Beyond Self-Declared Skills: Inferring Developer Specialization from Public Source Code

Enzo Augusto Valentim
Federal University of Lavras (UFLA)
Institute of Sciences, Technology and Innovation (ICTIN)
São Sebastião do Paraíso-MG, Brazil
enzo.valentim@estudante.ufla.br

Gustavo Vitor da Silva
Federal University of Lavras (UFLA)
Institute of Sciences, Technology and Innovation (ICTIN)
São Sebastião do Paraíso-MG, Brazil
gustavo.silva39@estudante.ufla.br

Johnatan Oliveira
Federal University of Lavras (UFLA)
Department of Computer Science (DCC)
Lavras-MG, Brazil
johnatan.oliveira@ufla.br

Abstract

Identifying the technical specialization of developers is relevant to software maintenance and evolution tasks, including expert recommendation, task assignment, onboarding, and the analysis of open-source communities. However, developer profiles are often based on self-declared information, which may be incomplete, outdated, or disconnected from actual development activity. This paper presents an automated approach for characterizing technical specialization from public source-code repositories. The approach combines identity resolution, Degree of Authorship-based file attribution, programming-language inference, language-specific calibration of relative technical experience, dependency extraction, and LLM-based semantic inference. The resulting profile includes four dimensions: main programming language, broad technical area, specific technical skills, and relative technical experience. We evaluated the approach with 154 GitHub developers across Java, JavaScript, Python, C, C++, and C#. The results show that dimensions closer to observable code evidence are inferred more accurately: main programming language achieved 97.01% accuracy, followed by specific technical skills with 86.77%, broad technical area with 80.00%, and relative technical experience with 75.22%. These findings indicate that public repositories can support evidence-based developer profiling for maintenance and evolution scenarios, while reinforcing that public-code activity should not be interpreted as a complete measure of professional competence or seniority.

Keywords

Mining Software Repositories, Developer Profiling, Software Maintenance, Software Evolution, Degree of Authorship

1 Introduction

Identifying the technical specialization of developers is relevant to software maintenance and evolution tasks, including expert recommendation, task assignment, onboarding, team formation, and the analysis of open-source communities [1, 13–15, 21]. In distributed development environments, understanding the languages, technologies, domains, and abilities of a developer can support coordination decisions and empirical studies on software ecosystems [18].

In practice, the characterization of developers is often based on self-declared information, such as resumes, professional profiles, GitHub descriptions, or technology lists. Although useful, these sources may be incomplete, outdated, generic, or influenced by how each developer describes previous experience. Prior studies also suggest that indicators such as years of experience, job seniority, and self-assessment do not necessarily provide a direct measure of programming ability or task-specific expertise [3, 8]. Therefore, self-declared information should be complemented with evidence extracted from actual development activity.

Public source-code repositories provide an observable source of evidence. Commit histories, maintained files, programming languages, dependencies, and contribution patterns may reveal technical activities not explicitly described in professional profiles. However, transforming repository data into interpretable profiles is challenging. Simple metrics, such as commits or changed lines, capture activity volume but not necessarily sustained technical responsibility. Similarly, dependencies found in code require semantic interpretation before mapping them to broader technical areas or specific skills.

Software repository mining has addressed parts of this problem. Authorship-based metrics, such as Degree of Authorship (DOA), estimate the relationship between developers and source-code files [2, 5]. Other approaches infer expertise from repository metrics, libraries, programming languages, or technical terms extracted from code [12, 19, 23]. Studies also map technologies to professional roles or skills using visualizations, tag-based vocabularies, or supervised classification [6, 15]. Nevertheless, these dimensions are commonly treated separately. Existing approaches often focus on authorship without semantic skill characterization, infer expertise for specific technologies without considering file responsibility, or depend on predefined vocabularies and labeled datasets.

This paper investigates to what extent public source-code evidence can be integrated into an automated process for characterizing the technical specialization of developers. We propose a repository-mining approach that combines identity resolution, DOA-based file attribution, programming-language inference, relative technical-experience classification, dependency extraction, and LLM-based semantic inference. The approach produces a multidimensional profile composed of four dimensions: main programming language, broad technical area, specific technical skills, and relative technical experience.

