

Towards Competence-Based Management for Open Source Software Projects

Sabahat Younas ✉ 

Independent Researcher, Lahore, Pakistan

Márcia Moraes ✉ 

Colorado State University, Fort Collins, CO, USA

Fabio Santos ✉ 

Colorado State University, Fort Collins, CO, USA

Abstract

Contributors to Open Source Software (OSS) projects are vital to maintaining the health of both communities and projects. However, the number of projects experiencing core contributors' disengagement has increased to the point that it risks the projects' survival. Finding new contributors to replace the workforce is challenging and time-intensive due to several factors, including a lack of precise knowledge about potential candidates' competences, which may need to be confirmed through interviews and exams. Previous studies provided indications of the contributor's competences, although they lack depth in understanding competence levels, which can result in poor knowledge about the contributor's capabilities. To address this gap, we assess contributors' competence by collecting code metrics related to source code from contributions. We also propose a competence model able to predict the competence level required to solve tasks. By properly assessing contributors' competences, we can identify and train candidates to replace core contributors. Our replication package, including code, data, and documentation, is available at <https://doi.org/10.5281/zenodo.21605122>

2012 ACM Subject Classification Software and its engineering → Open source model; Software and its engineering → Software maintenance tools; Computing methodologies → Feature selection

Keywords and phrases Contributor competence, MSR, project sustainability, AST, OSS, disengagement mitigation

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.65

Category Emerging Results, Vision & Reflection Track Paper

Supplementary Material *Software*: <https://zenodo.org/records/21605122>

1 Introduction

Open Source Software (OSS) exhibits increasing project counts, file counts, and churn, while the number of joiners remains stable or declines in most projects [34]. Many of the projects depend on a few core contributors, with numbers approaching 1 or 2 in about 70 of the most popular GitHub repositories [3], and projects tend to die when abandoned by maintainers due to vital technical debt [2].

Competence-based management is a key factor in Human Resources (HR) selection, development, planning, and performance management [25]; therefore, knowing the core contributors' competences can help recruit, onboard, and train new members. Competence is commonly described as a combination of attitude, skills, and knowledge [1]. However, it remains challenging to consistently identify, represent, and assess human competences [7]. Previous studies have demonstrated the importance of revealing the capabilities of the workforce and the existing tasks in OSS context. Liang et al. [22] used surveys to investigate skills that support the development of OSS projects. Dey et al. [10] showed that contributors



© Sabahat Younas, Márcia Moraes, and Fabio Santos;
licensed under Creative Commons License CC-BY 4.0

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

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 65; pp. 65:1–65:15



Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

are more likely to join projects aligned with their competence and have pull requests accepted. The onboarding of newcomers to OSS projects and the selection of tasks using proxies for skills were also the subjects of studies [28, 29]. However, previous studies are limited in depth since they ignored: differences between programming languages [10], security vulnerability [10], and the intricacy of competence levels [10, 28, 29]. The depth limit affects the performance of predictive models used to suggest issues for contributors [8, 31], thereby mitigating a core contributor shortage and supporting expert recruitment.

To close the gap of competence depth, we present a case study using source code data to evaluate contributors' competence in OSS, including levels mined from 5,793 unique functions. Our approach combines Abstract Syntax Tree (AST)-based competence extraction, hierarchical competence modeling, code complexity metrics (13 Lizard metrics), and machine learning models to build contributors' profiles and predict competence levels from pull requests. The profiles can later be used to generate recommendations for training or search for contributors to replace workforce as needed. Considering that, we are interested in answering the following research questions (RQ1): To what extent can we predict the competences to solve issues? (RQ2): How can contributor competences be automatically identified and profiled?

By answering these RQs, this work aims to contribute with (1) a novel hierarchical approach for competence profiling that aggregates function-level complexity metrics to competences based on usage patterns, (2) preliminary validation on a manually verified set of pull requests from JabRef demonstrating up to 90% accuracy in predicting competence levels for well-represented technical domains using a reduced, leakage-aware set of code complexity features, (3) empirical evidence that code metrics and structure are stronger predictors of competence requirements, and (4) a scalable, automated method for profiling competences to later use for training, planning task allocation and replacement when core maintainers disengage, therefore contributing to the competence management.

Related Work

The approach combines language-aware competence extraction with structural evidence and proficiency levels to address limitations in prior research. Unlike Dey et al. [10], who use vector representations from API changes, this approach preserves programming-language semantics and analyzes API and library usage through AST-level structure. It shifts the focus from task/issue skill signals [28, 29] to code contributions and models levels of competence using code complexity and structural metrics. Hannan et al. [16] used Large Language Models (LLMs) to identify competence and cosine similarity to match tasks and coders; however, the method did not reveal the level of competence. Additionally, unlike Liang et al.'s [21, 22] broad human-centered model, this study operationalizes competence in a measurable and automatable way from repository artifacts, aiding in scalable profiling and decision support for training and replacement when core maintainers disengage.

