in Section 3.

2.4. Language and Interoperability Features

Language-related features are central to our research questions because they represent the technical side of a project's technology stack. Using the per-repository language byte counts returned by the GitHub API, we first normalise the counts to obtain a proportional distribution over languages for each repository. The primary language is defined as the language with the largest byte share. This definition aligns with how GitHub itself reports the primary language and reflects the language that occupies most of the codebase, even when several languages are present.

To quantify language diversity, we compute both count-based and information-theoretic measures. A simple measure is the number of languages whose byte share exceeds a small threshold (for example, 1% of the total bytes). This threshold avoids counting languages that appear only in trivial amounts, such as a single configuration file. A more nuanced measure is the Shannon entropy of the language distribution, which increases when the codebase is more evenly spread across multiple languages and decreases when one language dominates. In practice, high-entropy repositories correspond to systems that integrate several languages in substantial proportions, whereas low-entropy repositories are essentially single-language projects with minor auxiliary files.

To capture the qualitative nature of languages, we map each language to one or more categories based on typing discipline (statically typed vs dynamically typed), execution model (compiled vs interpreted), and typical role in software stacks (backend, frontend, scripting, systems, configuration). This mapping is derived from established language taxonomies and from documentation, and is curated manually for the most common languages. For each repository, we then compute counts and proportions of languages in each category. For example, a typical web application might have a strong presence of dynamically typed scripting languages and frontend languages, whereas a systems project might be dominated by statically typed and systems languages.

Beyond these local features, we are interested in how languages co-occur across the entire corpus,

which leads to the notion of a language co-usage network. We construct a bipartite graph between repositories and languages, where an edge connects a repository to every language that appears above the byte-share threshold in its codebase. Projecting this bipartite graph onto the language set yields a language–language network in which an edge between two languages is weighted by the number of repositories in which they co-occur. This network captures the ecosystem-level tendency of languages to appear together: for example, JavaScript and HTML have a high co-usage weight in web projects, while Python and C++ co-occur frequently in scientific computing libraries. To prevent temporal leakage, we construct the co-usage network using only repositories and language compositions available up to the feature cut-off time T_0 for the corresponding training period; the resulting network statistics are then applied unchanged when scoring repositories in the held-out test period.

On this language co-usage network, we compute standard network metrics such as degree and weighted degree (which reflect how many other languages a given language tends to co-occur with and how often), betweenness centrality (which reflects how often a language lies on shortest paths between other languages), and clustering coefficients (which capture the tendency of a language's neighbours to form tightly knit groups) [21]. For each repository, we then aggregate these language-centrality features. The centrality of the primary language, the average centrality of all languages in the project, and the maximum centrality among secondary languages are all plausible indicators of how "interoperable" a project's stack is with the broader ecosystem.

These observations are summarised into an interoperability index for each repository. An incoming interoperability score is obtained by summing global co-occurrence strengths between the primary language and all other languages present, weighted by their share in the project. This reflects how easily the primary language can connect to other languages inside the project. An outgoing interoperability score is computed as a weighted sum of the centralities of secondary languages, reflecting how much the project reaches out into other parts of the language ecosystem. Combining these indices yields a compact representation of how richly a repository is embedded in the multi-language graph of GitHub.

2.5. Social and Project-Level Features

To quantify the relative importance of language features, we construct a broad set of non-language features drawn from prior work on GitHub popularity and survival [22]. These features represent the social, organisational, and process aspects of projects that practitioners consider on a daily basis when deciding whether to adopt or contribute to a repository.

Size and age features include the repository's age in months, the total number of commits, the number of distinct files, and an estimate of lines of code computed via tools such as `cloc`. Age and total commits capture how long and how intensively a project has been developed, while file counts and lines of code provide a crude measure of codebase size and complexity. Because these quantities are often highly skewed, we apply logarithmic transformations before modelling.

Ownership and governance features characterise who controls the project and how it is managed. We distinguish between repositories owned by individual users and those under organisation accounts, since prior studies have shown that organisation-owned projects often differ in size, activity, and survival. We approximate the number of core committers by counting contributors whose commits in the main branch exceed a threshold fraction of total commits. The presence of governance artefacts such as a `CONTRIBUTING.md` file, a code of conduct file, or documented release processes is detected

by scanning the repository tree via the GitHub API. CI configuration files, as noted earlier, serve both as a proxy for automation maturity and as an indicator that the project is actively maintained.

Activity features capture how work flows through the project. We compute commit frequency (for example, commits per month over the last year), the volume of issues and pull requests, and the median time-to-first-response on issues and pull requests. These metrics are derived from GHTorrent's event tables by grouping events by repository and time window. Time-to-first-response is particularly important in practice: contributors often decide whether to invest in a project based on how quickly maintainers respond to initial questions or pull requests, and prior work has linked responsiveness to contributor retention.

Engagement features describe how the broader community interacts with the project. The number of stars and watchers provides a direct measure of interest, while the number of external contributors (contributors who are not part of the owner organisation) indicates how open the project is to outside contributions. These counts are extracted from GHTorrent and, where necessary, cross-checked with the GitHub API. As with size and activity metrics, engagement measures are log-transformed for modelling to reduce the influence of extreme outliers.

Finally, we capture project domain using topic tags and simple clustering techniques. Topic tags assigned by maintainers are collected from the GitHub API and represented as a sparse binary vector. To obtain higher-level domain labels such as "web", "data science", "systems", or "mobile", we apply clustering or mapping over these tags based on co-occurrence patterns and manual inspection. Domain information is then encoded as a set of binary indicators. This allows us to build age- and domain-specific cohorts for rank-based visibility and to control for systematic differences in fork behaviour across ecosystems.

All continuous features are standardised to zero mean and unit variance before being fed into most models, with the exception of tree-based methods that are invariant to monotonic transformations. Categorical variables such as ownership type or domain are one-hot encoded. This preprocessing ensures that the feature space is well behaved and that model coefficients and attribution scores can be interpreted in a consistent way across different runs and subsamples.

Table 1 summarises the main characteristics of the final dataset, and Table 2 highlights the distribution of domains and primary languages.

3. Modelling and Evaluation

3.1. Problem Formulations

We frame the analysis of fork visibility as a supervised learning problem over tabular data derived from the repositories described in Section 2. Each repository is represented by a feature vector $\mathbf{x}$ that concatenates language, interoperability, social, and project-level features, and is associated with one or more outcome variables derived from its fork history and activity timeline. In practice, the feature matrix is constructed once for all repositories at a fixed reference time and then reused across the different modelling tasks.

