SOFTENG 2026

The Twelfth International Conference on Advances and Trends in Software Engineering

ISBN: 978-1-68558-398-9

May 24 - 28, 2026

Venice, Italy

SOFTENG 2026 Editors

Tsuyoshi Nakajima, Shibaura Institute of Technology, Japan

SOFTENG 2026

Forward

The Twelfth International Conference on Advances and Trends in Software Engineering (SOFTENG 2026), held between May 24-28, 2026 in Venice, Italy, continued a series of events focusing on the challenging aspects for software development and deployment, across the whole life-cycle.

Software engineering exhibits challenging dimensions in the light of new applications, devices and services. Mobility, user-centric development, smart-devices, e-services, ambient environments, e-health and wearable/implantable devices pose specific challenges for specifying software requirements and developing reliable and safe software. Specific software interfaces, agile organization and software dependability require particular approaches for software security, maintainability, and sustainability.

We welcomed academic, research and industry contributions. The conference had the following tracks:

- Challenges for dedicated software, platforms, and tools
- Software testing and validation
- Software requirements
- Maintenance and life-cycle management

We take here the opportunity to warmly thank all the members of the SOFTENG 2026 technical program committee, as well as all the reviewers. The creation of such a high quality conference program would not have been possible without their involvement. We also kindly thank all the authors who dedicated much of their time and effort to contribute to SOFTENG 2026. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

We also thank the members of the SOFTENG 2026 organizing committee for their help in handling the logistics and for their work that made this professional meeting a success.

We hope that SOFTENG 2026 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in the field of software engineering. We also hope that Venice provided a pleasant environment during the conference and everyone saved some time to enjoy the historic charm of the city.

SOFTENG 2026 Chairs

SOFTENG Steering Committee
Zeeshan Ali Rana, NUCES, Lahore, Pakistan
Tsuyoshi Nakajima(中島毅), Shibaura Institute of Technology, Japan
Simona Vasilache, University of Tsukuba, Japan

SOFTENG Publicity Chairs
José Miguel Jiménez, Universitat Politecnica de Valencia, Spain
Francisco Javier Díaz Blasco, Universitat Politècnica de València, Spain
Ali Ahmad, Universitat Politècnica de València, Spain
Laura Garcia, Universidad Politécnica de Cartagena, Spain
Sandra Viciano Tudela, Universitat Politecnica de Valencia, Spain

SOFTENG 2026

Committee

SOFTENG Steering Committee

Zeeshan Ali Rana, NUCES, Lahore, Pakistan
Tsuyoshi Nakajima(中島毅), Shibaura Institute of Technology, Japan
Simona Vasilache, University of Tsukuba, Japan

SOFTENG 2026 Publicity Chairs

José Miguel Jiménez, Universitat Politecnica de Valencia, Spain
Francisco Javier Díaz Blasco, Universitat Politècnica de València, Spain
Ali Ahmad, Universitat Politècnica de València, Spain
Laura Garcia, Universidad Politécnica de Cartagena, Spain
Sandra Viciano Tudela, Universitat Politecnica de Valencia, Spain

SOFTENG 2026 Technical Program Committee

Khelil Abdelmajid, Landshut University of Applied Sciences, Germany
Mo Adda, University of Portsmouth, UK
Bestoun S. Ahmed, Karlstad University, Sweden
Issam Al-Azzoni, Al Ain University of Science and Technology, UAE
Khubaib Amjad Alam, Al Ain University, Abu Dhabi, UAE
Vahid Alizadeh, College of Computing & Digital Media - DePaul University, USA
Washington Almeida, Cesar School | Center of Advanced Studies and Systems of Recife, Brazil
Vu Nguyen Huynh Anh, Université Catholique de Louvain, Belgium
Pablo O. Antonino, Fraunhofer IESE, Germany
Darlan Arruda, University of Western Ontario, Canada
Jocelyn Aubert, Luxembourg Institute of Science and Technology (LIST), Luxembourg
Heitor Augustus Xavier Costa, Federal University of Lavras (UFLA), Brazil
Ali Babar, University of Adelaide, Australia
Doo-Hwan Bae, Software Process Improvement Center - KAIST, South Korea
Mohamed Basel Almourad, College of Technological Innovation - Zayed University, Dubai, UAE
Bernhard Bauer, University of Augsburg, Germany
Imen Ben Mansour, University of Manouba, Tunisia
Maya Benabdelhafid, Ecole Supérieure de Comptabilité et de Finances (ESCF) de Constantine, Algeria
Marciele Berger, University of Minho, Portugal
Marcello M. Bersani, Politecnico di Milano, Italy
Anna Bobkowska, Gdansk University of Technology, Poland
Pierre Bourque, ETS Montreal, Canada
Fernando Brito e Abreu, ISCTE-IUL & ISTAR-IUL, Portugal
Antonio Brogi, University of Pisa, Italy
Azahara Camacho, Universidad de Cádiz, Spain
Patricia Camacho, Universidad de Cádiz, Spain