2 Methodology

2.1 Case Study Design

We designed a case study to explore archival data to create a competence model and build competence profiles looking for evidence of their utility in measuring OSS workforce competence [26].

Our approach follows a systematic workflow composed of four main components: (1) data mining, (2) competence and sub-competence extraction, (3) complexity analysis and level assignment, and (4) model training (Figure 1).

In the data collection section (Section 2.2), we detail the data gathering process including the justification for the data source, which was the OSS project targeted for this study and how our data was organized to find evidence of the competences based on the source code and their respective AST with the support of a taxonomy and a Large Language Model (LLM) for categorization. Next, we gathered static metrics from the source code to derive the levels using a composite score (Section 2.3). Finally, we built a prediction model (Section 2.4) to verify to what extent the proposed competence management can address the proposed motivation.

Figure 1: Research Design Overview

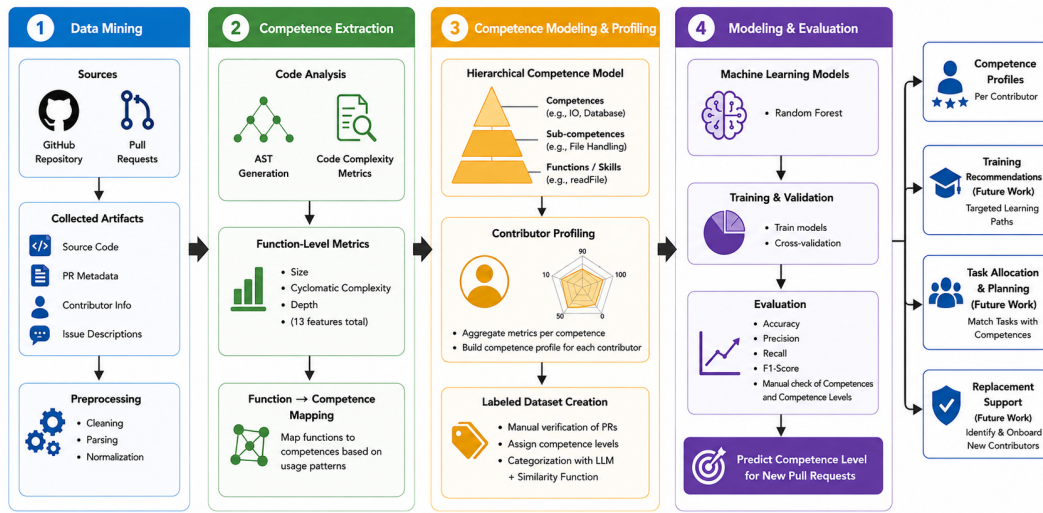


Figure 1 Overview of the proposed competence profiling workflow.

2.2 Data Collection

Data collection comprised the mining process (Section 2.2.1) and competence extraction (Section 2.2.2)

2.2.1 Data Mining

We collect data from the JabRef GitHub repository, a mature OSS Java project used as a case study in this work. We selected JabRef based on its project maturity and the breadth of technical competences represented in its codebase (e.g., database management, UI, network communication, parsing), which is important for testing a competence taxonomy spanning multiple domains. It is a popular OSS Java-based reference management tool with over 4,000 stars on GitHub, more than 900 contributors, 4,500 closed issues, and a history of research collaboration, where maintainers and researchers find solutions for OSS projects. It also has a complex codebase spanning multiple technical competences including database management, user interface design, network communication, and bibliographic parsing. The broad range of competences is important for testing our predictive model. Finally, one

author contributed to JabRef, enabling concrete evaluation of the competences. For this case study, we limited the dataset size to allow a rigorous verification of the AST results and categorization. The dataset consists of 5,793 unique functions. The functions collected encompass 85 PRs where two authors (one JabRef contributor and one software engineer) could evaluate the competence categorization.

From the repository, we extract all source files modified in merged PRs together with PR metadata, including author information, timestamps, changed files, and contribution statistics. These data are used as input for competence extraction, complexity analysis, and model training.

We filter source files by removing non-code files and focusing on substantial code contributions which contain at least one parsable source code file from which AST structures and Lizard complexity metrics can be extracted. These filtered source files feed into our competence extraction pipeline, while the PR metadata is used for both competence extraction and model training phases. We sent the source code from the 5,793 unique functions to the competence-level extraction pipeline.

2.2.2 Competence Extraction

For each selected source file, we perform the following steps:

Abstract Syntax Tree (AST) Generation. We parse each source file into its AST representation, capturing the complete structural and syntactic information of the code.

Extract APIs and Function Names. From the generated ASTs, we traverse the tree structure to identify and extract all API calls, library imports, and function/method definitions used in the code.