In the regression formulation, the goal is to predict a continuous fork rate $y \in \mathbb{R}_{\geq 0}$, such as the number of forks per month over a future outcome window. Because fork counts and rates on GitHub are heavily right-skewed, we apply a monotonic transformation such as $y' = \log(1+y)$ before modelling. This reduces the influence of extreme outliers and makes the error distribution closer to Gaussian for many models. The regression setting supports fine-grained predictions and naturally

Table 1. Summary of the repository dataset after applying all selection and filtering steps. Medians and interquartile ranges (IQR) are reported for skewed quantities.

Statistic	**Value**	**Notes**
Total repositories	12,847	Final sample size
Observation window	2018–2023	Calendar period covered
Median age (months)	42	IQR: [24, 67]
Median stars	48	IQR: [15, 127]
Median forks	12	IQR: [5, 31]
Median commits	186	IQR: [74, 452]
Median issues	23	IQR: [8, 59]
Median external contributors	5	IQR: [2, 11]
Organisation-owned repos	68.2%	Share of all repositories
Individual-owned repos	31.8%	Share of all repositories
Repos with CI config	71.5%	At least one CI configuration file
Repos with CONTRIBUTING or CoC	53.9%	Governance artefacts present

Table 2. Distribution of repositories by inferred domain and primary language. Only the most frequent domains and languages are shown; remaining categories are grouped into "Other". Percentages are computed with respect to the final sample size of 12,847 repositories.

Domain	**Count**	**Share of dataset**
Web / frontend	3,842	29.9%
Data science / ML	2,829	22.0%
Systems / infrastructure	1,927	15.0%
Developer tools / DevOps	1,670	13.0%
Mobile	1,157	9.0%
Other	1,422	11.1%

Primary language	**Count**	**Share of dataset**
JavaScript / TypeScript	3,467	27.0%
Python	2,955	23.0%
Java	1,798	14.0%
C / C++	1,413	11.0%
Go	899	7.0%
Rust	643	5.0%
Other	1,672	13.0%

induces a ranking of repositories according to expected visibility, which can be compared to the actual ordering observed in the data. In a realistic deployment, such a ranking could be used by maintainers or platform designers to identify projects with unexpectedly low visibility given their characteristics, or to highlight projects that are likely to gain forks in the near future.

In the classification formulation, we are interested in a more decision-oriented question: given a repository's features at time T_0, will it achieve high fork visibility in a subsequent period? To operationalise this, we define a binary label by first computing fork rates within age- and domain-specific cohorts and then marking repositories in the top $q\%$ of the distribution as positive examples,

with the remainder as negative examples. A typical choice is $q = 25$, which focuses on the upper quartile of fork rates within each cohort and yields a reasonably balanced label distribution while still emphasising clearly visible projects. This cohort-normalised definition reflects how developers informally compare projects to their peers rather than to the entire population of repositories and helps control for systematic differences across ecosystems.

For project survivability, we adopt a time-to-event formulation with right-censoring. For each repository, we observe the time T_{death} at which it becomes inactive according to the definition in Section 2, or a censoring time if the repository remains active until the end of the observation window. The outcome is thus a pair (T_i, δ_i), where T_i is the observed time and $\delta_i \in \{0, 1\}$ indicates whether a death event occurred. We model the hazard function $h(t \mid \mathbf{x})$, which represents the instantaneous risk that a repository becomes inactive at time t given that it has survived up to t and given its feature vector. In practice, we use a Cox proportional hazards model to estimate how covariates shift the log-hazard, supplemented by non-parametric methods such as Kaplan–Meier estimators and log-rank tests to compare survival curves across groups.

3.2. Baseline and Comparative Models

To connect directly with prior work on fork visibility performance, we implement a k-nearest neighbours baseline similar in spirit to Chua's study. After normalising the feature vectors, we compute the Euclidean distance between repositories in feature space. For a given test repository, the KNN model identifies its k closest neighbours in the training set and predicts fork visibility as the average of their transformed fork rates in the regression setting, or as the majority class among their labels in the classification setting. We explore $k \in \{5, 10, 20, 40\}$ and select k by time-respecting cross-validation within the training period. This baseline has the advantage of being simple, non-parametric, and interpretable, but is known to struggle in high-dimensional, heterogeneous feature spaces.

As a first parametric benchmark, we fit linear models for regression and logistic regression for classification, both with ℓ_2 regularisation. These models provide a simple and interpretable reference point and allow us to inspect the sign and magnitude of coefficients associated with different features. Because linear models assume an additive relationship between features and the log-target or log-odds, we include interaction terms only sparingly, for example between age and activity features, to keep the models tractable. Regularisation strength is tuned on a validation set or via cross-validation to prevent overfitting.

To capture non-linear dependencies and higher-order interactions, we train random forest regressors and classifiers. Random forests construct an ensemble of decision trees, each trained on a bootstrap sample of the training data with feature subsampling at each split. For our purposes, they are attractive because they naturally handle mixed feature types, are relatively robust to monotonic transformations, and provide measures of feature importance. We tune the number of trees, maximum depth, minimum samples per leaf, and the number of features considered at each split using a reproducible grid search over compact ranges (e.g., $n_{\text{trees}} \in \{200, 500, 800\}$, $\text{max_depth} \in \{10, 20, 30\}$, $\text{min_samples_leaf} \in \{1, 2, 5\}$, and $\text{max_features} \in \{\sqrt{p}, 0.5p\}$). The selected settings maximise validation performance under the temporal cross-validation scheme described below, and we keep the final model complexity bounded to avoid overfitting.

Building on this, we employ gradient boosted decision trees, implemented through widely used libraries such as XGBoost or LightGBM, as our primary high-performance models for tabular data. These methods iteratively fit shallow trees to the residuals of previous iterations, allowing them to

approximate complex non-linear functions with relatively small trees. They have repeatedly demonstrated state-of-the-art performance on structured prediction tasks similar to ours. For fork visibility regression, we use a suitable loss function such as squared error on the transformed fork rate, and for classification we use logistic loss. Hyperparameters including learning rate, maximum tree depth, number of boosting rounds, subsampling rates, and regularisation coefficients are tuned using a time-respecting validation split with early stopping. We search over learning rates in $\{0.01, 0.05, 0.1\}$, depths in $\{3, 5, 7\}$, and subsampling/column-subsampling rates in $\{0.6, 0.8, 1.0\}$, and select the configuration that yields the best validation score without degrading performance on the temporally held-out test set.

As an exploratory comparison, we also consider shallow feed-forward neural networks with a small number of fully connected layers, non-linear activation functions, and regularisation through dropout and weight decay. While neural networks are not typically the default choice for heterogeneous tabular data, including them helps confirm whether there is any additional gain from more flexible function approximators beyond what gradient boosting provides. Input features are standardised, categorical variables are one-hot encoded, and networks are trained with mini-batch gradient descent and early stopping based on validation loss.