We evaluate the approach using public GitHub evidence from developers associated with Java, JavaScript, Python, C, C++, and C#. The calibration phase uses 300 developers, 50 per language, to define language-specific thresholds for relative technical experience. The validation phase analyzes 154 developers and compares the inferred profiles against public evidence from GitHub and other professional sources when available. The results indicate that the approach is more reliable for dimensions closer to observable code evidence: main language identification achieved 97.01% accuracy, specific technical skills 86.77%, broad technical area 80.00%, and relative

technical experience 75.22%, with a weighted Kappa of 0.4427. These findings suggest that public repositories provide useful evidence for characterizing technical specialization, but this evidence should not be interpreted as a complete measure of professional seniority.

2 Background and Related Work

Mining Software Repositories (MSR) examines how development artifacts, such as commits, files, issues, dependencies, and contribution histories, can be analyzed to understand software evolution, maintenance, collaboration, and developer behavior [7]. Public repositories, especially GitHub repositories, provide rich evidence about technical activity, but also introduce methodological risks, such as incomplete histories, noisy identities, inactive forks, and misleading activity metrics [9]. Thus, repository-based characterization requires mechanisms that go beyond raw counts of commits or changed lines.

One relevant line of research focuses on authorship and maintenance responsibility. Fritz et al. [5] proposed the Degree of Authorship (DOA), a metric that estimates the relationship between developers and source-code files by considering file creation, subsequent contributions, and changes performed by others. Avelino et al. [2] investigated repository-mining techniques for identifying software maintainers, showing that authorship-based strategies can better capture responsibility over code artifacts than simple activity metrics. In this paper, authorship is not the final target of the analysis. Instead, DOA is used as an intermediate filtering mechanism to select files with stronger evidence of technical activity before inferring languages, dependencies, skills, and relative experience.

Another line of work infers technical expertise from repository evidence. Santos et al. [17] and Oliveira et al. [16] mined repositories to identify developers with expertise in libraries and technologies. Lopes et al. [12] proposed EXTRACTPRO, a tool that generates developer profiles from source-code metrics and compares repository evidence with self-declared skills. These studies show that public development activity can support expertise identification, but they often focus on specific technologies, libraries, or metrics, rather than combining authorship, language-specific calibration, dependency extraction, semantic skill inference, and experience classification into a single multidimensional profile.

A third group of studies maps source-code evidence to technical skills, roles, or interpretable profiles. Venkataramani et al. [23] mapped technical terms from GitHub projects to Stack Overflow tags to discover expertise. Saxena and Pedanekar [19] proposed SkillMap, which annotates imports and method calls with Stack Overflow tags to visualize candidate expertise. Greene and Fischer [6] presented CVExplorer, a tool that explores open-source contributions to support candidate developer identification. Montandon et al. [15] mined GitHub evidence to classify users into technical roles, such as Backend, Frontend, Mobile, DevOps, Data Science, and Full-Stack. These approaches translate low-level code evidence into human-interpretable categories, but usually depend on external vocabularies, predefined tags, supervised classifiers, or visual exploration.

Compared with prior work, our approach differs in three aspects. First, it uses authorship as an intermediate filtering mechanism before inferring technologies and skills, reducing the influence of incidental contributions. Second, it combines attributed files and dependencies with semantic inference to produce an interpretable developer profile. Third, it integrates four dimensions that are usually studied separately: main programming language, broad technical area, specific technical skills, and relative technical experience.

We use the term *relative technical experience* rather than professional seniority because seniority also involves organizational responsibility, leadership, communication, mentoring, and contextual factors that are not directly observable in public repositories [3, 8]. Thus, the goal is not to replace human assessment, but to investigate which dimensions of technical specialization can be supported by public source-code evidence.

3 Proposed Approach

We propose an automated pipeline for characterizing the technical specialization of developers from public source-code repositories. Given a set of public GitHub profiles, the pipeline produces a multidimensional profile with four dimensions: main programming language, broad technical area, specific technical skills, and relative technical experience. Figure 1 summarizes the proposed four-stage pipeline. First, the study scope and sample are defined according to the selected programming languages and inclusion criteria. Second, repository and commit data are collected from GitHub, developer identities are resolved using profile and commit information, and string similarity is applied when exact matches are unavailable. The Degree of Authorship is then calculated for each developer file pair to identify files with stronger authorship evidence. Third, these files are analyzed to determine the main programming language, estimate developers' relative technical experience, and extract dependencies and other evidence. An LLM interprets this information to infer technical areas and specific skills. Finally, the results are integrated into a multidimensional technical profile for each developer.

