

Assessing Code Quality and Software Architecture Using Static Analysis and Machine Learning

Péter János Császár^{1,*}

¹Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary

Abstract

Static analysis can support automated feedback on code quality and software architecture in software engineering education, but its practical value depends on whether warnings are relevant and actionable and whether metric thresholds are valid for small, milestone-based student projects. This doctoral research investigates empirically validated static-analysis methods for this setting. Student projects are the primary empirical context, while mature open-source systems provide a comparison baseline for context sensitivity and generalization. A Roslyn-based analyzer has been applied to 41 student projects across 123 milestone versions and to 22 releases of five open-source .NET systems. Initial results show substantial differences between empirically calibrated and literature-based thresholds and between warning evolution in the two contexts. Ongoing work validates warnings against independent human judgment, while instructor feedback separately evaluates their relevance and actionability, and tests whether contextual calibration improves agreement with human assessment. The same validation framework is then extended to architectural patterns and violations, followed by machine-learning models for transferring validated detection knowledge to ecosystems without dedicated analyzers. The expected contribution is an empirically validated methodology for actionable static-analysis feedback in software engineering education, together with evidence about the extent to which the underlying techniques generalize beyond this context.

Keywords

static analysis, software metrics, code quality, software architecture, machine learning, software engineering education

1. Introduction

Static analysis examines software source code without executing it and is commonly used to detect coding-rule, security, and maintainability issues or compute structural metrics. In software engineering education, such analysis can support formative feedback, but only when its findings are understandable, relevant, and actionable for instructors and students.

Software engineering education is the primary application and validation context of this dissertation. Student projects are not treated as proxies for professional software development; they form a distinct context characterized by limited experience, short development cycles, predefined milestones, and direct opportunities for feedback. Open-source systems play a complementary role as a mature comparison baseline for examining how metric distributions, warning thresholds, and warning evolution differ and which findings may generalize beyond the educational setting.

The research follows a staged progression. First, metrics including cyclomatic complexity, Bumpy Road [1], function parameter count, cohesion, coupling, and the Maintainability Index are evaluated against independent expert judgment, while instructor feedback separately assesses educational relevance and actionability. Context-specific thresholds are compared with literature-based and benchmark-derived alternatives. Second, the same validation principle is extended from method- and class-level findings to architectural pattern conformance and violations. Finally, validated analyzer results and human labels provide the basis for investigating whether machine-learning models can transfer detection knowledge to projects and programming languages for which no dedicated analyzer is available.

IDOESE'26: International Doctoral Symposium on Empirical Software Engineering, October 07, 2026, München, Germany

*Corresponding author.

✉ csaszarpeter@inf.elte.hu (P.J. Császár)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

2. Research Problem and Questions

Empirical methods for deriving software-metric thresholds from benchmark systems are well established, but their contextual validity for small, short-lived, milestone-based student projects remains unclear. Thresholds derived from mature systems may not transfer directly to an educational context with different developer experience, project duration, and feedback goals. Threshold calibration is therefore treated as a contextual replication and validation problem rather than only parameter tuning.

A second issue is practical validity. A statistically unusual metric value does not necessarily correspond to a maintainability problem that a reviewer would consider relevant or worth acting upon: high complexity, low cohesion, or high coupling may sometimes follow from legitimate domain or framework constraints. This motivates validation against independent human judgment and, separately in the educational context, instructor assessments of relevance and actionability.

A third issue concerns abstraction level. Many maintainability problems arise in dependencies, architectural layering, responsibility allocation, or incomplete design-pattern implementations rather than in individual methods or classes. Existing tools provide parts of this functionality, but their capabilities are distributed across abstraction levels, thresholding strategies, validation approaches, and language ecosystems. Across metric- and architecture-level analysis, the research-practice gap therefore concerns not the lack of static-analysis tooling itself, but the limited integration of context-specific calibration, human validation, educational actionability, and architecture-level analysis within a unified empirically evaluated workflow for software engineering education. Machine learning is considered only as a later mechanism for transferring validated detection knowledge where dedicated language- or framework-specific analyzers are unavailable.