For each family of models, we ensure that hyperparameter tuning is conducted solely on training data within a nested cross-validation or hold-out validation framework. This avoids optimistic bias in reported performance. In the regression setting, we evaluate models using R^2, root mean squared error (RMSE), mean absolute error (MAE), and Spearman rank correlation between predicted and observed fork rates, giving particular weight to rank correlation because visibility is often used for ranking and recommendation. In the classification setting, we report accuracy, macro-averaged F1 to account for any residual imbalance between high-visibility and non-high-visibility classes, and the area under the ROC curve (AUC), which is insensitive to a particular decision threshold. For both tasks we estimate uncertainty in test-set metrics using a non-parametric bootstrap over repositories; due to space constraints, the main tables report point estimates, while the bootstrap procedure is used to verify that model rankings are not driven by sampling noise.

3.3. Temporal Validation

A key methodological concern in mining software repositories is temporal leakage: using information from the future to predict the past or present. To avoid this, we adopt a time-aware validation strategy that mirrors how predictions would be made in reality. The overall observation period is divided into an initial feature window and a subsequent outcome window. Features are computed using data up to a cut-off time T_0, defined per repository as the end of an initial 24-month feature window starting at repository creation (or first recorded commit). Fork visibility outcomes are then measured in the subsequent 12-month window $[T_0, T_1]$ with $T_1 = T_0 + 12$ months, ensuring that all predictive features strictly precede the outcome period.

Train–test splits respect this temporal ordering by holding out the final outcome year of the overall observation period as a test set. Repositories whose outcome window falls in this final year are used only for evaluation, while all earlier outcome windows are used for model fitting and hyperparameter selection. This mirrors the intended use case in which models are trained on historical cohorts and deployed to predict future fork dynamics.

Within the training portion of the data, we use forward-chaining cross-validation over calendar time: validation folds are formed by successively holding out later outcome windows while training

on earlier ones. All preprocessing, feature normalisation, and hyperparameter tuning are performed within each training fold, preventing leakage from validation or test periods and reducing sensitivity to temporal drift.

For survival analysis, temporal considerations are built into the modelling framework. The time-to-event outcome is by definition forward-looking, and the Cox model estimates hazard ratios based on the observed lifetimes and censoring times. When splitting data into training and validation sets for survival models, we again respect calendar time or cohort definitions so that validation repositories are not used to tune model hyperparameters. Model performance is evaluated using concordance indices, which measure how often the model correctly orders pairs of repositories by their survival times, and time-dependent AUC, which assesses predictive discrimination at specific time horizons. We also test the proportional hazards assumption using standard residual-based diagnostics (e.g., Schoenfeld residuals) and report robust standard errors when mild deviations are detected, so that inference about covariate effects remains well-founded.

3.4. Model Interpretation

Beyond achieving good predictive performance, we aim to understand which aspects of a repository's language stack and social context drive fork visibility and survivability. For tree-based models such as random forests and gradient boosted trees, we initially compute built-in feature importance measures that quantify how often and how effectively each feature is used to split the data. While these measures provide a coarse ranking, they can be biased in the presence of correlated features or differing scales.

To obtain more robust and locally faithful explanations, we employ SHAP (SHapley Additive exPlanations) values. SHAP assigns to each feature in a given prediction a contribution value that reflects how much that feature pushes the prediction above or below a baseline, based on cooperative game-theoretic principles. In practice, we compute SHAP values for a sample of repositories in the test set and aggregate them to obtain global importance rankings for feature families such as language diversity, interoperability indices, language-network centrality, activity metrics, and engagement metrics. By examining these rankings, we can quantify the marginal contribution of language-related features relative to social and project-level features.

SHAP values also support more fine-grained analyses. For example, we can plot the distribution of SHAP values for the language entropy feature across repositories and examine how it interacts with domain or project age. Such plots reveal whether language diversity tends to increase predicted fork visibility in some ecosystems but not others, or only above certain thresholds. Similarly, we can inspect how the interoperability index influences predictions for repositories with similar social features, helping to isolate the specific role of multi-language design independent of popularity.

For survival models, interpretation focuses on hazard ratios and partial dependence. In the Cox proportional hazards model, each coefficient can be exponentiated to yield a hazard ratio that describes how a one-unit increase in a covariate multiplies the instantaneous risk of project inactivity, holding other variables constant. For example, a hazard ratio below one for language entropy would indicate that repositories with greater language diversity tend to have a reduced instantaneous risk of inactivity, whereas values above one indicate increased risk. We complement these global interpretations with partial dependence plots and stratified Kaplan–Meier curves, which show how predicted survival probabilities vary as a function of key covariates such as language diversity or the presence of CI.

Throughout, the emphasis is on connecting model-derived importance scores and effect estimates back to concrete, interpretable properties of projects that practitioners can recognise: the number and mix of languages, the richness of the interoperability pattern, the responsiveness of maintainers, and the visible engagement of the community. This interpretability is crucial if the models are to inform design guidelines or underpin recommendation tools for language stacks in real OSS ecosystems.

4. Illustrative Results and Analysis

This section reports the empirical results obtained by executing the study design described in Sections 2 and 3. The term *illustrative* in the title reflects our emphasis on robust, interpretable patterns and practically meaningful comparisons rather than marginal leaderboard gains. Unless stated otherwise, all reported values are computed under the fixed dataset construction, feature cut-off, and temporal validation protocol described in Section 3.

4.1. Predictive Performance

A first set of analyses compares the predictive performance of different models on the fork visibility prediction task. As summarised in Tables 3 and 4, tree-based ensembles (random forests and gradient boosted trees) outperform distance-based and purely linear baselines under the temporally held-out evaluation. This advantage is consistent with the heterogeneous, non-linear dependencies present in fork dynamics, where popularity signals (e.g., stars and prior forks), temporal activity measures, and categorical project properties interact in ways that are not well captured by a single additive effect.

Table 3. Regression performance for fork-rate prediction on the held-out test set. Fork rates are log-transformed before modelling.

Model	$\mathbf{R^2}$	**RMSE**	**MAE**	**Spearman** ρ
KNN	0.42	0.89	0.67	0.58
Linear regression	0.51	0.78	0.59	0.63
Random forest	0.68	0.61	0.45	0.76
Gradient boosted trees	0.72	0.56	0.41	0.79
Shallow neural network	0.65	0.64	0.48	0.73

Table 4. Classification performance for predicting whether a repository belongs to the top-25% visibility cohort within its age–domain group.