3.1 Data Collection and Identity Resolution

The pipeline collects public GitHub data, including repository metadata, commit histories, modified files, authorship signatures, file paths, extensions, modification dates, and dependency evidence. Repository histories are mined with PyDriller [20], while profile and repository metadata are obtained through the GitHub API. To reduce noise, we discard binary files, generated files, dependency directories, files without implementation evidence, file-level modifications with fewer than three changed lines, and commits modifying more than 100 files (heuristic threshold).

Identity resolution is performed because commit signatures do not always match public GitHub profiles. A developer may contribute using different names, e-mail addresses, abbreviations, or alternative accounts. The pipeline compares username, name, public e-mail, and commit signatures, consolidating equivalent identities before computing authorship metrics.

3.2 DOA-Based File Attribution

The approach uses the Degree of Authorship (DOA) to avoid treating incidental contributions as evidence of technical specialization. DOA estimates the relationship between a developer and a source-code file by combining file creation, subsequent contributions, and

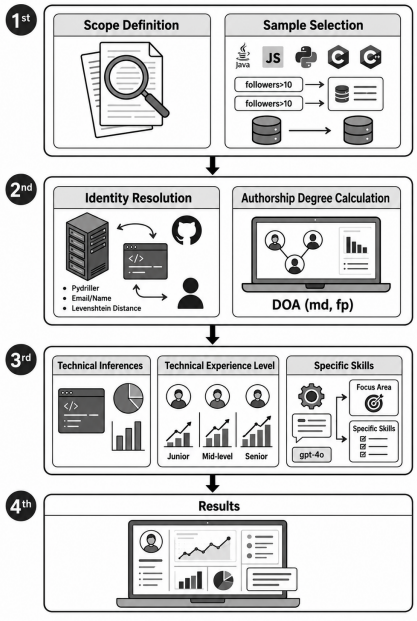

Figure 1: Overview of the proposed pipeline for public-code-based developer profiling.

changes performed by other developers [2, 5]. For each developer d and file f, DOA is computed as follows:

$$DOA(d,f) = 3.293 + 1.098 \times FA(d,f) + 0.164 \times DL(d,f) - 0.321 \times \ln(1 + AC(d,f)). \quad (1)$$

In Equation 1, $FA(d,f)$ indicates whether developer d was the first author of file f. The term $DL(d,f)$ represents the number of changes made by the developer to that file, while $AC(d,f)$ represents changes made by others. For each repository, DOA values are min–max normalized across all developer-file pairs before applying the threshold. A file is attributed to a developer only when the normalized DOA exceeds 0.75 and the absolute DOA is at least 3.293. The first criterion captures relative influence over the file, while the second prevents attribution based on minimal activity.

The resulting set of attributed files is the central evidence used by the remaining stages of the pipeline. Thus, the approach does not infer technical specialization from all files touched by a developer, but from files with stronger evidence of authorship or maintenance responsibility.

3.3 Main Language and Relative Technical Experience

The main programming language is inferred from attributed files. Each file is mapped to a programming language based on its extension and syntactic conventions. The language with the largest number of attributed files is selected as the main language. This dimension should be interpreted as the language with stronger evidence of public-code responsibility, not necessarily as the preferred or main professional language of the developer.

Relative technical experience is estimated from the number of attributed files in the main language. Since software ecosystems differ in file granularity, project structure, and coding conventions, thresholds are calibrated separately for each language. For each language, the pipeline computes quartiles from a calibration sample and classifies developers as *Junior* when $x \leq Q_1$, *Mid-level* when $Q_1 < x \leq Q_2$, and *Senior* when $x > Q_2$, where x is the number of attributed files in the main language.

These labels represent relative technical experience observed in public repositories. They do not represent professional seniority, organizational role, leadership ability, or overall competence. This distinction is important because public source-code evidence captures only part of the technical trajectory of a developer.