José Carlos Metrôlho, Polytechnic Institute of Castelo Branco, Portugal
Luis Fernando Castro Rojas, University of Quindío, Colombia
Pablo Cerro Cañizares, Universidad Complutense de Madrid, Spain
Allaoua Chaoui, University Constantine 2 - Abdelhamid Mehri, Algeria
Dickson K.W. Chiu, The University of Hong Kong, Hong Kong
Arpit Christi, Weber State University, USA
Stefano Cirillo, University of Salerno, Italy
Ian M. Cook, RTX BBN Technologies, USA
Andrea D'Ambrogio, University of Rome Tor Vergata, Italy
Lilian Michele da Silva Barros, Instituto Tecnológico de Aeronáutica, Brazil
Luciano de Aguiar Monteiro, Institute of Higher Education iCEV - Teresina-Piauí, Brazil
Serge Demeyer, Universiteit Antwerpen, Belgium
Juergen Doellner, Hasso-Plattner-Institute for Digital Engineering | University of Potsdam, Germany
Tadashi Dohi, Hiroshima University, Japan
Sigrid Eldh, Ericsson AB, Sweden
Gencer Erdogan, SINTEF Digital, Norway
Fernando Escobar, PMI-DF Brasilia, Brazil
Vladimir Estivill-Castro, Universitat Pompeu Fabra, Spain
Naser Ezzati Jivan, Brock University, Canada
Faten Fakhfakh, National School of Engineering of Sfax, Tunisia
Thomas Fehlmann, Euro Project Office AG, Zürich, Switzerland
Stefano Forti, University of Pisa, Italy
Atef Gharbi, National Institute of Applied. Sciences and Technology, Tunisia
Kambiz Ghazinour, State University of New York, Canton, USA
Pablo Gordillo, Universidad Complutense de Madrid, Spain
Adriana Guran, Babes-Bolyai University, Cluj-Napoca, Romania
Ulrike Hammerschall, University of Applied Sciences Munich, Germany
Noriko Hanakawa, Hannan University, Japan
Qiang He, Swinburne University of Technology, Australia
Philipp Helle, Airbus Group Innovations - Hamburg, Germany
Samedi Heng, Université de Liège, Belgium
Jang Eui Hong, Chungbuk National University, South Korea
Fu-Hau Hsu, National Central University, Taiwan
LiGuo Huang, Southern Methodist University, USA
Rui Humberto Pereira, ISCAP/IPP, Portugal
Carlos Hurtado Sánchez, Tecnológico Nacional de México - campus Tijuana, Mexico
Miren Illarramendi, Mondragon University, Spain
Shinji Inoue, Kansai University, Osaka, Japan
Anca Daniela Ionita, National University of Science and Technology POLITEHNICA Bucharest, Romania
Takashi Ishio, Future University Hakodate, Japan
Faouzi Jaidi, University of Carthage - Higher School of Communications of Tunis & National School of Engineers of Carthage, Tunisia
Jiajun Jiang, Tianjin University, China
Mira Kajko-Mattsson, Royal Institute of Technology, Sweden
Atsushi Kanai, Hosei University, Japan
Afrina Khatun, BRAC University, Bangladesh
Alexander Knapp, Universität Augsburg, Germany
Sondes Ksibi, University of Carthage | Higher School of Communications of Tunis, Tunisia

Luigi Lavazza, Università dell'Insubria, Italy
Dieter Landes, University of Applied Sciences Coburg, Germany
Seyong Lee, Oak Ridge National Laboratory, USA
Maurizio Leotta, University of Genova, Italy
Horst Lichter, RWTH Aachen University, Germany
Panos Linos, Butler University, USA
Hsin-Yu Liu, University of California San Diego, USA
Xiaobo Liu-Henke, Ostfalia University of Applied Sciences, Germany
Qinghua Lu, CSIRO, Australia
Damian M. Lyons, Fordham University, USA
Jianbing Ma, Chengdu University of Information Technology, China
Eda Marchetti, ISTI-CNR, Pisa, Italy
Johnny Marques, Aeronautics Institute of Technology, Brazil
Imen Marsit, University of Sousse, Tunisia
Núria Mata, Fraunhofer Institute for Cognitive Systems, Germany
Mohammadreza Mehrabian, South Dakota School of Mines and Technology, USA
Weizhi Meng, Lancaster University, UK
Edgardo Montes de Oca, Montimage, Paris, France
Fernando Moreira, Universidade Portucalense, Portugal
Ines Mouakher, University of Tunis El Manar, Tunisia
Mirna Muñoz, Centro de Investigación en Matemáticas (Cimat) - Unidad Zacatecas, Mexico
Malcolm Munro, Durham University, UK
Tsuyoshi Nakajima(中島毅), Shibaura Institute of Technology, Japan
Krishna Narasimhan, Itemis AG, Stuttgart, Germany
Risto Nevalainen, FiSMA (Finnish software measurement association), Finland
Jens Nicolay, Software Languages Lab | Vrije Universiteit Brussel, Belgium
Virginia Niculescu, Babes-Bolyai University, Cluj-Napoca, Romania
Stoicuta Olimpiu, University of Petrosani, Romania
Rafael Oliveira, UTFPR - The Federal University of Technology - Paraná, Brazil
Nelson Pacheco Rocha, University of Aveiro, Portugal
Alessandro Palma, Sapienza University of Rome, Italy
João Pascoal Faria, University of Porto, Portugal
Antonio Pecchia, Università degli Studi di Napoli Federico II, Italy
Fabiano Pecorelli, University of Salerno, Italy
Michael Perscheid, SAP Technology & Innovation, Germany
Dessislava Petrova-Antonova, Sofia University, Bulgaria
Tamas Pflanzner, University of Szeged, Hungary
Fumin Qi, National Supercomputing Center in Shenzhen (Shenzhen Cloud Computing Center), China
Zhengrui Qin, Northwest Missouri State University, USA
Stefano Quer, Politecnico di Torino, Italy
Łukasz Radliński, West Pomeranian University of Technology in Szczecin, Poland
Raman Ramsin, Sharif University of Technology, Iran
Zeeshan Ali Rana, National University of Computer and Emerging Sciences (FAST-NUCES), Lahore, Pakistan
Miary Andrianjaka Rapatsalahy, University of Fianarantsoa, Madagascar
Hajarisena Razafimahatratra, University of Fianarantsoa, Madagascar
Saif Ur Rehman Khan, Shifa Tameer-e-Millat University (STMU), Islamabad, Pakistan
Mohammad Reza Nami, Islamic Azad University-Qazvin, Iran