Competence Taxonomy. Using the extracted APIs and function names, we build a hierarchical competence taxonomy [8] which contains 217 competences organized in two layers (31 competences in the first layer and 186 sub-competences in the second layer). This taxonomy organizes programming constructs into meaningful concepts (e.g., “Database - Backup,” “IO - File Management,” and “UI - Event Handler,” where Database is the first layer, or the competence and Backup, the second layer, or the sub-competence).

Competence Classification. We classify each API and function into concepts from the taxonomy generated in the previous phase, producing classifications for each source file. This output maps each of the 5,793 unique functions declared in the source files to the competence taxonomy, capturing competence and sub-competence.

LLM-based Categorization. We used the zero-shot approach similar to SkillScope [8], but employing LLaMA-3.1-8B-Instant [13] via the Groq API to categorize APIs and functions into our competence taxonomy. We chose this approach because LLaMA-3.1-8B-Instant is an open-access model that offers a competitive cost-benefit. The LLM receives only taxonomy definitions without example classifications, using a near-zero temperature (temperature=0.1) to limit output variability while avoiding the degenerate or repetitive completions that strict determinism (temperature=0) can sometimes produce [20]. To mitigate hallucinations, we implement post-processing by ingesting the LLM output into spaCy [30] semantic similarity, to ensure outputs align with valid taxonomy entries. Complete prompt templates are available

in our replication package. Two authors evaluated a sample of 361 (from the 5,793 unique functions, using a 95% confidence level and 5% margin of error) functions to ensure correct classification.

The extracted PR metadata are combined with the competence and sub-competence categories defined in the taxonomy, using one-hot encoding, to create comprehensive training datasets that link contributors to their demonstrated competence.

2.3 Data Analysis

Using the Lizard static analysis tool [33], we extract comprehensive function-level code complexity metrics from each modified file in the pull request. Rather than relying solely on cyclomatic complexity, we employ a composite scoring approach that combines multiple complementary metrics to provide a more robust assessment of code difficulty and required competence.

Lizard computes cyclomatic complexity (CCN) by counting linearly independent paths through the code: each conditional statement (if, else if), loop (for, while, do-while), case in a switch statement, logical operator in conditions (and, or), and exception handler (catch) increments the CCN by one. Additionally, Lizard provides non-comment lines of code (NLOC), token count, parameter count, and function length metrics.

To avoid the limitations of single-metric approaches, we calculate a composite complexity score using weighted contributions from multiple metrics:

$$\text{Composite Score} = \sum_i w_i \times \frac{M_i}{T_i} \quad (1)$$

Where M_i represents each metric value, T_i is its normalization threshold, and w_i is its weight. The metrics and their weights are: **CCN Metrics (30% total)**: average CCN (10%), maximum function CCN (10%), and CCN density (10%). **Size Metrics (50% total)**: total NLOC (10%), average NLOC per function (10%), average parameters (10%), average function length (10%), total tokens (5%), and average tokens per function (5%). **Structural Metrics (20% total)**: number of functions (5%), number of files (5%), number of competences (5%), and number of sub-competences (5%).

Each metric is normalized to a 0-1 scale using empirically-derived thresholds based on JabRef data analysis. We use different normalization strategies depending on the metric distribution.

For metrics with highly skewed distributions and extreme outliers, such as NLOC and token counts, we use the 95th percentile as the normalization threshold to reduce the influence of unusually large pull requests. For example, NLOC is normalized by 100 (95th percentile observed: 52) and tokens by 200 (95th percentile observed: 435).

For metrics with more stable ranges and bounded behavior, such as CCN and function counts, we use empirically observed maximum reference values. CCN values are normalized by 30 (maximum observed: 78), while function counts are normalized by 100. This normalization strategy ensures that metrics with different scales contribute proportionally to the composite score while limiting distortion caused by extreme values.

We map composite scores to five competence levels: scores ≤ 0.25 indicate simple code (Level 1), $0.25 < \text{score} \leq 0.45$ suggests moderate complexity (Level 2), $0.45 < \text{score} \leq 0.65$ indicates complex code (Level 3), $0.65 < \text{score} \leq 0.85$ represent very complex code (Level 4), and $\text{score} > 0.85$ denotes highly complex code requiring Level 5 competence.

This composite approach addresses the limitation that cyclomatic complexity alone may not fully capture all dimensions of code difficulty. For example, a function with moderate CCN but extremely high token count (indicating dense logic) or a function with low CCN but

spanning many lines (indicating structural complexity) would both be appropriately weighted in the final assessment. The weights were tuned using a Grid Search technique aiming to increase the accuracy of the prediction model and prioritize decision-making complexity while accounting for code size and project structure. The number of complexity levels and thresholds were inspired by the work of Liang et al. [21]. Previous studies have similarly used static analysis metrics obtained from source code to investigate code violations, cognitive complexity, and the security and quality characteristics of LLM-generated code [15, 19, 27], supporting the use of such metrics as a meaningful signal for competence assessment in this study.