3.4 Dependency Extraction

The pipeline extracts dependencies from attributed files to collect semantic evidence about technologies used by each developer. Dependency extraction is language-specific: the pipeline searches for `import` statements in Java and Python, `import` and `require` statements in JavaScript, `using` directives in C#, and `#include` directives in C and C++.

The extracted dependencies are filtered to remove native libraries, generic modules, and uninformative packages. This step prioritizes third-party libraries, frameworks, and domain-specific dependencies that provide stronger evidence of technical activity. These dependencies are not final outputs; they serve as intermediate evidence for semantic inference.

3.5 Semantic Skill and Area Inference

The pipeline extracts dependencies from attributed files to collect semantic evidence about technologies used by each developer. Dependency extraction is language-specific: the pipeline searches for `import` statements in Java and Python, `import` and `require` statements in JavaScript, `using` directives in C#, and `#include` directives in C and C++. The extracted dependencies are filtered to remove native libraries, generic modules, and uninformative packages, prioritizing third-party libraries, frameworks, and domain-specific dependencies.

Recent studies report the use of LLMs in several software engineering tasks [4], and research on prompt-based learning shows that outputs may vary according to prompt design, model choice, and configuration strategies [11]. After filtering, the dependency list is submitted to an LLM through a structured prompt. The model returns two semantic dimensions: broad technical area and specific technical skills. The broad technical area is selected from the following categories used in the evaluation: *Backend*, *Frontend*, *Full-Stack*, *Mobile*, *Data Science*, *DevOps*, *Systems Programming*, *Game Development*, and *Other*. Specific skills are generated from the technologies and concepts suggested by the dependency list.

The use of an LLM is motivated by the difficulty of manually maintaining mappings between thousands of dependencies and higher-level technical categories. We treat this stage as an interpretive component, not as an authoritative source of truth. To improve consistency and reduce variability, we queried GPT-4o with temperature zero and a fixed structured prompt. The prompt constrained the output format, required the model to rely only on

the dependency evidence, and avoided free-form judgments about professional competence. For transparency and reproducibility, the replication package includes the model version, exact prompt, configuration details, query dates, and anonymized outputs.

At the end of the pipeline, each developer is represented by a profile with four dimensions: (i) main programming language; (ii) relative technical experience; (iii) broad technical area; and (iv) specific technical skills. These dimensions have different degrees of observability. Main language and relative experience are closer to repository evidence, whereas broad area and skills depend on dependency extraction and semantic interpretation.

4 Evaluation Design

The evaluation comprises three parts: dataset construction and calibration, manual validation against public evidence, and quantitative analysis using accuracy, coverage, agreement, and association metrics. The evaluation was guided by the following research questions:
RQ1: How accurately can the approach identify the main programming language associated with developer activity in public code?
RQ2: How accurately can the approach infer broad technical areas and specific technical skills from dependency evidence?
RQ3: How accurately can the approach classify developers into relative technical-experience levels based on attributed source-code files?

4.1 Dataset and Calibration

Developers were randomly sampled from public GitHub profiles identified through repository searches for Java, JavaScript, Python, C, C++, and C#. These languages were selected for their widespread use, presence in the TIOBE index [22], and diversity of programming paradigms, project structures, dependency formats, and application domains. We included profiles with source-code evidence in at least one target language and excluded those with unavailable repositories, insufficient commit history, or no files retained after DOA filtering.

The evaluation used two disjoint samples. The calibration sample comprised 300 developers, 50 per language, and defined language-specific thresholds for relative technical experience. For each developer, the number of DOA-attributed files was computed, and its quartiles were used to define the *Junior*, *Mid-level*, and *Senior* categories for each language.

The validation sample comprised 154 developers. The pipeline inferred each developer's main programming language, broad technical area, specific skills, and relative technical experience. These outputs were compared with public evidence from GitHub profiles, repositories, project descriptions, source code, dependency files, documentation, professional pages, and other available sources.

4.2 Manual Validation Protocol

Manual validation was done to determine whether each inferred dimension was supported by public evidence. Each inferred item received one of 3 labels: *Validated*, when explicit or strongly consistent public evidence supported the inference; *Invalidated*, when evidence contradicted the inference; and *No Information*, when public evidence was insufficient to confirm or reject the inference.