Oliviero Riganelli, University of Milano - Bicocca, Italy
Simona Riurean, University of Petrosani, Romania
António Miguel Rosado da Cruz, Higher School Technology and Management - Polytechnic Institute of Viana do Castelo, Portugal
Gunter Saake, Otto-von-Guericke-Universitaet, Magdeburg, Germany
Sébastien Salva, University Clermont Auvergne, France
Hiroyuki Sato, University of Tokyo, Japan
Andreas Schweiger, Airbus Defence and Space GmbH, Germany
Mahaboob Subhani Shaik, Veristat, USA
Richa Sharma, Commonwealth University, Lock Haven Campus, USA
Josep Silva Galiana, Universitat Politècnica de València, Spain
Rocky Slavin, University of Texas at San Antonio, USA
Jacopo Soldani, University of Pisa, Italy
Cristovão Sousa, Polytechnic Institute of Porto / INESC TEC, Portugal
Sinan Tanilkan, Norwegian Computing Center, Norway
Christos Troussas, University of West Attica, Greece
Anuj Tyagi, RingCentral Inc., USA
Harsh Vardhan, Vanderbilt University, USA
Simona Vasilache, University of Tsukuba, Japan
Miroslav Velev, Aries Design Automation, USA
Flavien Vernier, Université Savoie Mont Blanc, France
László Vidács, University of Szeged, Hungary
António Vieira, University of Minho, Portugal
Gianmario Voria, University of Salerno, Italy
Shaohua Wang, New Jersey Institute of Technology, USA
Ralf Wimmer, Concept Engineering GmbH / Albert-Ludwigs-Universität Freiburg, Freiburg im Breisgau, Germany
Andreas Wübbeke, Fachhochschule Südwestfalen - University of Applied Sciences, Germany
Xiaofei Xie, Nanyang Technological University, Singapore
Rui Yang, Xi'an Jiaotong-Liverpool University, China
Cemal Yilmaz, Sabanci University, Istanbul, Turkey
Levent Yilmaz, Auburn University, USA
Jinquan Zhang, Palo Alto Networks, USA
Zidong Zhang, Simon Fraser University, Canada
Rui Zhong, Palo Alto Networks, USA
Alejandro Zunino, ISISTAN, UNICEN & CONICET, Argentina
Ahang Zuo, Université de Pau et des Pays de l'Adour, France

Copyright Information

For your reference, this is the text governing the copyright release for material published by IARIA.

The copyright release is a transfer of publication rights, which allows IARIA and its partners to drive the dissemination of the published material. This allows IARIA to give articles increased visibility via distribution, inclusion in libraries, and arrangements for submission to indexes.

I, the undersigned, declare that the article is original, and that I represent the authors of this article in the copyright release matters. If this work has been done as work-for-hire, I have obtained all necessary clearances to execute a copyright release. I hereby irrevocably transfer exclusive copyright for this material to IARIA. I give IARIA permission or reproduce the work in any media format such as, but not limited to, print, digital, or electronic. I give IARIA permission to distribute the materials without restriction to any institutions or individuals. I give IARIA permission to submit the work for inclusion in article repositories as IARIA sees fit.

I, the undersigned, declare that to the best of my knowledge, the article is does not contain libelous or otherwise unlawful contents or invading the right of privacy or infringing on a proprietary right.

Following the copyright release, any circulated version of the article must bear the copyright notice and any header and footer information that IARIA applies to the published article.

IARIA grants royalty-free permission to the authors to disseminate the work, under the above provisions, for any academic, commercial, or industrial use. IARIA grants royalty-free permission to any individuals or institutions to make the article available electronically, online, or in print.

IARIA acknowledges that rights to any algorithm, process, procedure, apparatus, or articles of manufacture remain with the authors and their employers.

I, the undersigned, understand that IARIA will not be liable, in contract, tort (including, without limitation, negligence), pre-contract or other representations (other than fraudulent misrepresentations) or otherwise in connection with the publication of my work.

Exception to the above is made for work-for-hire performed while employed by the government. In that case, copyright to the material remains with the said government. The rightful owners (authors and government entity) grant unlimited and unrestricted permission to IARIA, IARIA's contractors, and IARIA's partners to further distribute the work.

Table of Contents

Knowledge Reuse in ML Prototyping: Insights from an Interview Study — 1
Selin Coban, Patrick Chrestin, and Horst Lichter

Enhancing IoT Requirements Through Layered Contextual Information — 9
Lasse Harjumaa

Fine-tuned Random Forest-based Software Defect Prediction via Metaheuristics — 15
Tadashi Dohi, Jingchi Wu, Junjun Zheng, and Hiroyuki Okamura

Towards Trust Engineering in Open Data Systems: A Layered Conceptual Framework Integrating Quality Assurance and Governance Perspectives — 21
Luciano Santos Pinheiro, Cristiane de Holanda de Barros e Silva, Vitor Barros Aquino, Thays Maria da Conceicao Silva Carvalho, Cristiano Vale do Rego Barros Filho, and Washington Henrique Carvalho Almeida

Towards the Static Detection of Compromised Software Components by Topological Anomalies — 29
Oscar Z. de Paiva, Wilson V. Ruggiero, and Marcos A. Simplicio Jr.

Knowledge Reuse in ML Prototyping: Insights from an Interview Study

Selin Coban
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
e-mail: coban@swc.rwth-aachen.de

Patrick Chrestin
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
e-mail: patrick.chrestin@rwth-aachen.de

Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
e-mail: lichter@swc.rwth-aachen.de

***Abstract*—Prototyping is a core activity in Machine Learning (ML) solution development, yet research on reuse in this context has largely focused on source code, overlooking more comprehensive forms of knowledge that ML developers rely on. This paper reports on a qualitative interview study with 18 ML developers from academia and industry that investigates how knowledge is sought, evaluated, reused, and retained during ML prototyping. Our thematic analysis reveals that ML developers reuse not only code but also declarative knowledge (e.g., terminology and foundational concepts) and procedural knowledge (e.g., design patterns and instructions). Search strategies are opportunistic and context-driven. Reuse decisions are based on quality, intuition, and direct experimentation. Notably, retention practices are often unsystematic, relying on memory or unstructured notes, which hinders the effective transfer of knowledge across projects. With this paper, we contribute a more structured perspective on knowledge reuse in ML prototyping. We further highlight the need for dedicated tools to support knowledge management in ML solution prototyping and suggest directions for systematically retaining this knowledge to increase its reuse.**

Keywords-Knowledge Reuse; Prototyping; Machine Learning.