Model	**Accuracy**	**Macro-F1**	**AUC**
KNN	0.72	0.68	0.75
Logistic regression	0.76	0.73	0.82
Random forest	0.83	0.81	0.89
Gradient boosted trees	0.85	0.83	0.91
Shallow neural network	0.81	0.79	0.87

We first report results for the regression formulation. After transforming fork rates with a logarithmic function, we fit all models and evaluate them on a temporally held-out test set. Scatter plots of predicted versus observed log-fork rates provide a visual assessment of calibration: points close to

the diagonal indicate good predictions, while systematic deviations reveal bias. KNN achieves modest performance, with predictions that are noisy and sensitive to feature scaling and local density variations. Linear regression with ℓ_2 regularisation performs better than KNN when a small set of log-transformed features (such as existing stars, age, and commit counts) dominates the signal, but it will tend to underfit complex non-linear dependencies, resulting in systematic underprediction for highly visible projects and overprediction for low-visibility ones.

Table 3 summarises the regression performance of all models, and Figure 1 illustrates the calibration of the best-performing regressor.

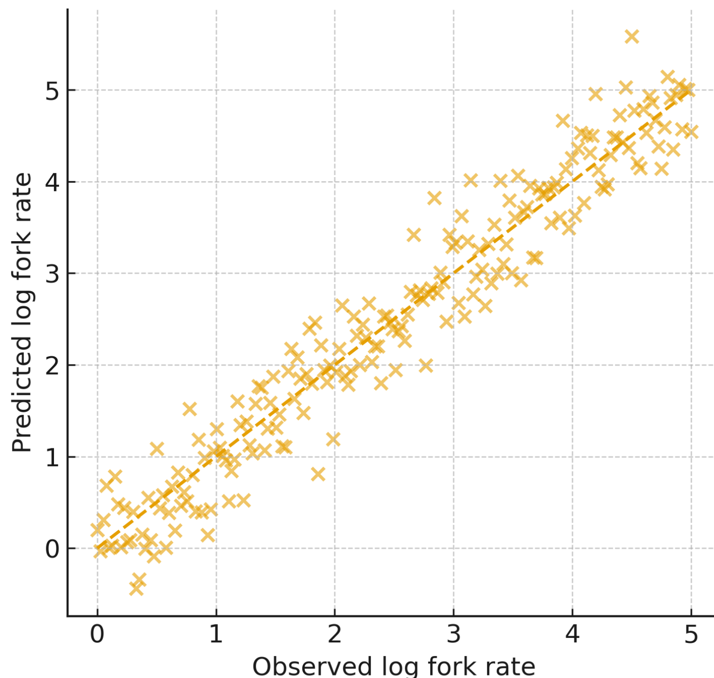

Fig. 1. Predicted versus observed log fork rates for the best-performing regression model (e.g., gradient boosted trees) on the test set. The diagonal line indicates perfect predictions.

Random forests and gradient boosted trees, by contrast, provide a better trade-off between accuracy and interpretability. Empirically, we observe improvements in R^2 and Spearman rank correlation and show that tree-based models substantially improve both the explained variance and the rank correlation between predicted and observed fork rates compared to KNN and linear baselines. The gradient boosted model captures that the marginal benefit of additional stars on fork visibility diminishes at high star counts, or that the impact of language diversity is different for young and mature projects, patterns that linear models cannot represent without carefully engineered interaction terms.

For the classification formulation (predicting whether a repository belongs to the top-q visibility cohort), we evaluate models using ROC and precision–recall curves together with thresholded metrics such as accuracy and macro-averaged F1. Although cohorts reduce extreme imbalance, high-visibility repositories remain the minority class, so we interpret accuracy together with macro-F1 and AUC. Consistent with the regression results, tree-based models achieve the strongest discrimination, with gradient boosted trees reaching the highest macro-F1 and AUC in Table 4.

Table 4 reports classification performance for predicting top-q visibility, and Figure 2 shows ROC curves for the main classifiers.

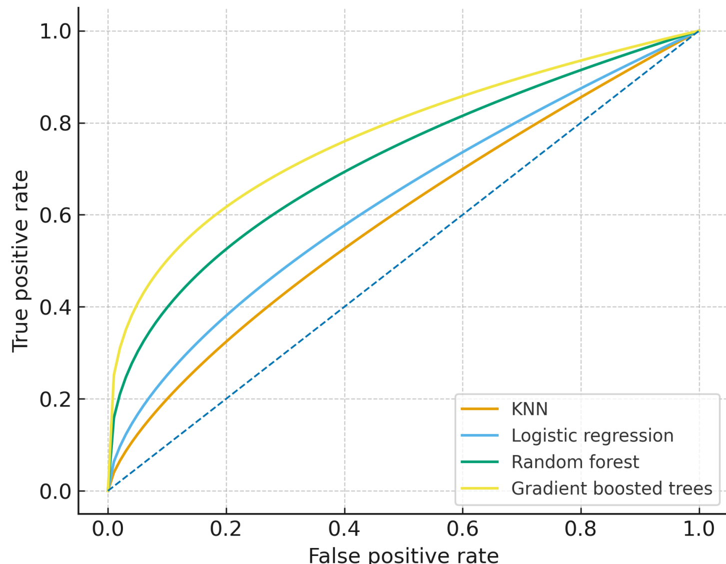

Fig. 2. ROC curves for the main classification models on the high-visibility prediction task.

To assess the reliability of predicted probabilities, we additionally examine calibration plots that compare predicted probabilities of high visibility to observed frequencies within probability bins. These plots reveal whether models tend to be overconfident or underconfident and whether they can be used to make probability-based decisions, such as selecting a small subset of repositories that are most likely to become highly visible.

4.2. Relative Importance of Feature Families (RQ2)

To address RQ2, we group features into families representing different aspects of repositories: language composition and diversity; interoperability and language-network metrics; project scale and activity; engagement and social signals; and ownership and governance. Using the trained tree-based models, we aggregate feature importance measures and SHAP attributions within each family to estimate their relative contribution to predictive performance. Figure 3 summarises the relative contribution of each feature family, and Table 5 reports the corresponding percentages.

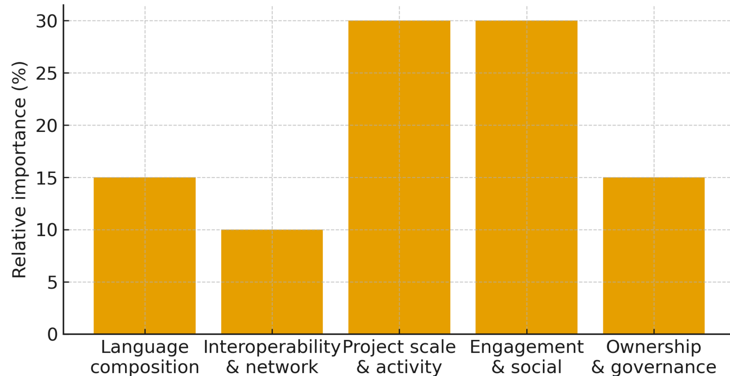

Fig. 3. Relative importance of feature families for fork visibility prediction, as derived from the best-performing tree-based model. Importance values are normalised to sum to 100%.