Similar function-level combinations of static code metrics (NLOC, cyclomatic complexity, token count, and parameter count) alongside Term Frequency-Inverse Document Frequency (TF-IDF) features have been used effectively for lightweight software engineering classification tasks, such as vulnerability triage in C/C++ codebases [9], supporting the use of this metric family as an established, non-arbitrary foundation for our composite scoring approach.

To determine whether the surrogate model’s predictive performance differs significantly from a naive baseline, rather than diverging only by chance, we apply McNemar’s test [23], a non-parametric test for paired nominal data that is well suited to comparing two classifiers evaluated on the same set of samples, as recommended by Dietterich [11] for the statistical comparison of supervised learning algorithms.

2.4 Model Training

We implemented a hierarchical complexity extraction and prediction system using the Lizard static analysis tool to measure code complexity at multiple levels of granularity. Our system extracts 13 distinct complexity metrics and aggregates them hierarchically from the function level; of these, a reduced set of 6 structural metrics (Table 1) is used as input to the surrogate competence level model.

■ **Table 1** Features for Competence Level Prediction.

Feature	Description	Competence Signal
Total NLOC	Sum of non-comment code lines	Contribution size
Total Tokens	Sum of lexical tokens	Lexical density
Num Functions	Count of modified functions	Breadth of change
Num Files	Count of modified files	Cross-file impact
Num Competences	Count of technical domains	Domain diversity
Num Sub-competences	Count of sub-domains	Fine-grained breadth

Hierarchical Aggregation. We aggregate function-level metrics following a three-tier hierarchy: **Function Level:** Individual function metrics from Lizard. **Sub-competence Level:** Functions are grouped by API usage patterns (e.g., “Database → Query Execution”, “User Interface → Layout Design”). For each sub-competence, we calculate: total CCN, total NLOC, total tokens, number of functions, average CCN, average NLOC per function, average tokens per function, and maximum function CCN. **Competence Level:** Sub-competences are aggregated into parent competences, and metrics are the sum of all sub-competence metrics within that competence.

Composite Complexity Scoring. We calculate a weighted composite score from all 13 metrics to assign competence levels. The composite score combines the metrics using equation (1) and competence levels (Section 2.2.2).

This multi-metric approach provides a more robust assessment than single-metric thresholds, as it captures multiple dimensions of code complexity.

Machine Learning Models. We adopt the one-vs-all strategy [24] to deal with the large number of categories in the multi-label problem aggravated by the data imbalance. One-vs-all strategy also enables the study to scale to more layers, potentially leading to an extreme multi-label problem in the future [32].

We train two complementary models:

Competence Classification: Predicts which technical areas are involved in a pull request using issue and linked PR text (title and description) processed through TF-IDF vectorization following previous research indicating best performance with lexical approaches instead of semantic for noisy text [16]. This multilabel model identifies which technical areas are involved.

Competence Level Prediction Model: Predicts the required competence level (1-5 scale) using both textual features and a surrogate model with six code complexity metrics [14]. The option for the surrogate model is twofold: it deals with the data leakage of predicting the levels using the same features used to obtain them, and enables the model to use only essential data available when the PR is submitted. This comparative feature-reduction approach follows standard practice in feature selection [14, 17]. This multiclass model determines *who* (at what competence level) should work on the contribution.

The models work in tandem to provide comprehensive competence profiling: the competence model identifies technical scope, while the competence level model quantifies required experience. The predictive models demonstrate that the proposed metrics capture signals aligned with the requirements for identifying competence levels. We ran the models using Random Forest (RF) which reported competitive results in Software Engineering research [8, 28, 29, 31].

3 Results

In this section, we present the results of the study.

3.1 Dataset Characteristics

We successfully extracted complexity metrics from 19,005 function-level records (a function counted once for each commit in which it was modified), corresponding to 5,793 functions uniquely identified by name and file path across the dataset. From the 85 pull requests analyzed, we discarded 21 PRs that did not declare any source code and contained documentation or configuration-only updates. In total, this yielded 64 PRs that contained substantive code modifications with at least one extracted function and measurable complexity metrics. Table 2 summarizes the dataset characteristics.

Note that the 296.9 average functions per PR (Table 2) is computed from the 19,005 total function-level records, not the 5,793 uniquely identified functions, since a function modified across multiple commits is counted once per commit.

We organize technical knowledge using a two-level hierarchy composed of competences and sub-competences. Competences represent broad technical areas (e.g., User Interface, Network, Database), while sub-competences capture more specific activities within them (e.g.,

■ **Table 2** Dataset Summary Statistics (n=64).