I. INTRODUCTION

Prototyping is a central practice in software engineering, enabling developers to explore alternatives and mitigate risk. In Machine Learning (ML), however, prototyping serves a distinct purpose. Rather than emphasizing architectural or implementation decisions, ML prototyping is driven by rapid experimentation with data, algorithms, and models to assess the feasibility and potential value of a solution before committing to product development.

ML solution prototyping is inherently exploratory. When addressing a specific problem, practitioners navigate the solution space by examining related problems and leveraging existing solutions, reusing ideas, approaches, and artifacts throughout the process. Although code reuse in ML prototyping has been studied, considerably less is known about how practitioners reuse more comprehensive forms of knowledge, such as methodological choices or evaluation strategies.

Reuse is an essential means of developing more efficiently. Reuse in software development is generally defined as a process guided by a specific strategy that specifies what can be reused and how [1]. This strategy dictates the development and management of software artifacts to ensure they are easily accessible for reuse. The reuse process must be adapted to its context, as reuse activities must also be integrated with development activities. Dedicated tools for reuse activities are required to quickly and easily retrieve and apply reusable artifacts.

Without a clear understanding of how knowledge is reused during ML solution prototyping, valuable insights generated in this phase risk remaining fragmented, inconsistently retained, or lost altogether, limiting opportunities for reuse across projects. A deeper understanding of ML developers' knowledge-reuse practices can inform the design of processes and tools that better support ML work, reduce duplication of effort, and enable more systematic exploration of alternative solutions. To address this gap, we conducted and analyzed an interview study of ML developers focused on knowledge reuse during prototyping.

The remainder of this paper is structured as follows. Section II summarizes the related work on knowledge reuse. We formulate the research questions in Section III. Section IV describes the study method used to answer these research questions. Section V presents the results of the study for each question. In Section VI, we discuss these results and formulate some conclusions for improved prototyping tool support. Section VII outlines the threats to validity. Finally, Section VIII concludes the paper and suggests directions for future research.

II. RELATED WORK

This section reviews existing research on knowledge reuse in software engineering and ML solution prototyping. It further briefly introduces the Information Foraging Theory, which explains how people search for and select information.

A. Knowledge Reuse in Software Engineering

In knowledge management, a distinction is made between *explicit* knowledge, which is documented and readily shareable, and *tacit* knowledge, which encompasses experiential know-how [2]. In software engineering, knowledge is commonly categorized into four types: organizational, managerial, technical, and domain-specific. Such knowledge may be embedded within software artifacts, for example, domain models, or exist solely in the minds of developers, as in the case of design decisions.

Knowledge reuse has long been studied in software engineering, with most work focusing on source code and model reuse, often supported by recommender systems [3]–[5]. Adjandra et al. conducted a systematic literature review to provide an overview of the challenges of knowledge reuse

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

in software engineering. They identified insufficient documentation, knowledge silos, and the inherent complexity of knowledge management as key factors hindering efficient reuse [6]. Exploratory empirical studies, such as [7], show that source code and design patterns are among the most frequently reused artifacts, and that the quality of artifact documentation strongly influences the likelihood of reuse. Source code is often copied and reused across projects, a practice that can negatively impact software quality [8].

Stack Overflow has been studied extensively as a source of code reuse, particularly with respect to reuse behavior and recommender support [9]–[12].

B. Knowledge Reuse in ML Solution Prototyping

With regards to ML prototyping, previous studies have identified widespread source code reuse in Jupyter Notebooks (notebooks short), often in the form of source code cloning [13]–[16], particularly from sources such as Stack Overflow [17] and GitHub [18]. Researchers have also examined the types of activities supported by such source code reuse, with data visualization being among the most common [19].

Since reusing source code is an essential activity during prototype development, researchers have also investigated how tools integrated into notebooks can support this activity by retrieving relevant code more efficiently than manual searches. NBSEARCH supports semantic code search in notebooks [20]. JUPYSIM helps developers retrieve similar notebooks based on user queries, but constructing complex graph-based queries may hinder adoption [21]. ELYRA enables manual storage and management of source code snippets extracted from notebooks [22]. A fully automatic recommendation approach is presented in TYPHON, which primarily relies on the similarity of markdown text cells rather than code cells [23]. For data analysis and exploration, PYSNIPPET recommends code snippets from ML library documentation and Stack Overflow [24], while EDAASSISTANT offers code search and recommendation functionality [25]. Recently, Large Language Models (LLMs) have been explored to enable semantic search notebooks [26].

Regarding ML model reuse, Peixoto et al. [27] propose a recommendation approach for ML models based on data similarity. AutoML techniques aim to automate the process of selecting and configuring ML models for a given task. Some AutoML approaches, particularly those using meta-learning, leverage knowledge from previous ML problems to guide model selection and hyperparameter optimization [28].

Further best practices supporting source code reuse include ensuring the reproducibility of notebooks, sharing data along with notebooks, and publishing ML libraries [15][29]–[31].

C. Information Foraging Theory

The Information Foraging Theory (IFT) explains how people seek information by balancing its expected value against the cost of acquiring it [32]. According to IFT, users aim to maximize information gain while minimizing time and cognitive effort, particularly in complex and uncertain environments, such as the Internet.

A key concept in IFT is information scent, which refers to cues that signal the potential usefulness of an information source, such as titles or images on a webpage. Users rely on these cues to quickly decide which sources to explore further and which to abandon.

IFT also characterizes information-seeking behavior as opportunistic and iterative. Users refine their queries iteratively, switch between sources, and adjust their goals based on intermediate results. Hereby, users often favor shallow exploration of multiple sources over deep analysis of a few, especially when time is constrained. In this paper, IFT is used as additional context to interpret the findings of our interview study.

III. RESEARCH QUESTIONS

Despite its importance, knowledge reuse in ML prototyping remains insufficiently understood. Studies of Jupyter Notebook practices have highlighted extensive code cloning but have paid little attention to the broader spectrum of knowledge artifacts and to the strategies practitioners use to search for, evaluate, and retain them. As a result, we lack a comprehensive understanding of how ML developers actually engage in knowledge reuse during prototyping.