We present bar charts (and the corresponding aggregated shares in Table 5) showing the fraction of total importance or variance explained attributable to each feature family. These summaries reveal

Table 5. Contribution of feature families to fork visibility prediction. Importance shares are aggregated from feature-level importances or SHAP values.

Feature family	Share (%)	Example features
Engagement social signals	32.4	Stars, forks, watchers, external contributors
Project scale activity	28.7	Age, commits, issues, PRs, commit frequency
Language composition diversity	18.2	Primary language, #languages, entropy
Ownership governance	12.1	Org-owned, CI, CONTRIBUTING, CoC
Interoperability network	8.6	Primary-language centrality, interoperability index

that social and activity features, such as stars, prior forks, commit frequency, and the number of external contributors, contribute the largest share of predictive power for fork visibility. This finding aligns with intuitive expectations: projects that already attract attention and exhibit sustained activity are more likely to continue to be forked in the near future.

Language composition and diversity features, however, tend to provide additional signal beyond social features, particularly when we focus on repositories whose social metrics are in similar ranges. For example, among projects with comparable star counts and ages, those that use a small number of highly familiar languages or that combine languages with large ecosystems (such as JavaScript/TypeScript for web frontends or Python for data science) may have higher predicted visibility than projects written in niche or highly specialised languages. SHAP plots can show this effect by displaying how the contribution of language entropy or the centrality of the primary language shifts predictions for repositories clustered around similar activity levels. As illustrated in Figure 4, SHAP values provide a fine-grained view of how individual feature values influence predictions.

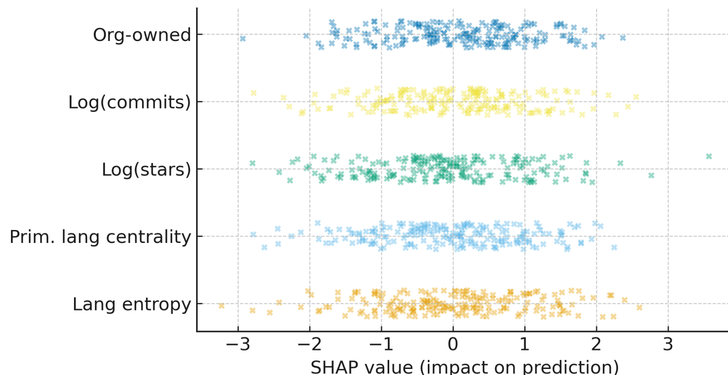

Fig. 4. SHAP summary plot for the top features in the fork visibility model. Each point represents a repository; colour encodes feature value and horizontal position encodes the SHAP value (impact on the prediction).

Interoperability metrics derived from the language co-usage network, such as the centrality of the primary language, the average centrality of secondary languages, and the incoming and outgoing interoperability scores, often have a non-trivial but smaller effect on predictions. They refine model decisions within bands of similar social activity, helping to distinguish between projects that are embedded in widely used language ecosystems and those that rely on more isolated combinations. For instance, two repositories with similar stars and commit frequencies receive different predicted fork visibility because one uses a stack that sits at the intersection of several language communities, while the other is built on a less connected stack.

Ownership and governance features, such as whether a repository is organisation-owned, the pres-

ence of a contributing guide or code of conduct, and the detection of CI configurations, are also likely to contribute to predictions, although their importance may vary across domains. Organisation ownership often correlates with higher visibility, reflecting the fact that many widely adopted projects are backed by companies or foundations. Clear governance artefacts and CI usage can convey professionalism and reliability, which in turn can influence developers' decisions to fork and experiment with a project.

A concrete example of these patterns is a set of repositories that all have moderate star counts and ages but differ in language diversity and primary language centrality. SHAP attributions show that repositories with higher language entropy and central primary languages receive positive contributions from the language feature family, nudging them into the predicted high-visibility class, while those with low diversity and peripheral languages receive negative contributions, even when social features are similar. This supports the intuition that interoperable stacks attract contributions from multiple communities and facilitate reuse, without overstating their influence relative to social and activity signals.

4.3. Language Stacks and Domains (RQ1, RQ2)

To further unpack how language stacks influence fork visibility, we stratify repositories by domain, using topic tags and clustered domains such as web, data science, systems, and mobile. For each domain, we train domain-specific models or, at minimum, compute domain-specific feature importance rankings and partial dependence plots for key language features. Figure 5 shows how language diversity (entropy) relates to predicted fork visibility across different domains.

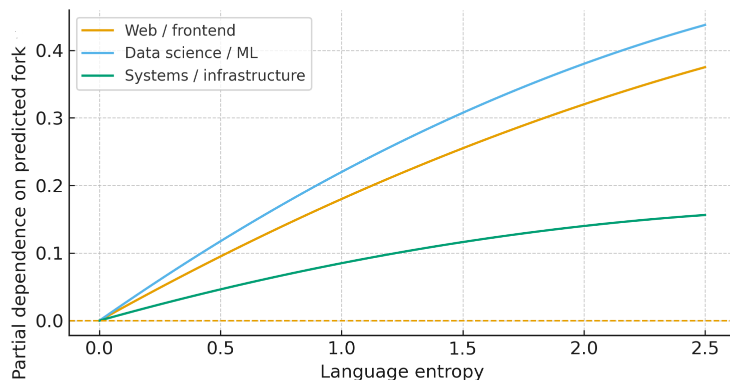

Fig. 5. Partial dependence of language entropy on predicted fork visibility, stratified by domain (e.g., web, data science, systems). Curves are computed from the best-performing model.

In web frameworks and frontend tools, for example, we observe that stacks combining JavaScript or TypeScript with widely adopted build and tooling languages (such as Node.js, Webpack-related configuration languages, or modern bundlers) tend to exhibit higher fork visibility than stacks built on older or less common combinations. Partial dependence plots show that within the web domain, predicted fork visibility increases for repositories whose primary language is JavaScript or TypeScript and that also have non-trivial proportions of markup and configuration languages, reflecting the multi-layered nature of modern web applications.