For main programming language, validation considered repository languages, attributed files, and recurrent use in public projects. For broad technical area and specific skills, validation considered dependencies, frameworks, project descriptions, documentation, and public profile information. For relative technical experience, validation considered the calibrated category assigned by the pipeline and public indicators of sustained development activity in the corresponding language. Ambiguous, generic, or insufficient evidence was conservatively classified as *No Information*. This conservative strategy reduces the risk of counting unsupported inferences as correct, especially when public information is sparse or generic.

The validation followed a protocol to reduce subjectivity. Uncertain cases were assigned to *No Information*, and invalidated cases were revisited before computing metrics. The protocol was designed to assess whether the approach produces plausible, evidence-supported profiles, not whether it captures each developer's complete professional background. Therefore, validation focused on public-code evidence and publicly observable information.

4.3 Metrics

We used accuracy as the main metric for main language, broad technical area, specific skills, and relative technical experience. Let V, I, and NI denote the numbers of cases labeled as *Validated*, *Invalidated*, and *No Information*, respectively. Accuracy was computed only over validable cases:

$$\text{Accuracy} = \frac{V}{V + I} \tag{2}$$

Because some inferences could not be confirmed or rejected from public evidence, we also report validable coverage:

$$\text{Coverage} = \frac{V + I}{V + I + NI} \tag{3}$$

For ordinal experience classification, we used the weighted Cohen kappa coefficient to account for partial agreement between adjacent classes, such as *Junior* and *Mid-level*. To analyze whether errors in broad technical-area inference and specific-skill inference tended to co-occur, we encoded each validable inference as a binary error indicator, where 0 represents a validated inference and 1 represents an invalidated inference. Cases labeled as *No Information* were excluded from this analysis. We then computed Spearman rank correlation over these paired indicators as an association measure between error patterns in both dimensions. Confidence intervals were computed to estimate uncertainty for the main accuracy values.

Together, these metrics allow the evaluation to distinguish dimensions that are directly observable from repository evidence, such as programming language, from dimensions that require higher-level interpretation, such as technical area, skills, and relative experience.

5 Results

This section reports the validation results for the four inferred profile dimensions and relates them to the research questions. We first present accuracy and coverage, then discuss agreement and error association patterns across dimensions. Table 1 summarizes the validation results. Accuracy was computed over verifiable cases,

classified as *Validated* or *Invalidated*; cases labeled as *No Information* were excluded because public evidence was insufficient to confirm or refute the inference. We also report coverage, which indicates the proportion of cases for which validation was possible. Overall, the approach performs better for dimensions closer to observable source-code evidence: main programming language achieved the highest accuracy, followed by specific skills, broad technical area, and relative technical experience. Confidence intervals (CIs) were computed for the accuracy values to estimate the uncertainty associated with each result.

Table 1: Overall validation results.

Dimension	Val.	Inv.	N/I	Cov.	Acc.	95% CI
Main language	130	4	20	87.01%	97.01%	[94.13, 99.89]
Broad area	108	27	19	87.66%	80.00%	[73.25, 86.74]
Specific skills	105	16	33	78.57%	86.77%	[80.74, 92.81]
Relative exp.	82	27	45	70.78%	75.22%	[67.12, 83.33]

5.1 RQ1: Main Programming Language

The approach achieved 97.01% accuracy for main language identification, with 130 validated cases, four invalidated cases, and 87.01% coverage. This was the most stable dimension in the evaluation. The result supports the use of DOA-based file attribution before language inference, since the language was inferred from files with stronger evidence of authorship or maintenance responsibility, rather than from all files touched by a developer.

The few invalidated cases reveal a relevant distinction between public-code evidence and self-declared specialization. In some cases, developers described themselves as specialists in one language, while public repositories contained stronger DOA-based evidence in another language. Thus, the inferred language should be interpreted as the main language observed in public-code responsibility, not necessarily as the preferred language or professional identity of the developer.