The overall objective is therefore to investigate how empirically validated static-analysis methods can support actionable code-quality and architectural assessment in software engineering education. The research also examines which findings are specific to short-lived student projects and which remain meaningful in more mature software systems or can be transferred to other programming ecosystems.

The dissertation is organized around the following research questions:

RQ1: To what extent do warnings produced by selected software metrics correspond to code regions that independent human reviewers consider maintainability problems?

This establishes the technical validity of the metric-based warnings against independently assigned human labels.

RQ2: Does context-specific threshold calibration improve agreement with human judgment compared with literature-based or benchmark-derived thresholds?

This tests whether calibration for the educational project context improves agreement with the same human reference labels rather than merely changing the number of reported warnings.

RQ3: To what extent do static-analysis warnings identify issues that instructors consider relevant and actionable during student project assessment?

This addresses educational usefulness separately from technical accuracy by examining whether a warning would justify closer inspection, feedback, or a recommendation for refactoring.

RQ4: Are persistent static-analysis warnings associated with subsequent change, defect-fixing, or refactoring activity?

This introduces a longitudinal perspective and tests whether warning persistence has observable consequences in later repository activity.

RQ5: To what extent can architectural pattern conformance and violations be identified through static analysis, and do the detected violations correspond to problems identified by expert reviewers?

This extends the same validation principle from methods and classes to higher-level structural conformance and architectural violations.

RQ6: To what extent can machine-learning models trained on validated static-analysis labels transfer code-quality and architectural problem detection to projects or programming languages for which no dedicated analyzer is available?

This evaluates whether validated detection knowledge can be transferred beyond the language- and framework-specific rules on which the earlier phases rely.

3. Background and Related Work

Software metrics quantify properties such as complexity, cohesion, coupling, and dependency structure. The initial metric set builds on cyclomatic complexity [2], object-oriented metric suites [3], coupling measures [4], instability, and composite measures such as the Maintainability Index, whose formulations and interpretation vary across tools [5].

Empirical threshold derivation provides an alternative to fixed metric cutoffs and has been studied on established software corpora [6, 7, 8, 9]. Its applicability to short-lived educational projects has received less attention, motivating contextual replication in a C#/.NET setting. Distributional validity alone is also insufficient: false positives, low-value warnings, and poor prioritization can reduce the usefulness and adoption of static analysis [10, 11, 12, 13]. Accordingly, this work evaluates thresholds against human judgment and separately examines whether instructors consider warnings actionable.

Code quality in programming education has been studied from both assessment and automated-analysis perspectives. Prior work has characterized recurring code-quality issues in student programs and mapped how code quality is addressed in educational research [14, 15]. Assessment-oriented research has also proposed empirically validated models for judging code quality in student programs [16]. Static analysis can complement such assessment by automatically directing attention to suspicious code fragments [17], although the resulting warnings still need to be relevant and interpretable for instructors. At a higher abstraction level, architecture-aware static analysis has also been applied to C# student projects [18]; however, architectural and design patterns may be partial, framework-dependent, or violated rather than simply present or absent.

Existing tools such as SonarQube, NDepend, Designite, Arcan, Understand, and CodeScene¹ cover different subsets of code metrics, dependency analysis, architectural smells, and conformance checking. Their capabilities are nevertheless distributed across different abstraction levels, thresholding strategies, validation approaches, and language ecosystems. This motivates a structured comparison that distinguishes methodological research gaps from tool-specific implementation gaps.

Machine-learning methods for source-code analysis use representations such as tokens, syntax trees, and graphs [19]. Their usefulness for the present problem depends not only on predictive performance but also on the quality of the labels on which they are trained and on generalization beyond the projects from which those labels were obtained. In this dissertation, learning-based methods are therefore treated not as a replacement for rule-based analysis, but as a later transfer mechanism built on analyzer findings that have already been validated by human review. Previous work established the initial metric-analysis baseline on student and open-source projects [20]; the present work strengthens its empirical validation and extends the same framework toward architecture-level assessment and learning-based transfer.