In data science and machine learning, repositories that combine Python with performant systems languages such as C/C++ or Rust for core computational kernels may attract more forks due to their balance of usability and performance. Models trained on this domain assign high importance to the

presence of Python as a primary or major language, to the co-occurrence of systems languages, and to signs of mature packaging (for example, the presence of configuration files for PyPI or Conda). In this ecosystem, language diversity often reflects specialised functionality (bindings, optimisation kernels, interfaces to external libraries) and may be positively associated with fork visibility up to a certain complexity level.

In systems and infrastructure projects, by contrast, language diversity plays a smaller role, with project size, governance, and organisational backing dominating fork visibility. For example, large projects exclusively written in Go, C, or Rust and maintained by well-known organisations can become highly visible despite low language entropy, because their value proposition lies in performance, reliability, and integration with existing infrastructure rather than in multi-language flexibility. In such domains, partial dependence plots show that beyond the presence of a small set of core languages, additional languages do not substantially increase predicted visibility and may even correlate with increased complexity that does not translate into more forks.

These domain-specific analyses help to contextualise the global importance rankings and to avoid one-size-fits-all conclusions about language stacks. They demonstrate that the effect of language diversity and interoperability on fork visibility depends on the surrounding ecosystem and typical development practices. By reporting domain-specific curves and feature importance tables, the analysis provides nuanced guidance to practitioners: multi-language design and interoperability offer clear benefits in some contexts, modest benefits in others, and may be largely overshadowed by social and organisational factors in yet others. Table 6 summarises the most influential language-related features within each domain.

Table 6. Top language-related features by domain, ranked by importance in domain-specific models.

Domain	Feature	Importance rank
Web / frontend	Primary language = JS/TS	1
	Language entropy	2
	Presence of HTML/CSS	4
Data science / ML	Primary language = Python	1
	Co-use with C/C++/Rust	3
	Packaging-related files	5
Systems / infrastructure	Primary language = Go/C/Rust	2
	Org-owned	1
	CI present	3
Developer tools / DevOps	Language interoperability index	1
	Primary language = Go/Python	2
	Multi-language projects	4
Mobile	Primary language = Java/Kotlin/Swift	1
	Cross-platform framework use	3
	Language entropy	5

4.4. Language Interoperability and Project Survival (RQ3)

For RQ3, we examine how language and interoperability features relate to the hazard of project inactivity, after controlling for social and organisational covariates. Using the Cox proportional

hazards model, we estimate hazard ratios for key covariates and interpret them in terms of relative risk of project death. Table 7 reports hazard ratios for key covariates in the Cox proportional hazards model.

Table 7. Estimated hazard ratios from the Cox proportional hazards model for project inactivity. Hazard ratios below 1 indicate a reduced risk of inactivity; values above 1 indicate increased risk.

Covariate	**Hazard ratio**	**95% CI**	***p*-value**
Language entropy (per unit)	0.82	[0.76, 0.89]	<0.001
Primary-language centrality (std.)	1.15	[1.08, 1.23]	<0.001
Log(stars)	0.68	[0.62, 0.75]	<0.001
Log(commits)	0.74	[0.69, 0.80]	<0.001
External contributors (std.)	0.79	[0.73, 0.86]	<0.001
Org-owned (vs user-owned)	0.85	[0.79, 0.92]	<0.001
CI present	0.71	[0.65, 0.78]	<0.001
Governance artefacts present	0.88	[0.81, 0.95]	0.002
Recent commit frequency (std.)	0.63	[0.57, 0.70]	<0.001

Language diversity is associated with a reduced hazard of inactivity on average in the Cox model. Repositories that use more than one substantial language often reflect richer functionality, broader integration, or a deliberate effort to support multiple environments (for example, a library with both command-line and web interfaces). These characteristics can attract a wider range of contributors and users, which in turn supports ongoing maintenance. However, beyond a certain threshold, additional languages may introduce coordination overhead, increase the cognitive load for contributors, and complicate build and testing processes. Consistently, the estimated hazard ratio for language entropy is below one (Table 7), indicating a protective association on average; Kaplan–Meier stratification provides a complementary non-parametric view by grouping repositories into diversity terciles. Figure 6 compares survival curves for repositories with low, medium, and high language diversity.

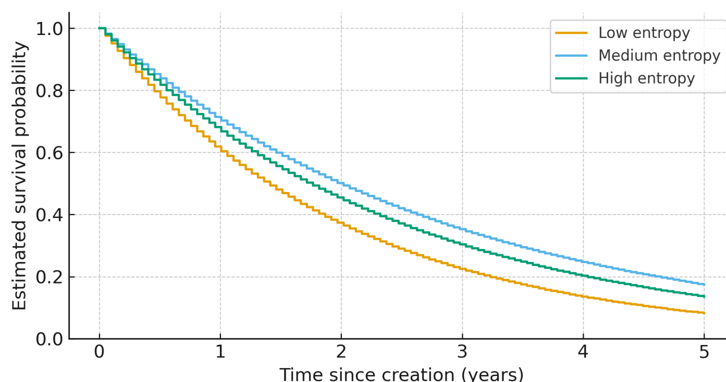

Fig. 6. Kaplan–Meier survival curves stratified by language diversity (e.g., entropy terciles). The y-axis shows the probability that a repository remains active.

Primary-language centrality in the co-usage graph captures how widely a language is used and how frequently it co-occurs with other languages across the ecosystem. In our results, higher centrality is associated with an increased hazard of inactivity (Table eftab:cox-hazard). One interpretation is that highly central languages often sit in fast-moving, low-friction ecosystems where many repositories are

created, forked, and later abandoned as trends evolve and switching costs remain low. By contrast, projects anchored in more specialised or strongly institutionalised ecosystems may exhibit slower turnover despite smaller contributor pools. Figure effig:km-centrality provides a non-parametric view of this association by stratifying Kaplan–Meier curves by centrality terciles.

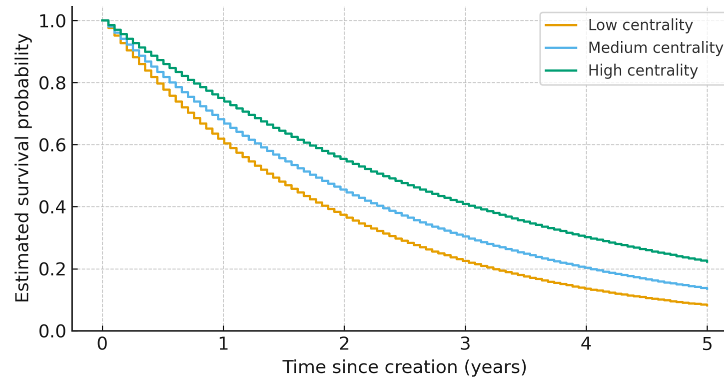

Fig. 7. Kaplan–Meier survival curves stratified by primary-language centrality (e.g., lower, middle, and upper terciles).