The review of publications related to the knowledge reuse in ML prototyping provided the basis for the following research questions:

- **RQ1**: What types of knowledge are reused?
- **RQ2**: How do ML developers search for knowledge?
- **RQ3**: Where do ML developers search for knowledge, and what tools do they use to do so?
- **RQ4**: How do ML developers evaluate knowledge and its sources to decide whether they want to reuse them?
- **RQ5**: How do ML developers retain relevant knowledge?

Overall, by answering these research questions, we aim to provide a holistic conceptual view of knowledge reuse in ML solution prototyping, focusing on the type of knowledge reused (RQ1), the reuse process (RQ2, RQ3, RQ5), and the decision-making mechanisms underlying reuse (RQ4).

This paper addresses these questions, providing a basis for examining the full range of knowledge types reused by ML developers and their strategies for searching, evaluating, and retaining knowledge.

IV. STUDY METHOD

We opted for Cognitive Task Analysis (CTA) [33], in which participants verbalize their reasoning as they mentally walk through a task guided by semi-structured interview questions. Participants were presented with a concrete ML solution prototyping scenario to ensure that their responses were grounded in actual practice. Our goal was to understand why ML developers make specific reuse decisions during prototyping, insights that cannot be reliably captured through observation alone. Many reuse decisions stem from internal cognitive processes, such as recalling prior solutions, evaluating alternatives, and weighing trade-offs, which are not directly observable.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

To this end, we conducted semi-structured interviews to collect qualitative data, followed by a thematic analysis of the interview transcripts to generate both qualitative insights and quantitative observations. This approach enables us to examine not only the knowledge and practices reported, but also their frequency of occurrence across participants.

A. Interview Design and Execution

Participants were asked how they would approach the following simple and abstract ML classification task:

> You are given the map of a public building. The map contains rooms, hallways, the library, the dining room, as well as all further offices. You are also given an extensive dataset of walking patterns, that are classified in GROUP-1, GROUP-2, GROUP-3, and OTHER. Your task now is to train a model that predicts the type of group when given a walking pattern.

We selected this task to evaluate not technical proficiency or domain knowledge, but the reasoning processes underlying knowledge search and reuse. With this task as a foundation, the interview questions were closely aligned with our research questions.

Before the main study, we conducted a pilot interview to refine the protocol and estimate its expected duration. In total, we interviewed 18 participants from both academic and industry contexts. An overview of the demographic information is presented in Table I. Participants were recruited through professional networks to ensure heterogeneity in disciplinary backgrounds and expertise in software engineering and machine learning. All participants reported having experience with ML solution prototyping.

TABLE I. INTERVIEW PARTICIPANTS BY POSITION TYPE AND DOMAIN.

ID	Role	Domain	ML Exp. (yrs)	Org. Size
Industry Practitioners				
1	Software Developer	IT Consulting and Software Dev.	3	<250
2[A]	ML Practitioner	Public Transport	4.5	<50
3[A]	ML Practitioner	Public Transport	9	<50
4	Student Worker	Information Processing	2	<50
5	Data Analyst	Finance	1	1
8	Data Owner	Clothing Retail	3	>10k
10	Software Developer	Real Estate	2	<250
12	Consultant	Corporate Consulting	4	<50
14	Software Developer	Process Automation	3	<50
15	Software Developer	AI Consulting	4.5	<50
16	AI Expert	Optimization Software	1	<5k
18	Consultant	Green Energy	1.5	<5k
Academic Practitioners				
6	Ph.D. Student	Manufacturing	3	<10
7	Student	Computer Science	2	N/A
9	Ph.D. Student	Materials Engineering	5	<100
11	Ph.D. Student	Medicine	0.5	<25
13	Researcher	Mechanical Engineering	1	N/A
17*	Ph.D. Student	Computer Vision	6	<25

* Interview split into two parts due to interviewee availability. IDs assigned in interview order; same letter indicates participants from the same organization. N/A means that the organization size for these participants is unknown.

The interviews were conducted by one researcher via videoconferencing between December 2024 and March 2025. All participants were informed about the purpose of the interview and the procedures for handling data. Each interview lasted between 32 and 87 minutes. All interviews were audio-recorded with participant consent, transcribed verbatim, pseudonymized, and stored for analysis.

B. Interview Analysis

To analyze the transcripts, we applied Thematic Analysis (TA) according to Braun and Clarke [34], using an inductive and semantic coding approach. This choice was driven by the exploratory nature of our study, which aimed to understand how ML developers search for and reuse knowledge.

Two researchers, referred to as the "TA Team," carried out the coding and theme development. In line with TA, codes were generated directly from the data, reflecting participants' explicit statements. Through discussion and refinement, the TA team organized these codes into conceptual categories, which it then developed into themes. The TA team determined the frequency with which each code and topic appeared in the interviews.

The TA process involved the following steps:

- *Data familiarization:* The TA team immersed themselves in the transcripts to gain a comprehensive understanding of the content.
- *Initial coding cycle:* Next, the TA team independently conducted open, inductive coding using the qualitative analysis tool *Delve* [35]. This step involved identifying and labeling meaningful data segments related to the research questions. Codes were kept descriptive and grounded closely in the participants' language, minimizing interpretive bias and preserving original intent.
- *Consensus and second coding cycle:* After the initial coding cycle, the TA team compared results, discussed differences, and reached consensus on the codebook. The codebook was then systematically applied to all transcripts in a second coding cycle to ensure consistency and analytical rigor.
- *Theme development:* The TA team organized the codes into categories, serving as candidate themes. These candidate themes were iteratively refined and aligned with the research questions.
- *Theme refinement and naming:* The TA team refined the themes through collaborative discussions to ensure clarity. Each theme was named to reflect its essence.
- *Reporting:* In this paper, we report on the results achieved. We excluded all codes that appeared in fewer than four interviews (less than 20% of the total), except where a code provided significant interpretive value.