4. Current Work and Research Methodology

The current Roslyn-based pipeline analyzes C# projects in a reproducible, containerized environment and produces JSON reports for aggregation and HTML reports for manual inspection. Implemented metrics include Bumpy Road, function parameter count, LCOM4 and LCOM5, Maintainability Index, cyclomatic complexity, and class coupling; planned module-level measures include afferent/efferent coupling and instability. The current set is an initial, non-exhaustive selection. CI integration is intended to run the analysis at milestones or during development, supporting timely formative feedback rather than only post-hoc assessment. The metric set deliberately spans several levels of abstraction: method-level measures capture local complexity and parameterization, class-level measures capture cohesion and coupling, and planned module-level measures describe dependency structure. This layered selection is intended to support both localized feedback and later comparison with architecture-level findings rather than to form an exhaustive catalogue of software metrics.

The primary dataset consists of team projects from a BSc Software Technology course developed

¹Official tool websites: <https://www.sonarsource.com/products/sonarqube/>; <https://www.ndepend.com/>; <https://www.designite-tools.com/>; <https://www.arcan.tech/>; <https://scitools.com/>; <https://codescene.com/>.

through predefined milestones. These repeated snapshots make it possible to observe how code-quality problems emerge during development and, at the same time, provide direct opportunities for formative instructor feedback. The analyzer is intended to direct instructor attention to code that may warrant inspection or feedback, not to replace instructor judgment.

A second dataset comprises open-source .NET systems with longer development histories. These systems provide a mature comparison baseline for metric distributions, thresholds, and warning evolution and make longer-term repository analysis possible. Projects are selected to cover different application domains, architectural styles, and sizes. Projects that cannot be restored, built, or analyzed are documented, since buildability is itself a practical limitation of static-analysis workflows.

The two datasets therefore serve different evaluation purposes. Student projects support both independent expert validation and a separate assessment of whether instructors find the resulting warnings relevant and actionable. Instructor feedback is not treated as the technical ground truth for metric accuracy and is not applicable to open-source projects; the latter instead rely on independent expert review and repository-based evidence.

For RQ1 and RQ2, warnings are selected through stratified sampling so that metrics, project types, and severity ranges are represented. At least two reviewers independently label sampled code using predefined criteria while blinded to the threshold strategy. The criteria focus on whether the highlighted code contains a maintainability problem that would reasonably justify modification, rather than on the metric value alone; borderline cases can therefore be recorded separately before adjudication. Inter-rater agreement is quantified using Cohen’s kappa and disagreements are resolved before final classification. Precision is estimated from sampled warnings; recall additionally requires independently sampled code regions not reported by the analyzer so that false negatives can be identified. This second sample is necessary because validating only reported warnings can estimate precision but cannot reveal relevant problems that the analyzer missed. These expert labels form the reference data for RQ1 and RQ2.

Literature-based, benchmark-derived, and context-specific thresholds are evaluated on projects separated from calibration data, using held-out projects or project-level cross-validation. For RQ2, classifications from the threshold strategies are compared on the same labeled regions using pairwise McNemar tests [21]; precision, recall, and F1-score quantify the practical differences. RQ3 is evaluated separately through a structured questionnaire in which instructors rate warning relevance, actionability, and whether a finding would justify closer inspection, feedback, or refactoring.

For RQ4, warning instances are tracked across milestones and releases. Persistent warnings are linked, where repository history permits, to subsequent code changes, defect-fixing, and refactoring activity. Changes touching the affected method, class, or surrounding code will be traced through version history; commits will be classified using repository metadata, issue links, commit messages, and manual inspection where necessary. The analysis will then compare persistent and resolved warnings with later maintenance activity rather than treating warning-count trends alone as evidence of impact. Open-source histories provide the primary longitudinal evidence, complemented by the shorter student repositories.