Metric	Value
Total PRs Analyzed	85
PRs with Code Changes	64 (75.3%)
Empty PRs (docs/config)	21 (24.7%)
Avg Functions per PR	296.9
Avg NLOC per PR	4,411
Avg Tokens per PR	32,598
Min Functions	3
Max Functions	3,196
Min Sub-competences	1
Max Sub-competences	33

Layout Design, Connection Management, Query Execution). Each pull request may involve one or multiple competences and sub-competences depending on the APIs and functions identified during AST analysis and classification.

The dataset exhibits substantial variance in complexity, ranging from simple, single sub-competence changes (e.g., PR #34 with 8 functions in one sub-competence) to large architectural modifications (e.g., PR #80 with 2,156 functions across 33 sub-competences and 15 competences). This diversity enables robust model training across different competence levels.

McNemar’s test confirmed the surrogate model’s improvement over a majority-class dummy baseline was not due to chance: across 5,793 unique functions, the surrogate was correct where the dummy was wrong 981 times, versus 369 in the reverse, yielding a significant McNemar statistic of 276.53 ($p < 0.0001$).

3.2 Complexity Metrics Distribution

Table 3 presents the distribution of extracted complexity metrics across all 64 PRs with code changes.

■ **Table 3** Complexity Metrics Distribution.

Metric	Mean	Min	Max
Cyclomatic Complexity	3.32	1.00	78.00
Lines of Code (NLOC)	4,411	24	50,671
Token Count	32,598	175	367,105
Functions per PR	296.9	3	3,196
Sub-competences per PR	7.2	1	33
Competences per PR	5.1	1	15

The mean cyclomatic complexity of 3.32 indicates that most code in JabRef is moderately complex, falling into the Level 1 - Level 2 range. However, the maximum CCN of 78.00 suggests that some contributions require Level 5 competence. The wide range in NLOC (24 to 50,671) and tokens (175 to 367,105) reflects the diversity of contribution sizes, from small, low-line-count changes to major feature additions.

3.3 Model Performance

Despite the relatively small dataset, both models achieved strong performance. Table 4 summarizes the results.

The Competence Classification model achieved 92.91% accuracy in identifying which technical competences are involved in a PR. For competence level prediction, we adopted a surrogate model using six structural features to mitigate the risk of target leakage inherent

■ **Table 4** Model Performance Metrics.

Model	Metric	Value
Competence Classification	Accuracy	92.91%
	F1-Score	75.80%
	Training Samples (functions)	5,793
Surrogate Model (6 features)	Test Accuracy	69.23%
	F1-Score (weighted)	67.18%
	CV Accuracy (3-fold)	62.41%
	Training Samples (functions)	5,793
Dummy Baseline (Competence Level, most-frequent)	Accuracy	38.46%
	F1-Score (weighted)	21.37%
	Training Samples (functions)	5,793

in predicting complexity levels from complexity-derived metrics themselves. The surrogate competence level model (69.23% accuracy, 67.18% F1) substantially outperforms the strongest naive dummy baseline (38.46% accuracy, 21.37% F1), supporting that it captures genuine predictive signal from code complexity metrics rather than exploiting class imbalance alone.

The competence level distribution across our 64 PR-level training samples was Level 1 (20.3%), Level 2 (30.8%), Level 3 (40.6%), Level 4 (6.2%), and Level 5 (2%), with contributions concentrated in the moderate-complexity range (Levels 2–3) and few reaching the highest complexity tiers. Note this reflects individual PR labels prior to aggregation; Section 3.6 reports a markedly different distribution once competence is aggregated to the contributor level using a maximum rule (Table 7).

3.4 Per-Domain Performance Analysis

To understand how the competence level prediction model performs across different technical domains, we analyzed prediction accuracy for each of the competences identified in our dataset. Table 5 presents the results, sorted by F1-Score.

■ **Table 5** Competence Level Prediction Accuracy (Surrogate Model), by Technical Domain.

Domain	Accuracy	Precision	Recall	F1-Score	Samples	Avg Level
Databases	90.00	91.67	90.00	89.45	10	2.30
Error Handling	85.71	89.29	85.71	84.35	7	2.14
Operating System	75.00	76.94	75.00	75.43	12	2.50
Computer Graphics	70.00	70.09	70.00	69.97	30	2.50
Input-Output	68.97	73.78	68.97	69.27	29	2.17
Event Handling	65.52	65.80	65.52	65.34	29	2.45
Network	63.64	69.26	63.64	64.69	11	2.55
User Interface	63.41	65.83	63.41	63.62	41	2.56
Language	57.14	65.00	57.14	60.32	14	2.71
Data Structure	56.25	56.35	56.25	56.00	32	2.41

The surrogate model achieved up to 90.00% prediction accuracy in the Databases domain, demonstrating its effectiveness in predicting contributor competence in specialized technical areas. Performance decreased in broader and more heterogeneous domains such as Language (57.14%) and Data Structure (56.25%), where the wider variety of development tasks and complexity patterns makes competence-level prediction more challenging. Overall, the results indicate that domain specialization enables more accurate prediction, whereas heterogeneous domains introduce greater classification uncertainty. Label cardinality is 7.1094 and label density is 0.0846, confirming the multilabel nature of the problem and the integrity of the learning process [4, 18].