Across the 18 interviews, we identified 89 unique codes. Only ten appeared in fewer than four interviews, and only four were mentioned a single time. This distribution indicates that most practices occurred across participants, suggesting that thematic saturation was achieved primarily. Additional interviews would likely have revealed only minor or context-specific practices rather than new overarching themes.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

C. Interview Material

We provide the complete dataset, including themes, codes, quotes, their mappings, interview questions, full transcripts, and a code-by-code occurrence matrix as supplementary material on Zenodo [36].

V. Results

This section presents all identified themes and subthemes, organized by research question. Each theme is labeled (e.g., T1 for theme 1); participants are labeled similarly (e.g., P1 refers to participant 1).

A. What types of knowledge are reused (RQ1)?

Established concepts in knowledge management guided the coding and categorization of identified knowledge into broader themes. Our analysis revealed that ML developers actively search for *explicit* knowledge. We distinguish three partially interrelated categories: *declarative*, *procedural*, and *executable* knowledge. We use the term *knowledge element* to refer to explicit, codified units of knowledge that can be stored, communicated, and reused. Based on these categories, Table II provides an overview of the types of knowledge elements mentioned by ML developers.

T1 Declarative Knowledge: It refers to factual and conceptual knowledge that ML developers use to understand the problem domain or to frame their solution approach [37]. Seven participants searched for relevant *terminology* within the problem domain to further refine their search queries. They also explored *foundations*, such as commonly used metrics or benchmarks, which they considered necessary for evaluating the relevance and quality of reusable solutions. 12 ML developers searched for *ML algorithms* to determine which ones could be useful for the given problem. Although algorithms inherently involve procedures, participants typically sought high-level overviews and conceptual distinctions (e.g., when to use a random forest), rather than implementation details.

T2 Procedural Knowledge: It refers to knowledge that combines conceptual understanding with guidance on how to act [37]. 11 participants searched for a *solution design pattern*, meaning a general sequence of steps required to achieve a specific goal, including data preprocessing and feature engineering. Others explicitly sought *step-by-step instructions* that combine solution steps with practical guidance, such as code snippets or tutorials, directly applicable to the given problem. Both guide ML developers in applying their knowledge and skills to solve the given ML problem.

T3 Executable Knowledge: It refers to knowledge elements that can be executed directly or after integration into software systems. These elements often embed both declarative and procedural knowledge, typically in the form of code. Examples include code snippets, ML libraries, and pre-trained ML models. Such elements are reused to reduce implementation effort and accelerate development.

TABLE II. Searched Knowledge Elements.

Category	Knowledge Element Type	#Part.
T1 Declarative Knowledge	ML Algorithm	12
	Foundations	8
	Terminology	7
T2 Procedural Knowledge	Solution Design Pattern	11
	Step-by-Step Instruction	6
T3 Executable Knowledge	Code Snippet or Repository	12
	ML Library	6
	Pre-trained ML Model	4

In practice, ML developers blend different types of knowledge: understanding key concepts (declarative), applying structured approaches or steps (procedural), and adapting existing code (executable).

B. How do ML developers search for knowledge (RQ2)?

To answer this question, the TA team analyzed participants' accounts of their search approaches and grouped the identified codes into three themes (see Table III), which we describe in the following section.

TABLE III. Applied Search Approaches.

Search Approaches	#Part.
T4 Opportunistic Behavior	18
T5 Executable First	10
T6 Solution Idea First	8

T4 Opportunistic Behavior: In line with IFT, the TA team observed *opportunistic search behavior* among all participants in their choice of search and retrieval tools, knowledge sources, and how they evaluated the quality of knowledge elements. Many relied on surface-level quality evaluation, such as recognizing familiar sources or drawing on prior experiences, to rapidly select and evaluate the quality of knowledge elements. This behavior aligns with the concept of information scent, where individuals are drawn to sources that appear to offer valuable and actionable information while requiring minimal cognitive effort.

T5 Executable First: Ten participants mentioned that they first search for executable knowledge that could directly address the problem at hand. This was especially dominant with academic practitioners. For example, P7 noted that understanding why an existing solution works is often easier than defining an entire solution concept from scratch. Two participants, one from industry and one from academia, noted that they did not care about the internal mechanisms as long as the solution effectively addressed the problem. This behavior aligns with opportunistic behavior as these participants favor rapid progress over deeper conceptual understanding.

T6 Solution Idea First: In contrast to T5, eight participants stated that when searching, they first focus on understanding the underlying reasoning behind an existing solution design pattern before translating it into executable code. These approaches were not mutually exclusive: Some participants first explored an executable knowledge element and subsequently worked to understand the conceptual ideas behind it, while

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

others started with forming a conceptual idea before selecting an executable knowledge element. Both approaches were observed among ML developers from both industry and academia.

C. Where do ML developers search for knowledge, and what tools do they use to do so (RQ3)?

During coding, the TA team identified different knowledge sources as well as search and retrieval tools. These were grouped into the theme *search & retrieval tools*, the purpose-based themes *sources for learning* and *sources for coding*, and the separate theme *human sources*. A complete overview is provided in Tables IV and V.

TABLE IV. TOOLS USED FOR SEARCHING AND RETRIEVING KNOWLEDGE.

T7 Search and Retrieval Tool Type	#Part.
Web Search Engine	18
Scholarly Literature Search Engine	10
LLM	9

TABLE V. USED KNOWLEDGE SOURCES.

Category	Knowledge Source Type	#Part.
T8 Sources for Learning	Blog Article	16
	Scientific Paper	11
	Video	10
	Book	6
T9 Sources for Coding	Stack Overflow Post	14
	Technical Documentation	10
	Personal Code Archive	10
	Git Repository	9
T10 Human Sources	Colleague / Expert	17

T7 Search & Retrieval Tools: ML developers often begin by using web search engines, such as Google, or scholarly literature search engines, like Google Scholar, to investigate unfamiliar problems and identify existing solutions. Nine participants also utilized LLMs (e.g., ChatGPT) for this purpose, highlighting that they enable a more interactive and conversational approach to exploring knowledge elements. Six participants also mentioned using LLMs for code generation and content summarization. Strikingly, a larger proportion of ML developers from academia (4 out of 6) reported using some form of AI assistance compared to their counterparts in industry (2 out of 12).