4.1. Results to Date

The initial analyzer and results are reported in an accepted ICAI 2026 paper [20]. The Roslyn-based implementation covers Bumpy Road, function parameter count, LCOM4 and LCOM5, the Maintainability Index, cyclomatic complexity, and class coupling, and is released under the BSD-3 license². Using the containerized pipeline, the tool was applied to 41 of 61 collected student projects across three milestones, yielding up to 123 project versions, and to 22 tagged releases of five open-source .NET systems: OpenRA, Jellyfin, nopCommerce, xUnit, and IronPython 3. These datasets establish the empirical baseline on which the ongoing human validation and longitudinal analyses build. The comparison is not intended to treat open-source projects as a gold standard for student code; instead, it provides a contrasting development context in which warning distributions and evolution can be observed under longer project

²<https://github.com/petercsaszar/code-metrics/>

lifetimes and different maintenance practices.

Thresholds for the student dataset were calibrated from cumulative warning counts using the elbow method and Kneede [22]. The resulting cutoffs differ considerably from literature-recommended values, providing preliminary evidence that thresholds established in other software contexts should not be assumed to transfer directly to milestone-based student projects. This result motivates, but does not yet answer, RQ2: the context-specific thresholds still have to be compared with literature- and benchmark-based alternatives against the same human reference labels. No robust threshold was identified for LCOM5 or the Maintainability Index. LCOM5 showed limited discriminative power in the analyzed data, while Maintainability Index values are sensitive to the formulation used; both therefore appear more suitable for comparative trend analysis than threshold-based warning generation in the current setting.

The longitudinal comparison also revealed different warning trajectories. Student projects generally accumulate normalized warning counts across milestones, with limited evidence of systematic reduction through refactoring, whereas the open-source releases are largely stable or slightly improving. Cohesion- and coupling-related measures, particularly LCOM4 and class coupling, distinguish the two groups most clearly. Together with the threshold differences, these results suggest that warning behavior is strongly context dependent and therefore justify the planned human validation rather than a claim that one threshold strategy is universally preferable. The findings provide a descriptive baseline for RQ4, but they do not yet establish that a persistent warning predicts subsequent change, defect-fixing, or refactoring; that association is evaluated separately from the warning-count trends.

Table 1 summarizes the empirical plan and status.

Table 1

Research questions, empirical methods, datasets, and status.

RQ	Empirical method	Dataset	Status
RQ1	Independent expert validation; Cohen’s kappa; per-metric precision and recall	Stratified student and open-source code samples	In progress
RQ2	Comparison of literature-based, benchmark-derived, and context-specific thresholds against expert labels; McNemar’s test	Validated student-project samples	In progress; calibration results available [20]
RQ3	Structured instructor questionnaire on warning relevance and actionability	Warnings from student projects	In progress
RQ4	Longitudinal analysis linking warning persistence to later change, defect-fixing, and refactoring activity	Student and open-source repository histories	In progress
RQ5	Architecture-conformance and violation detection validated against independent expert review	Student and open-source projects	Planned (2026/27)
RQ6	Cross-project and cross-language transfer of models trained on validated labels	Validated C# projects and unseen target ecosystems without dedicated analyzers	Planned (2027/28)

5. Planned Research

The initial metric set will be refined through a targeted review of empirical software-metrics literature. Additional method-, class-, and module-level measures will be selected according to evidence of relevance to maintainability, architectural quality, or actionable feedback and integrated into the Roslyn framework; selected implementations may be developed with supervised BSc thesis students. A structured comparison of mainstream analysis tools will also examine language support, metric and architecture coverage, threshold configuration, validation evidence, CI integration, and educational applicability. The comparison will be used to separate capabilities that are already available in existing

tools from methodological questions that still require empirical investigation, and to position the proposed workflow relative to current practice.

The next phase extends the validated workflow to architecture-level structures, including Model–View, MVC, MVVM, and selected GoF patterns. Static rules will use dependencies, namespace organization, inheritance, calls, and responsibility boundaries to detect conformance, incomplete implementations, and violations. For example, in an MVC-like system, dependencies from the view directly to persistence components may indicate a violation of the intended responsibility structure. Because real implementations vary across frameworks and projects, the rules must be precise enough to detect meaningful violations without assuming one canonical implementation. Expert reviewers will assess the expected pattern roles, dependencies, and responsibility boundaries independently of the analyzer output; the comparison will distinguish correct conformance findings from partial, ambiguous, or incorrectly reported violations. Detected findings will therefore be evaluated in both student and open-source projects using the same principle of independent human validation established for the metric-level analysis.