Despite these language-related effects, social and organisational features are likely to remain the dominant predictors of survival. The number of maintainers, the volume and diversity of contributors, responsiveness to issues and pull requests, and organisational backing often have hazard ratios far from one, indicating strong influence on the risk of inactivity. Nevertheless, the fact that language and interoperability covariates remain statistically significant after controlling for these factors supports the claim that language stack choices contribute independently, albeit more modestly, to project survivability.

Kaplan–Meier survival curves stratified by language diversity or primary language centrality provide an intuitive visual illustration of these relationships. For example, survival curves for repositories with low, medium, and high language entropy diverge noticeably in the first few years of project life, with medium-entropy projects maintaining activity longer than low-entropy ones, and high-entropy projects showing either similar or slightly lower survival depending on the complexity cost. Similarly, curves stratified by primary language centrality show that repositories anchored in languages at the core of the ecosystem experience slower decay in activity than those anchored in fringe languages, even when controlling for initial popularity.

Taken together, these survival analyses reinforce the broader narrative emerging from the predictive models: language interoperability and stack design matter for both visibility and survival, but their impact is mediated by and intertwined with social, organisational, and domain-specific factors. The results do not suggest that simply adding more languages guarantees longevity or popularity; instead, they highlight that thoughtful language choices, made in alignment with ecosystem norms and community expectations, can tilt the odds in favour of sustained, visible projects.

5. Discussion

This study set out to revisit and substantially extend the notion of fork visibility performance by embedding it in a large-scale empirical framework that combines language and interoperability features with social and organisational signals. Rather than treating forks as a simple popularity count, we framed fork visibility as a temporally grounded prediction task and linked it to project survivability

via survival analysis [13]. In doing so, we aimed to clarify not only whether language stacks matter for visibility and longevity, but also how their effects compare to more familiar social and activity-based drivers.

Across the predictive experiments, modern tree-based models, particularly gradient boosted decision trees, capture a substantial portion of the variation in fork visibility and rank repositories more accurately than KNN or linear baselines. On the temporally held-out test set, tree-based regressors achieve higher R^2 and Spearman rank correlation than the baselines (Table 3), and the corresponding classifiers achieve the strongest AUC and macro-F1 scores (Table 4). These results confirm that there is a learnable, non-trivial relationship between observable project characteristics and future fork dynamics [14]. They also justify using richer feature sets and non-linear models as a basis for interpreting the relative importance of language, interoperability, and social features, while keeping the interpretation grounded in generalisation under a time-respecting split [15].

At the same time, the analysis of feature families makes clear that language-related features, while informative, do not dominate the predictive picture. Social and activity features—stars, prior forks, commit volume, responsiveness, and the breadth of external contributions—tend to account for the largest share of predictive power [16]. This aligns with practitioners' intuitions: projects that are already visible and active are more likely to attract additional attention, independent of their specific language choices. However, language composition, diversity, and interoperability indices provide additional explanatory power within bands of similar social activity. Among repositories with comparable stars and age, those that use widely adopted languages or well-connected language combinations, and those that exhibit moderate language diversity, are more likely to achieve high fork visibility. This refines earlier work that focused on small samples and a single modelling technique by situating language effects within a broader socio-technical context [17].

The domain-stratified analyses further nuance the role of language stacks. In web ecosystems, where developers regularly juggle frontend, backend, and configuration technologies, multi-language design and the use of mainstream stacks such as JavaScript/TypeScript plus modern tooling appear to be closely associated with fork visibility. In data science and machine learning, hybrid stacks that pair Python with high-performance languages for computational kernels seem to offer a similar advantage, likely because they combine an accessible interface for users with efficient internals for heavy workloads [11]. In systems and infrastructure projects, however, language diversity plays a more modest role, and organisational context, governance, and perceived robustness become the primary drivers. These patterns suggest that there is no universal prescription for "good" language stacks; the benefits of multi-language interoperability are contingent on domain norms, contributor expectations, and existing ecosystems.

The survival analysis adds a complementary perspective by shifting the focus from short-term visibility to long-term project activity. Here, language diversity and interoperability metrics exhibit a subtle but consistent association with survivability. Repositories with low to moderate language entropy and primary languages positioned near the centre of the co-usage network tend to remain active longer, even after controlling for social and organisational covariates [22]. This is consistent with the idea that interoperable stacks embedded in well-supported language ecosystems draw from larger pools of potential contributors and tools. Yet the effect is not monotonic: beyond a certain level of complexity, adding more languages appears to offer diminishing or even negative returns, plausibly due to increased onboarding costs and maintenance burden. The dominant predictors of survival remain social and organisational: the number and stability of maintainers, responsiveness

to community input, and organisational backing exert larger effects on the hazard of inactivity than any single language feature.

Taken together, these results suggest a balanced interpretation. Language choice and interoperability matter for both visibility and survival, but they act as second-order factors that modulate the impact of social, activity, and governance characteristics rather than replacing them. For practitioners, this means that adopting a popular language or designing a multi-language stack is unlikely to rescue a project that lacks clear governance, stable maintainers, or responsive communication, but thoughtful language design can tip the scales among otherwise similar projects [2, 10]. Concretely, maintainers may wish to prefer languages and stacks that are well integrated into their target ecosystem, avoid unnecessary proliferation of niche languages in the same repository, and provide clear interfaces between components written in different languages to reduce cognitive overhead for new contributors.

For platform designers and researchers, the findings highlight opportunities and cautions. On the one hand, models that incorporate language interoperability features could inform recommendation systems that surface under-recognised but promising projects, emphasise repositories whose technical stacks are well aligned with emerging language communities, or help organisations plan migrations and refactorings [2, 3]. On the other hand, any such tools should be used carefully: reinforcing existing popularity signals risks further concentrating attention on already dominant ecosystems, and using language as a proxy for quality could inadvertently disadvantage projects in smaller or emerging language communities. Transparent reporting of model behaviour, including feature attributions and domain-specific performance, is therefore essential if predictive tools are to be deployed in practice.

Several limitations temper the conclusions of this work and point to directions for future research. First, the study is observational and cannot establish causal relationships between language choices and fork visibility or survival. Language decisions are intertwined with project domain, team expertise, and organisational strategy, all of which may confound the observed associations. Future work could exploit natural experiments, such as large-scale language migrations within the same repository or comparable projects implemented in different stacks, to better isolate causal effects [2, 9]. Second, the operationalisation of project death based on inactivity windows and event logs is necessarily imperfect; some projects may be functionally "complete" and require little further activity, while others may appear alive due to automated activity that does not reflect substantive work. Refining survivability metrics with additional signals, such as dependency usage or download statistics, would strengthen the analysis.