T8 Sources for Learning: To explore the solution space, ML developers used scientific papers. This tendency was particularly pronounced among academic ML developers (5 out of 6), whereas two participants from industry reported that the knowledge in scientific papers was too specific. Blogs were preferred for clarity and practical examples. Videos were considered useful for hands-on learning and tutorials. However, they were sometimes considered less efficient than skimming text. Books were mentioned as a source of declarative knowledge, but were also said to be more difficult to search through.

T9 Sources for Coding: For coding-related questions, Stack Overflow was the primary source, valued primarily for the community feedback attached to each answer. Technical documentation (e.g., the scikit-learn user guide) was particularly valued for guiding reuse, with eight ML developers from industry and only two from academia. Git repositories offer reusable code, often accessed through links provided in blogs and papers. Ten ML developers reported searching their code archives, although disorganization sometimes limited their usefulness. The executable files within the coding sources are typically "copied and pasted" or downloaded and then modified.

T10 Human Sources: Participants consulted colleagues or experts, although some hesitated to seek help due to concerns about interrupting or disturbing them. Although not discussed in the interviews, human knowledge sources can also share tacit knowledge, i.e., personal experiences with specific solution approaches.

D. How do ML developers evaluate knowledge and its sources to decide whether they want to reuse them (RQ4)?

To answer this research question, the TA team identified codes for each mentioned criterion, by which a knowledge element and its source are evaluated. Then, these criteria were grouped by the overarching concept they describe. As a result, the TA team identified three themes, namely *quality-based*, *intuition-based*, and *experiment-based* evaluation approaches (see Table VI). We describe each approach below.

TABLE VI. APPLIED KNOWLEDGE EVALUATION APPROACHES.

Evaluation Approach	#Part.
T11 Quality-based	18
T12 Experiment-based	15
T13 Intuition-based	10

TABLE VII. QUALITY CRITERIA USED FOR EVALUATING KNOWLEDGE ELEMENTS.

Quality	Quality Criterion	#Part.
T11.1 Usability	Compatibility with development environment	15
	Writing style	14
	Ease of understanding	14
	Presence of examples, code, or tutorials	12
	Effort required to reuse the solution	10
T11.2 Credibility	Recognized author or publisher	14
	Community feedback	12
	Transparency of decisions and results	11
	Match with personal knowledge or experience	8
	Cited or mentioned frequently	6
T11.3 Relevance	Context match	17
	Recency of publication or update	14

T11 Quality-based Evaluation: Three key qualities were used to evaluate knowledge elements and sources: *usability*, *credibility*, and *relevance* (see Table VII).

T11.1 Usability: It refers to how easily an ML developer can understand and apply a knowledge element. This includes, among other things, the perceived ease of understanding, the effort required for reuse, and ease of integration into the ML developer's development environment. If the perceived effort required for reuse is too high, ML developers discard solutions

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

even when they would yield high performance. This occurs, for example, when programming languages are incompatible.

T11.2 Credibility: It describes the perceived trustworthiness of the source. ML developers consider criteria such as the author's expertise, community feedback, or the number of citations. ML developers tend to favor sources created by recognized experts, which are perceived as more credible due to the assumed rigor of their development, and therefore spend less time critically analyzing them.

T11.3 Relevance: It concerns the degree to which a knowledge element matches with the ML developer's problem context, i.e., problem definition and data. The publication or update date is a frequently mentioned criterion, since older knowledge elements may be outdated. Practitioners typically assess context match first, before investing further effort in analyzing the knowledge element.

T12 Experiment-based Evaluation: In the case of executable knowledge elements, such as code snippets, 15 ML developers prefer to directly experiment with the artifact to evaluate its strengths, limitations, and overall suitability. This also helps them gain a better understanding (P13) or a "better feeling" (P14) of the solution space. This was frequently reported by all ML developers from academia.

T13 Intuition-based Evaluation: ML developers frequently rely on their intuition to make quick judgments about the value of a knowledge element or the credibility of its source. This trend was particularly pronounced among ML developers from industry (7 out of 12), compared to those from academia (3 out of 6). Furthermore, this tendency is more pronounced among ML developers with at least two years of experience (9 out of 10).

E. How do ML developers retain relevant knowledge (RQ5)?

ML developers employ a range of approaches to document and manage relevant knowledge as they solve problems. To develop themes, the TA team grouped codes by the nature of the retention approaches, distinguishing between *absent*, *ad-hoc*, *unstructured*, and *structured* approaches (see Table VIII).

TABLE VIII. APPLIED KNOWLEDGE RETENTION APPROACHES.

Retention Approach	Mentioned Example	#Part.
T14 Absent	Rely on memory only	10
T15 Ad-hoc Bookmarking	Keeping tabs open Revisiting search history	7 2
T16 Unstructured Note-taking	Taking digital or physical notes Leaving comments in code	8 3
T17 Structured Documentation Tools	Notes in dedicated apps Curating collection of links	6 5

T14 Absent: Ten participants relied solely on memory, keeping knowledge elements "in mind" as they worked. Among participants adopting this strategy, seven out of ten were from industry.

T15 Ad-hoc Bookmarking: A commonly reported approach was to rely on the search history or keep browser tabs open, often resulting in what P2 described as "tab chaos". Tabs were used to temporarily store promising knowledge sources, which participants planned to revisit later in the process. These ML developers also reported retaining the information only until the problem was resolved, indicating no intention to reuse the knowledge in the future.

T16 Unstructured Note-Taking: Another approach is to keep physical or digital notes. Eight participants described writing down key ideas and partial solutions in notebooks or using digital note-taking tools, motivated by the desire to explain their design decisions at any time. Three participants also added comments in the code to document the origin or logic of reused code snippets. These notes served both as external memory aids and as a way to organize thoughts during exploration.

T17 Structured Documentation Tools: A subset of participants used more organized approaches for long-term knowledge retention. Commonly mentioned tools included digital note-taking applications with filtering and tagging capabilities, as well as Overleaf, Citavi, and Microsoft Office software.