The final phase investigates machine learning as a transfer mechanism. The motivation is that implementing and maintaining language- and framework-specific rules for every target ecosystem is costly and may not always be practical. This phase therefore depends on the earlier validation work: transferring noisy analyzer output would merely reproduce the biases of the original rules, whereas expert-reviewed labels provide a stronger basis for testing whether the learned representation captures a more general structural problem. Models will therefore be trained on analyzer findings validated through human review rather than raw warnings. Token sequences, syntax trees, dependency graphs, and control/data-flow representations will be considered. Evaluation will test generalization across projects and languages; in target ecosystems without dedicated analyzers, independently reviewed samples will provide the reference labels.

6. Threats to Validity

Construct validity is limited because software metrics approximate structural properties associated with maintainability rather than maintainability itself. A high metric value is therefore not treated as a problem solely because it crosses a threshold. Independent expert labels provide the reference for RQ1 and RQ2, while instructor ratings are analyzed separately as evidence of educational relevance and actionability.

Internal validity may be affected by framework conventions, generated code, or project requirements that influence metric values independently of maintainability. Another risk is optimistic evaluation if context-specific thresholds are assessed on the same projects from which they were calibrated. Calibration and validation are therefore separated at project level, and exceptional framework- or generation-dependent cases are documented rather than silently discarded.

External validity is limited by student projects from one course and by the selected .NET open-source systems. The two datasets intentionally represent different development contexts, but neither is assumed to represent all educational or professional software. Later cross-project and cross-language evaluation is intended to test which findings transfer beyond these settings.

Manual labeling introduces reliability concerns. Reviewers use predefined criteria, label samples independently, and inter-rater agreement is reported using Cohen's kappa. Because the author developed the analyzer and teaches the course, the author will not establish the RQ1/RQ2 reference labels, and RQ3 questionnaires will be completed by other instructors. Finally, projects that cannot be restored or built because of missing dependencies or unsupported frameworks are documented as tool-related limitations rather than omitted without explanation.

7. Conclusion

This dissertation investigates empirically validated static analysis for actionable code-quality and architecture feedback in software engineering education, using open-source systems as a comparison baseline. Metric validity, threshold calibration, instructor actionability, and longitudinal evidence form the foundation for architecture-level analysis and later learning-based transfer. The expected contribution is an empirically evaluated methodology and evidence about the extent to which its findings generalize beyond the educational context.

Acknowledgments

The author would like to thank his supervisor, Dr. Máté Cserép, for his continuous guidance, valuable feedback, and support throughout this doctoral research.

Declaration on Generative AI

During the preparation of this work, the author used ChatGPT in order to support language editing, translation, text restructuring, and the drafting of preliminary versions of selected sections. No generative AI tool was used to produce empirical results, perform the software analysis, collect data, or draw scientific conclusions.

After using this tool, the author reviewed, edited, and validated the content as needed and takes full responsibility for the publication's content.