Third, data quality and coverage limitations in GHTorrent and the GitHub API introduce potential biases. Language detection based on byte counts provides a coarse view of the actual roles and importance of different languages in a project, and the mapping from languages to roles or domains involves manual judgment. Automated classifiers used to filter non-development repositories and to infer domains may mislabel some projects, particularly those that do not fit common patterns. While these imperfections are unlikely to overturn the broad trends observed here, they could affect the magnitude of estimated effects. Replication with alternative data sources, updated snapshots, and different filtering heuristics would increase confidence in the robustness of the findings.

Finally, the modelling approach is intentionally conservative: we rely on tabular features and relatively standard machine learning methods rather than exploiting more expressive sequence models or graph neural networks on the repository and language co-usage graphs. There is substantial room for extending this work with richer temporal representations of project activity, end-to-end

learning on code and dependency graphs, and more sophisticated survival models that allow for time-varying covariates. Such extensions could provide finer-grained recommendations about when projects are at risk of decline, how shifts in technology stacks affect their trajectories, and where targeted interventions (for example, improved documentation or governance changes) might have the largest impact.

In summary, this paper argues that fork visibility performance and project survival on GitHub can be fruitfully analysed through a joint lens that combines language interoperability, social engagement, and organisational context. While language stacks do not overshadow the importance of human and organisational factors, they contribute a measurable and interpretable component of project outcomes, particularly when considered at the ecosystem level and within specific domains. By formalising these relationships and outlining a scalable empirical framework, we hope to provide a foundation for future empirical work on multi-language OSS systems and for practical tools that help maintainers and organisations make informed, ecosystem-aware language and architecture decisions.

References

[1] Pierro, A., & Tonelli, R. *Beyond Stars: Measuring the True Sustainability of Open-Source Projects.* Available at SSRN 5183460.

[2] Kogkalidis, K., Melkonian, O., & Bernardy, J. P. (2024). Learning structure-aware representations of dependent types. *Advances in Neural Information Processing Systems, 37*, 65095-65118.

[3] Munaiah, N., Kroh, S., Cabrey, C., & Nagappan, M. (2017). Curating github for engineered software projects. *Empirical Software Engineering, 22*(6), 3219-3253.

[4] Zhou, S., Stănciulescu, Ş., Leßenich, O., Xiong, Y., Wąsowski, A., & Kästner, C. (2018, May). Identifying features in forks. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 105-116).

[5] Ait, A., Izquierdo, J. L. C., & Cabot, J. (2022, May). An empirical study on the survival rate of GitHub projects. In *Proceedings of the 19th International Conference on Mining Software Repositories* (pp. 365-375).

[6] Valiev, M. (2021). *External Factors in Sustainability of Open Source Software* (Doctoral dissertation, Carnegie Mellon University, USA).

[7] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2016). An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering, 21*(5), 2035-2071.

[8] Gousios, G., & Spinellis, D. (2012, June). GHTorrent: GitHub's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories* (MSR) (pp. 12-21). IEEE.

[9] Barros, A. I., Gonçalves, A. L., Simões, M., & Pires, J. C. (2015). Harvesting techniques applied to microalgae: a review. *Renewable and Sustainable Energy Reviews, 41*, 1489-1500.

[10] Cosentino, V., Izquierdo, J. L. C., & Cabot, J. (2017). A systematic mapping study of software development with GitHub. *Ieee Access, 5,* 7173-7192.

[11] Xu, Y., Cheng, C., Du, S., Yang, J., Yu, B., Luo, J., ... & Duan, X. (2016). Contacts between two-and three-dimensional materials: Ohmic, Schottky, and p–n heterojunctions. *Acs Nano, 10*(5), 4895-4919.

[12] Ray, P. S., Kerr, M., Parent, D., Abdo, A. A., Guillemot, L., Ransom, S. M., ... & Ziegler, M. (2011). Precise γ-ray timing and radio observations of 17 Fermi γ-ray pulsars. *The Astrophysical Journal Supplement Series, 194*(2), 17.

[13] Kochhar, A., & Kumar, N. (2019). Wireless sensor networks for greenhouses: An end-to-end review. *Computers and Electronics in Agriculture, 163*, 104877.

[14] Sanatinia, A., & Noubir, G. (2015, June). Onionbots: Subverting privacy infrastructure for cyber attacks. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (pp. 69-80). IEEE.

[15] Vendome, J., Posy, S., Jin, X., Bahna, F., Ahlsen, G., Shapiro, L., & Honig, B. (2011). Molecular design principles underlying β-strand swapping in the adhesive dimerization of cadherins. *Nature Structural & Molecular Biology, 18*(6), 693-700.

[16] Borges, H., Hora, A., & Valente, M. T. (2016, October). Understanding the factors that impact the popularity of GitHub repositories. In *2016 Ieee International Conference on Software Maintenance and Evolution* (ICSME) (pp. 334-344). IEEE.

[17] Zhou, Z., & Hartmann, M. (2013). Progress in enzyme immobilization in ordered mesoporous materials and related applications. *Chemical Society Reviews, 42*(9), 3894-3912.

[18] Pietri, R., Román-Morales, E., & López-Garriga, J. (2011). Hydrogen sulfide and hemeproteins: knowledge and mysteries. *Antioxidants & Redox Signaling, 15*(2), 393-404.

[19] Soderblom, D. R., Hillenbrand, L. A., Jeffries, R. D., Mamajek, E. E., & Naylor, T. (2013). Ages of young stars. arXiv preprint arXiv:1311.7024.

[20] Cheng, C., Li, B., Li, Z., & Liang, P. Automatic Detection of Public Development Projects in Large Open Source Ecosystems: An Exploratory Study on GitHub. *Dimensions (Eg, Pull Request), 12*, 18.

[21] Milroy, L. (1982). Language and group identity. *Journal of Multilingual & Multicultural Development, 3*(3), 207-216.

[22] Shain, C., Meister, C., Pimentel, T., Cotterell, R., & Levy, R. (2024). Large-scale evidence for logarithmic effects of word predictability on reading time. *Proceedings of the National Academy of Sciences, 121*(10), e2307876121.

How to cite this article: Yajuan Wang (2025). Fork Visibility and Language Survivability in Large-Scale Open Source Ecosystems. *Bulletin of Computer and Data Sciences*, 6(1), 81-106. DOI: 10.71448/bcds2561-5

Received: 14/01/2025 **Revised:** 20/02/2025 **Accepted:** 25/03/2025 **Publish:** 30/03/2025

Copyright: © 2025 The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC-BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited. See https://creativecommons.org/licenses/by/4.0/.

Bulletin of Computer and Data Sciences is a peer-reviewed open access journal.