3.5 Feature Importance Analysis

Feature importance analysis revealed that code complexity metrics substantially outperform textual features. Table 6 shows the top features.

■ **Table 6** Surrogate Model Feature Importance (6 Features).

Feature	Importance
Total Tokens	0.2725
Total NLOC	0.2324
Num Files	0.1561
Num Functions	0.1428
Num Competences	0.1058
Num Sub-competences	0.0904

Total token count emerged as the strongest predictor (importance: 0.2725), followed by total NLOC (0.2324) and number of files touched (0.1561). Structural breadth metrics (number of competences and sub-competences) contributed less, suggesting that raw contribution size and lexical density carry more signal for level prediction than the breadth of technical areas touched.

RQ1 Answer. The competence level can be predicted with up to 90% accuracy in the best-performing domain (Databases) using a reduced, leakage-aware surrogate model; the most influential predictors are total token count and total NLOC, indicating that contribution size and lexical density are stronger signals than technical breadth alone.

3.6 Contributor Profiling

Beyond analyzing individual pull requests, our approach enables the construction of comprehensive contributor profiles by aggregating competence across all contributions. For each contributor, we apply a maximum aggregation rule: the highest observed level in each sub-competence is retained in the contributor profile. This reflects the principle that when managers want to search for a contributor for a possible allocation, they can start by searching for the highest-skilled contributor, and then investigate the individual competence/sub-competence levels to reveal more details about the contributor’s competences that demonstrate capability at a certain level.

Contributor Analysis Results. From 5,793 unique functions, we identified 16 contributors with varying competence levels. The contributor-level distribution below reflects, per contributor, the maximum level reached across their individual sub-competence scores, not the PR-level distribution reported earlier. Because each contributor spans many sub-competences (avg. 20.5, Table 8), a high level in even one area is enough to register, so contributor-level Level 5 rates run higher than PR-level rates. Table 7 presents the distribution of contributors by their maximum competence level achieved.

■ **Table 7** Contributor Distribution by Competence Level.

Max Competence Level	Count	Percentage
Level 5	3	18.75%
Level 4	2	12.5%
Level 3	7	43.75%
Level 2	4	25.0%

The distribution reveals that 56.25% of contributors (9 of 16) demonstrate Level 3 or Level 4 capabilities in at least one technical area, suggesting that a substantial proportion of contributors possess intermediate to advanced technical capabilities. This expertise distribution can help project maintainers identify contributors with similar technical capabilities when planning contributor replacement or assigning new tasks.

Competence Breadth Analysis. Contributors demonstrated competence across an average of 20.5 competence/sub-competence combinations (SD = 10.8, range = 6-43). Table 8 shows the most versatile contributors.

■ **Table 8** Top Contributors by Competence Breadth.

ID	Areas	Max Lvl	PRs
d	43	5	4
c	39	5	3
o	35	5	4
b	28	4	5

Contributor “d” demonstrated the broadest competence, achieving Level 5 proficiency in four competences while maintaining capabilities across 43 distinct sub-competence areas. This breadth-and-depth combination makes such contributors particularly difficult to replace, as finding candidates with comparable versatility requires matching not just peak competence but also the range of secondary competences.

Competence Profile Example. To illustrate the granularity of our profiles, Table 9 shows the competence distribution for contributor “a”, who achieved Level 4 status across multiple competences.

■ **Table 9** Competence Profile for Contributor "a".

Level	Competence	Sub-competence	PRs
4	Computer Graphics	Image Rendering	4
4	Data Structure	Linear Structures	5
4	Event Handling	User Interactions	5
4	User Interface	Layout Design	5
3	Error Handling	Exception Handling	1
3	Input-Output	Data Reading	3
3	Network	Protocol Implementation	1
3	Operating System	Memory Management	1
...and 19 additional areas at Levels 1-2			

This profile reveals that contributor “a” achieved Level 4 mastery in four core areas, Level 3 competence in four additional areas, and maintained working proficiency across 19 other technical competences at Levels 1–2. Such detailed profiles enable precise matching: if a project needs to replace this contributor, the search can prioritize candidates with demonstrated Level 4 capabilities in at least Computer Graphics and User Interface competences, rather than simply seeking “experienced Java developers.”

RQ2 Answer. The proposed method identified contributor profiles aggregated from historical contributions revealing competences, sub-competences and levels.