The per-language analysis reinforces this interpretation. Java, C, C++, and C# achieved 100% accuracy among verifiable cases, while JavaScript and Python achieved 92.59% and 91.66%, respectively. These two languages are commonly used in heterogeneous contexts and multi-language repositories, which may explain the higher number of divergences. Overall, DOA-based file attribution provided robust evidence for identifying the main programming language associated with developer activity in public code.

5.2 RQ2: Broad Technical Area and Specific Skills

The approach achieved 80.00% accuracy for broad technical area and 86.77% for specific technical skills, with coverage values of 87.66% and 78.57%, respectively. This difference suggests that the approach is more reliable for dimensions closer to dependency-level evidence. Specific skills, such as frameworks, libraries, platforms, or technical concepts suggested by imports and package usage, are directly connected to extracted dependencies. In contrast, broad technical areas require higher-level abstraction and are more sensitive to ambiguity.

This ambiguity was especially visible in versatile ecosystems. JavaScript dependencies, for example, may indicate front-end, back-end, or full-stack development depending on the project context. Python dependencies may suggest data science, automation, back-end development, or machine learning, sometimes simultaneously. In such cases, the approach may produce an inference that is coherent with the analyzed repositories, but different from the self-declared professional focus of the developer.

To investigate whether errors in broad technical-area inference were associated with errors in specific-skill inference, we computed Spearman rank correlation between the validation states of both dimensions. The coefficient was 0.6655, indicating a moderate positive association. This result is expected because both dimensions depend on the same underlying evidence: dependencies extracted from DOA-attributed files and semantically interpreted by the LLM. However, the association is not perfect, indicating that the approach may correctly infer the broad area while missing some specific skills, or identify specific skills while assigning an overly broad or partially incorrect area.

Overall, dependency extraction combined with semantic inference was effective for characterizing technical areas and skills, but the two dimensions had different levels of stability. Specific skills were more reliably inferred because they are closer to observable code evidence.

5.3 RQ3: Relative Technical Experience

Relative technical experience achieved 75.22% accuracy, the lowest among the four evaluated dimensions. It also had the lowest coverage, with 45 out of 154 developers labeled as *No Information*. This result indicates that public evidence is often insufficient to validate experience-related inferences, since professional seniority may depend on private work, organizational roles, leadership, mentoring, architectural decision-making, and non-public contributions.

The error distribution shows that most mistakes were related to underestimating developers with Senior-compatible public evidence. Among the verifiable cases, 82 developers were correctly classified: 9 as *Junior*, 6 as *Mid-level*, and 67 as *Senior*. The remaining 27 cases were misclassified. The main pattern was that 24 developers with Senior-compatible evidence were classified as *Junior* or *Mid-level*, suggesting that experienced developers may not expose enough public-code activity to reflect their professional trajectory. Only two cases showed the opposite pattern, in which developers with *Junior* or *Mid-level* reference evidence were classified as *Senior*. This reinforces that the metric works better when substantial public-code responsibility is available, but may underestimate experience that is not fully visible in public repositories.

We also computed the Cohen kappa coefficient and the weighted Cohen kappa coefficient for this ordinal classification. The values were 0.4379 and 0.4427, respectively, indicating moderate agreement according to Landis and Koch [10]. The weighted value is relevant because errors between adjacent classes are less severe than errors between *Junior* and *Senior*. Still, the moderate agreement reinforces that this dimension should be interpreted with caution.

Overall, the approach provides a useful but limited approximation of relative technical experience. It can identify developers with strong public-code responsibility, but should not be interpreted as

a direct measure of professional seniority. Among the evaluated dimensions, relative experience is the most dependent on evidence that may be absent from public repositories.

6 Discussion

The results indicate that public repositories can support evidence-based developer profiling, but the inferred dimensions have different levels of observability. Main programming language and specific technical skills achieved the best results because they are directly connected to attributed source-code files and extracted dependencies. In contrast, broad technical area and relative technical experience require higher-level interpretation and are more sensitive to ambiguity, missing information, and the gap between public activity and professional background.