References

- [1] A. Tornhill, The bumpy road code smell: Measuring code complexity by its shape and distribution, CodeScene Engineering Blog, 2021. URL: <https://codescene.com/blog/bumpy-road-code-complexity-in-context/>, accessed: 2026-07-03.
- [2] T. J. McCabe, A complexity measure, in: Proceedings of the 2nd International Conference on Software Engineering, 4, IEEE Computer Society, 1976, pp. 308–320.
- [3] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (1994) 476–493. doi:10.1109/32.295895.
- [4] E. Arisholm, L. Briand, A. Føyen, Dynamic coupling measurement for object-oriented software, Software Engineering, IEEE Transactions on 30 (2004) 491 – 506. doi:10.1109/TSE.2004.41.
- [5] T. Heričko, B. Šumak, Exploring maintainability index variants for software maintainability measurement in object-oriented systems, Applied Sciences 13 (2023). URL: <https://www.mdpi.com/2076-3417/13/5/2972>. doi:10.3390/app13052972.
- [6] T. L. Alves, C. Ypma, J. Visser, Deriving metric thresholds from benchmark data, in: 2010 IEEE International Conference on Software Maintenance, 2010, pp. 1–10. doi:10.1109/ICSM.2010.5609747.
- [7] P. Oliveira, M. T. Valente, F. P. Lima, Extracting relative thresholds for source code metrics, in: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 254–263. doi:10.1109/CSMR-WCRE.2014.6747177.
- [8] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, H. C. Almeida, Identifying thresholds for object-oriented software metrics, Journal of Systems and Software 85 (2012) 244–257. URL: <https://www.sciencedirect.com/science/article/pii/S0164121211001385>. doi:<https://doi.org/10.1016/j.jss.2011.05.044>, special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering.
- [9] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, 2006. doi:10.1007/3-540-39538-5.

- [10] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, 2013, p. 672–681.
- [11] M. Christakis, C. Bird, What developers want and need from program analysis: an empirical study, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 332–343. URL: <https://doi.org/10.1145/2970276.2970347>. doi:10.1145/2970276.2970347.
- [12] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, C. Jaspan, Lessons from building static analysis tools at google, Commun. ACM 61 (2018) 58–66. URL: <https://doi.org/10.1145/3188720>. doi:10.1145/3188720.
- [13] S. Heckman, L. Williams, A systematic literature review of actionable alert identification techniques for automated static code analysis, Information and Software Technology 53 (2011) 363–387. URL: <https://www.sciencedirect.com/science/article/pii/S0950584910002235>. doi:<https://doi.org/10.1016/j.infsof.2010.12.007>, special section: Software Engineering track of the 24th Annual Symposium on Applied Computing.
- [14] H. Keuning, B. Heeren, J. Jeuring, Code quality issues in student programs, in: Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17), Association for Computing Machinery, Bologna, Italy, 2017, pp. 110–115. doi:10.1145/3059009.3059061.
- [15] H. Keuning, J. Jeuring, B. Heeren, A systematic mapping study of code quality in education, in: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023), Association for Computing Machinery, Turku, Finland, 2023, pp. 5–11. doi:10.1145/3587102.3588777.
- [16] M. Stegeman, E. Barendsen, S. Smetsers, Towards an empirically validated model for assessment of code quality, in: Proceedings of the 14th Koli Calling International Conference on Computing Education Research, Association for Computing Machinery, 2014, pp. 99–108. doi:10.1145/2674683.2674702.
- [17] P. Kaszab, M. Cserép, Detecting programming flaws in student submissions with static source code analysis, Studia Universitatis Babeş-Bolyai Informatica 68 (2023) 37–54. doi:10.24193/subbi.2023.1.03.
- [18] B. Orosz, J. Szűcs, M. Cserép, Architecture-aware static analysis and violation detection of C# student submissions, Computers 15 (2026) 404. doi:10.3390/computers15070404.
- [19] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, F. Sarro, A survey on machine learning techniques applied to source code, Journal of Systems and Software 209 (2024) 111934. URL: <https://www.sciencedirect.com/science/article/pii/S0164121223003291>. doi:<https://doi.org/10.1016/j.jss.2023.111934>.
- [20] P. J. Császár, M. Cserép, Assessing software metrics and complexity in student and open-source projects using static analysis, in: Proceedings of the 13th International Conference on Applied Informatics (ICAI 2026), 2026. Accepted for publication.
- [21] Q. McNemar, Note on the sampling error of the difference between correlated proportions or percentages, Psychometrika 12 (1947) 153–157. doi:10.1007/BF02295996.
- [22] V. Satopaa, J. Albrecht, D. Irwin, B. Raghavan, Finding a "kneedle" in a haystack: Detecting knee points in system behavior, in: 2011 31st International Conference on Distributed Computing Systems Workshops, 2011, pp. 166–171. doi:10.1109/ICDCSW.2011.20.