4 Discussion

4.1 Implications for OSS Sustainability

Competence modeling has been explored by the industry to understand how to improve performance with qualified personnel [6]. Our study mines data from OSS repositories aiming to build profiles to benefit OSS communities by utilizing contributor's competences to optimize task allocation towards populating an ontology to anchor the concepts [5, 7] for automatic matching [12] at the next stage of this research.

Our approach has several implications for OSS project sustainability. First, by enabling automated candidate search at scale, projects can proactively build contributor pipelines before disengagement occurs. Second, the deep competence profiling allows for more accurate matching, reducing onboarding friction and increasing the likelihood of successful integration. Third, maintainers can identify competence gaps in their current team and train or recruit strategically to improve project resilience.

4.2 Limitations

External Validity: Case studies are limited in generalization on their own. Internal Validity: Metrics derived from static code analyzers may not always accurately reflect real-world code complexity; nevertheless, they are widely used tools in industry to give feedback to coders. LLM categorization cannot ensure determinism. However, we mitigated this through a low sampling temperature (0.1) to reduce output variability, combined it with a similarity metric to filter out hallucinations, and we manually analyzed 361 samples (functions) to confirm the results. Construct validity: Competence management is a broad concept, and this study aims to explore one topic: the level of competence. We employed programming language libraries, functions, and static metrics as proxies for competence, which may not reflect the true theoretical meaning of the concept, and reflect task complexity, PR size, or legacy code structure rather than the contributor's actual capabilities.

5 Future Work and Conclusion

Future work will build profiles for AI coding agents as contributors' teammates and will explore competence levels with additional metrics, at scale across multiple repositories, applying data source triangulation to assess whether the composite scoring thresholds and predictive models generalize beyond JabRef while exploring diverse LLMs, temperature and tuning parameters. New metric thresholds will be evaluated with industry practitioners using card sort approach. We plan to validate our approach through multiple methods: comparing our competence profiles against self-reported competence and levels in developer profiles, conducting surveys and interviews with project maintainers to assess the relevance of recommended candidates, and conducting experiments to evaluate maintainers' and contributors' feedback about the competences, levels and allocation recommendations. The accuracy to infer training needs, search, and recruitment will be explored in future work, along with an estimation of the costs associated with applying this approach in practice. Future work will also explore the use of a competence ontology to map the task and contributor profile to perform allocation through ontology matching. This work presents an initial approach for building competence profiles to later support search, recruitment, and training for the OSS workforce. By implementing competence-based management, we aim to contribute to the sustainability of OSS projects by leveraging complexity level and competence prediction for allocation recommendation.

Data Availability Statement

The replication package is available through Zenodo <https://doi.org/10.5281/zenodo.21605122>. It will be publicly available during the review process and after acceptance.

References

- 1 Nana Assyne, Hadi Ghanbari, and Mirja Pulkkinen. The essential competencies of software professionals: A unified competence framework. *Information and Software Technology*, 2022.
- 2 Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. On the abandonment and survival of open source projects: An empirical investigation. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12. IEEE, 2019.
- 3 Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. A novel approach for estimating truck factors. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.
- 4 Alberto Blanco, Arantza Casillas, Alicia Pérez, and Arantza Diaz de Ilarraza. Multi-label clinical document classification: Impact of label-density. *Expert Systems with Applications*, 138:112835, 2019.
- 5 Rodrigo F. Calhau and João Paulo A. Almeida. Zooming in on competences in ontology-based enterprise architecture modeling. In *Lecture Notes in Business Information Processing*, pages 198–213. Springer International Publishing, 2023.
- 6 Rodrigo F. Calhau, João Paulo A Almeida, Tiago Prince Salse, Pedro Paulo F Barcelos, and Giancarlo Guizzardi. Core-o: A competence reference ontology for professional and learning ecosystems. In *Formal Ontology in Information Systems: Proceedings of the 14th International Conference (FOIS 2024)*. IOS Press, 2024. doi:10.3233/FAIA241289.
- 7 Rodrigo F Calhau, Carlos LB Azevedo, and João Paulo A Almeida. Towards ontology-based competence modeling in enterprise architecture. In *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 71–81. IEEE, 2021.
- 8 Benjamin C Carter, Jonathan Rivas Contreras, Carlos A Llanes Villegas, Pawan Acharya, Jack Utzerath, Adonijah O Farner, Hunter Jenkins, Dylan Johnson, Jacob Penney, Igor Steinmacher, et al. Skillscope: A tool to predict fine-grained skills needed to solve issues on github. In *Proceedings of the Fourth ACM/IEEE International Workshop on NL-based Software Engineering*, 2025.
- 9 Chun Yin Chiu. Lightweight vulnerability detection from code metrics and token features, 2026. Preprint. doi:10.48550/arXiv.2605.04260.
- 10 Tapajit Dey, Andrey Karnauch, and Audris Mockus. Representation of developer expertise in open source software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 995–1007. IEEE, 2021. doi:10.1109/ICSE43902.2021.00094.
- 11 Thomas G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923, 1998. doi:10.1162/089976698300017197.
- 12 Daniel Faria, Catia Pesquita, Emanuel Santos, Matteo Palmonari, Isabel F Cruz, and Francisco M Couto. The agreementmakerlight ontology matching system. In *International Conferences On the Move to Meaningful Internet Systems*, 2013.
- 13 Grok. Grok, 2026. URL: <https://console.groq.com/docs/model/llama-3.1-8b-instant>.
- 14 Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003. URL: <https://jmlr.org/papers/v3/guyon03a.html>.
- 15 Philipp Haindl and Gerald Weinberger. Does chatgpt help novice programmers write better code? results from static code analysis. *IEEE access*, 12:114146–114156, 2024. doi:10.1109/ACCESS.2024.3445432.