A central implication is that repository-based profiling should be interpreted as evidence of *public-code responsibility*, not as a complete assessment of professional competence. This distinction is especially important for relative technical experience: limited public activity, private repositories, or unresolved identities may lead to underestimation, while extensive public activity does not necessarily imply a senior professional role. Thus, the proposed approach is better suited to maintenance and evolution tasks that require preliminary evidence of technical familiarity, such as maintainer identification, expert search, onboarding, or maintenance handoff, rather than definitive decisions about seniority. In these scenarios, the profile can reduce the search space for human decision-makers by highlighting developers whose public-code evidence is compatible with a language, technology, or maintenance context.

The profile produced by the approach should therefore be treated as a decision-support artifact. Main language and specific skills can be used with higher confidence, broad technical area as an approximation derived from dependency evidence, and relative technical experience with greater caution. The use of an LLM avoids manually maintaining large mappings between dependencies and technical categories, but may be affected by ambiguous dependencies, incomplete context, or model overgeneralization. Future work should compare LLM-based inference with curated taxonomies, Stack Overflow tags, package registries, and supervised classifiers.

7 Threats to Validity

We discuss threats to validity following the classification commonly adopted in empirical software engineering [24]. Regarding construct validity, the inferred dimensions may not fully represent the concepts they approximate. This is especially relevant for relative technical experience, which is based on attributed files and should not be equated with professional seniority. The labels *Junior*, *Mid-level*, and *Senior* are used as ordinal indicators of public-code responsibility within a language-specific distribution, not as direct measures of professional maturity, organizational role, or overall competence. Similarly, broad technical area and specific skills are inferred from dependency evidence and may not capture all technical activities performed by a developer. Thus, absence of public evidence should not be interpreted as absence of expertise.

Regarding internal validity, errors may arise from identity resolution, DOA computation, dependency extraction, LLM-based semantic inference, or manual validation. We mitigated these threats by filtering noisy files, using DOA-based attribution, calibrating thresholds per language, using reproducible LLM configurations, and assigning uncertain cases to *No Information*. Invalidated cases were also revisited before computing the metrics. However, identity aliases, incomplete commit metadata, ambiguous dependencies, incomplete public profiles, evaluator judgment, and future changes in the behavior of proprietary LLMs may still affect the inferred profiles.

Regarding external validity, the evaluation considered six programming languages and public GitHub developers, but the results may not generalize to private repositories, industrial projects, other ecosystems, or developers with little public activity. The approach is also affected by the visibility of public evidence, since substantial private or organizational contributions may be underrepresented by repository-based indicators. Regarding conclusion validity, the results may be affected by sample size, uneven availability of public evidence, and the exclusion of *No Information* cases from accuracy computation. Therefore, accuracy values should be interpreted together with coverage values. This is particularly important for relative technical experience, which had the largest proportion of cases without enough public evidence for validation.

8 Conclusion

This paper presented an automated approach for characterizing the technical specialization of developers from public source-code repositories. The approach combines identity resolution, DOA-based file attribution, programming-language inference, language-specific calibration of relative technical experience, dependency extraction, and LLM-based semantic inference. The resulting profile includes four dimensions: main programming language, broad technical area, specific technical skills, and technical experience.

The evaluation with 154 GitHub developers across six programming languages showed that the approach is more accurate for dimensions closer to observable code evidence. Main programming language achieved 97.01% accuracy, followed by specific technical skills with 86.77%, broad technical area with 80.00%, and relative technical experience with 75.22%. These results indicate that public repositories can support evidence-based developer profiling in maintenance and evolution scenarios, especially when preliminary evidence of technical familiarity is needed. However, public-code evidence should be interpreted as partial and observable evidence, not as a complete measure of professional competence or seniority.

Future work includes expanding the evaluation to other languages and ecosystems, comparing LLM-based semantic inference with curated taxonomies and supervised classifiers, improving identity resolution, and evaluating the approach in concrete maintenance tasks, such as expert recommendation, bug triaging, and onboarding support. To support reproducibility while preserving the double-blind review process.

Artifact Availability

Replication materials are available on[1].

Acknowledgments

This research is supported by FAPEMIG (APQ-00763-24).