- 16 Md Abdul Hannan, Mohammad Habibullah Rakib, Khondaker Masfiq Reza, and Fabio Santos. Contribution history as a key feature in OSS task recommendation: An llm-based empirical study. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2025, Honolulu, HI, USA, October 2-3, 2025*, pages 365–371. IEEE, 2025. doi:10.1109/ESEM64174.2025.00070.
- 17 Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2nd edition, 2009.
- 18 Francisco Herrera, Francisco Charte, Antonio J. Rivera, and Mara J. del Jesus. *Multilabel Classification: Problem Analysis, Metrics and Techniques*. Springer Publishing Company, Incorporated, 1st edition, 2016.
- 19 Mohammed F Kharma, Soohyeon Choi, Mohammad Alkhanafseh, and David Mohaisen. Security and quality in llm-generated code: A multi-language, multi-model analysis. *IEEE Transactions on Dependable and Secure Computing*, 2026.
- 20 Belkacem Laïmouche. Comparison of semantic reproducibility and repeatability in large and small language models: The impact of temperature. In *International Conference on Artificial Intelligence and Virtual Reality*, pages 589–602. Springer, 2025.
- 21 Jenny T Liang, Thomas Zimmermann, and Denae Ford. Towards mining oss skills from github activity. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 106–110, 2022. doi:10.1109/ICSE-NIER55298.2022.9793514.
- 22 Jenny T Liang, Thomas Zimmermann, and Denae Ford. Understanding skills for oss communities on github. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 170–182, 2022. doi:10.1145/3540250.3549082.
- 23 Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.
- 24 Bilal Mirza and Zhiping Lin. One-vs-all for class imbalance learning. In *2013 9th International Conference on Information, Communications & Signal Processing*, pages 1–5. IEEE, 2013. doi:10.1109/ICICS.2013.6782785.
- 25 Donna Rodriguez, Rita Patel, Andrea Bright, Donna Gregory, and Marilyn K Gowing. Developing competency models to promote integrated human resource practices. *Human Resource Management: Published in Cooperation with the School of Business Administration, The University of Michigan and in alliance with the Society of Human Resources Management*, 41(3):309–324, 2002.
- 26 Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009. doi:10.1007/S10664-008-9102-8.
- 27 Alfred Santa Molison, Marcia Moraes, Glaucia Melo, Fabio Santos, and Wesley KG Assuncao. Is llm-generated code more maintainable & reliable than human-written code? In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 151–162. IEEE, 2025.
- 28 Fabio Santos, Joseph Vargovich, Bianca Trinkenreich, Italo Santos, Jacob Penney, Ricardo Britto, João Felipe Pimentel, Igor Wiese, Igor Steinmacher, Anita Sarma, et al. Tag that issue: applying api-domain labels in issue tracking systems. *Empirical Software Engineering*, 28(5):116, 2023. doi:10.1007/S10664-023-10329-4.
- 29 Fabio Santos, Igor Wiese, Bianca Trinkenreich, Igor Steinmacher, Anita Sarma, and Marco A Gerosa. Can i solve it? identifying apis required to complete oss tasks. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 346–357. IEEE, 2021.
- 30 SpaCy. Industrial-strength natural language processing (nlp) in python, 2026. URL: <https://pypi.org/project/spacy/>.

- 31 Joseph Vargovich, Fabio Santos, Jacob Penney, Marco A Gerosa, and Igor Steinmacher. Givemelabeledissues: An open source issue recommendation system. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 402–406. IEEE, 2023.
- 32 Marek Wydmuch, Kalina Jasinska, Mikhail Kuznetsov, Róbert Busa-Fekete, and Krzysztof Dembczynski. A no-regret generalization of hierarchical softmax to extreme multi-label classification. *Advances in neural information processing systems*, 31, 2018.
- 33 Terry Yin. Lizard: an extensible cyclomatic complexity analyzer. *Astrophysics Source Code Library*, pages ascl–1906, 2019.
- 34 Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. On the scalability of linux kernel maintainers’ work. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 27–37, 2017. doi:10.1145/3106237.3106287.