[1] https://github.com/sendVeem/EnvioVeem.git

References

[1] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who Should Fix This Bug?. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, 361–370. doi:10.1145/1134285.1134336

[2] G. Avelino, L. Passos, F. Petrillo, and M. T. Valente. 2018. Who can maintain this code?: Assessing the effectiveness of Repository-Mining Techniques for identifying software maintainers. *IEEE Software* 36, 6 (2018), 34–42. doi:10.1109/MS.2018.185140155

[3] S. Baltes and S. Diehl. 2018. Towards a Theory of Software Development Expertise. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*. ACM, New York. doi:10.1145/3236024.3236061

[4] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[5] Thomas Fritz, Gail C. Murphy, Emerson Murphy-Hill, Jingwen Ou, and Emily Hill. 2014. Degree-of-knowledge: modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 2 (2014), 14:1–14:42.

[6] G. J. Greene and B. Fischer. 2016. CVExplorer: identifying candidate developers by mining and exploring their open source contributions. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. ACM, New York, 804–809. doi:10.1145/2970276.2970285

[7] Ahmed E. Hassan. 2008. The Road Ahead for Mining Software Repositories. In *Proceedings of the Frontiers of Software Maintenance*. IEEE, 48–57. doi:10.1109/FOSM.2008.4659248

[8] Magne Jørgensen, Gunnar Rye Bergersen, and Knut Liestøl. 2021. Relations Between Effort Estimates, Skill Indicators, and Measured Programming Skill. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2892–2906. doi:10.1109/TSE.2020.2973638

[9] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, 92–101. doi:10.1145/2597073.2597074

[10] J. R. Landis and G. G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 1 (1977), 159–174.

[11] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *Comput. Surveys* 55, 9, Article 195 (2023). doi:10.1145/3560815

[12] Jefferson G. M. Lopes, Johnatan Alves, and Eduardo Figueiredo. 2022. EXTRACT-PRO: A Data Mining Tool for Developer Profile Generation Based on Source Code Analysis. In *Anais do XXXVI Simpósio Brasileiro de Engenharia de Software (SBES)*. Sociedade Brasileira de Computação, Porto Alegre, 112–117.

[13] Shawn Minto and Gail C. Murphy. 2007. Recommending Emergent Teams. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR '07)*. IEEE Computer Society, 5.

[14] Audris Mockus and James D. Herbsleb. 2002. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, 503–512. doi:10.1145/581339.581401

[15] João Eduardo Montandon, Marco Tulio Valente, and Luciana L. Silva. 2021. Mining the Technical Roles of GitHub Users. *Information and Software Technology* 131 (2021), 106485. doi:10.1016/j.infsof.2020.106485

[16] J. A. Oliveira, M. Viggiato, D. Pinheiro, and E. Figueiredo. 2021. Mining Experts from Source Code Analysis: An Empirical Evaluation. *Journal of Software Engineering Research and Development* 9, 1 (2021), 1:1–1:16. doi:10.5753/jserd.2021.548

[17] Adriano Santos, Maurício Souza, Johnatan Oliveira, and Eduardo Figueiredo. 2018. Mining Software Repositories to Identify Library Experts. In *Anais do XII Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*. Sociedade Brasileira de Computação, Porto Alegre, 83–91.

[18] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. 2009. Tesseract: Interactive Visual Exploration of Socio-Technical Relationships in Software Development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, 23–33. doi:10.1109/ICSE.2009.5070513

[19] R. Saxena and N. Pedanekar. 2017. I know what you coded last summer: Mining candidate expertise from GitHub repositories. In *Companion of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. ACM, New York, 299–302.

[20] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python Framework for Mining Software Repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 908–911. doi:10.1145/3236024.3264598

[21] Igor Steinmacher, Tayana Conte, Christoph Treude, and Marco Aurélio Gerosa. 2015. A Systematic Literature Review on the Barriers Faced by Newcomers to Open Source Software Projects. *Information and Software Technology* 59 (2015), 67–85. doi:10.1016/j.infsof.2014.11.001

[22] TIOBE Software BV. 2026. TIOBE Index for June 2026. https://www.tiobe.com/tiobe-index/. Accessed: 2026-06-30.

[23] R. Venkataramani, A. Gupta, A. Asadullah, B. Muddu, and V. Bhat. 2013. Discovery of Technical Expertise from Open Source Code Repositories. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, New York, 97–98. doi:10.1145/2487788.2487832

[24] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer. doi:10.1007/978-3-642-29044-2