VI. DISCUSSION

Our sample included participants with diverse backgrounds, roles, and domains. While this variation might be expected to introduce substantial differences in prototyping and reuse practices, our analysis revealed a high degree of consistency across participants. The vast majority of codes occurred in multiple interviews, and the core themes emerged independently of role, experience level, or industry context. This suggests that the identified practices are robust across heterogeneous ML developer groups.

Based on the research questions and study results, we identified the central reuse-related concepts and their relationships (see Figure 1). This conceptual model can be interpreted as follows: Depending on the selected *search approach*, the *knowledge search* is performed using dedicated *search and retrieval tools*. These tools retrieve *knowledge sources* that capture *knowledge elements* of different *knowledge types*. The retrieved knowledge elements are systematically assessed using a selected *evaluation approach* to determine their suitability for the current ML prototyping context. Accepted knowledge elements may be retained using a *retention approach*, thereby facilitating their reuse in subsequent projects or iterations.

A. Implications for Knowledge Providers

By identifying evaluation approaches, we provide insights into how ML developers decide whether to reuse knowledge and which criteria influence these decisions. Knowledge providers can leverage this understanding to strategically strengthen information scent, thereby increasing the likelihood of reuse. Moreover, organizations can use these insights to establish standardized practices for sharing knowledge generated during ML solution prototyping.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

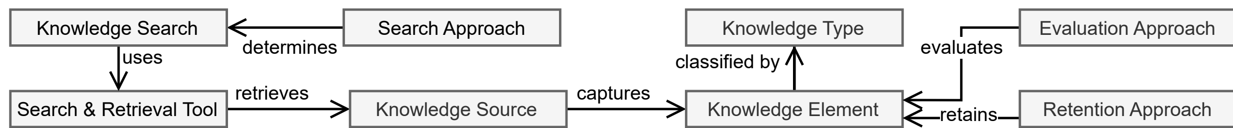


Figure 1. Concept Model of Reuse in ML Solution Prototyping

B. Implications for Tools

More than half of the participants do not employ sustainable methods to retain reusable knowledge. Knowledge thus tends to become siloed at the individual level, either mentally or in personal code archives, as evidenced by the fact that 17 participants reported relying on human sources.

The results of our study highlight several important directions for future tool development to address the challenges of knowledge retention in ML prototyping. The absence of systematic approaches to retaining reusable knowledge is due not only to insufficient tooling but also to limited integration with ML developers' daily workflows and collaborative platforms.

To address these issues, future tools should prioritize features that support *collaborative management* and *long-term preservation* of both knowledge sources and elements. Seamless integration with widely used IDEs and collaborative platforms can help embed retention practices into everyday work. Additionally, *automated tracing* between artifacts and reused knowledge elements and their sources would facilitate better organization and retrieval of information over time. Incorporating mechanisms for *quality-based evaluation* into knowledge retention systems could further support ML developers, particularly given our finding that ML developers rely on numerous quality criteria but lack systematic means to manage them.

By addressing these needs, new tools have the potential to reduce individual knowledge silos, foster broader sharing of valuable knowledge across projects and teams, and ultimately increase the degree of reuse in ML solution prototyping.

VII. Threats to Validity

Internal Validity: To mitigate threats to internal validity, we designed a semi-structured interview aligned with our research questions. Two researchers independently coded all transcripts and resolved differences through discussion to limit individual bias. However, some subjectivity remains inherent in thematic analysis, and our results depend on what participants chose to share during their reflections. Furthermore, internal validity may be affected by recall and social desirability bias, as participants may misremember their actions or present their decisions in a more favorable light. To mitigate this, we assigned participants a prototyping task.

External Validity: Our findings are based on interviews with 18 ML developers from industry and academia, covering a range of domains and roles. While this diversity supports the breadth of our insights, the limited sample size constrains generalizability. Future studies with larger samples could further strengthen the transferability of our findings.

VIII. Conclusion and Future Work

This interview study examines how ML developers search for, evaluate, reuse, and retain knowledge during solution prototyping. By broadening the focus beyond code reuse, our findings reveal that practitioners reuse a broad spectrum of explicit knowledge, including declarative, procedural, and executable knowledge.

Our results show that their search and evaluation behaviors align with the principles of information foraging theory and are strongly influenced by opportunistic decision-making. The entire reuse process is shaped by the trade-off between speed and systematic development. Activities such as understanding the solution, documenting design decisions, and ensuring sustainable knowledge retention are often neglected in favor of quickly producing "good enough" solutions.

Because opportunistic behavior discourages ML developers from investing time in manual knowledge retention, we want to investigate how retention processes can be supported or automated to promote knowledge reuse. Furthermore, we aim to evaluate the validity of our knowledge reuse concept, assessing the extent to which it reflects and accurately represents real-world practices.

References

[1] H. Mili, F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp. 528–562, 1995. DOI: 10.1109/32.391379.

[2] M. Paul, M. Engelhart, I. Rus, and S. Sinha, "Knowledge Management in Software Engineering - A State-of-the-Art report," Jan. 2001.

[3] A. Dyck, A. Ganser, and H. Lichter, "On Designing Recommenders for Graphical Domain Modeling Environments," in *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, INSTICC, SciTePress, 2014, pp. 291–299.

[4] L. Heinemann, "Facilitating Reuse in Model-Based Development With Context-Dependent Model Element Recommendations," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, 2012, pp. 16–20.

[5] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel, "Identifier-Based Context-Dependent API Method Recommendation," in *2012 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 31–40.

[6] W. Adjandra, Y. Putrapratama, A. Wiraguna, D. Sensuse, and N. Safitri, "Systematic Literature Review Knowledge Reuse in Software Development," in *Proceedings - 2nd International Conference on Computer Science and Engineering*, United States: Institute of Electrical and Electronics Engineers Inc., 2021.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

[7] V. Bauer, J. Eckhardt, B. Hauptmann, and M. Klimek, “An Exploratory Study on Reuse at Google,” in *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices*, ser. SERIPs 2014, Hyderabad, India: ACM, 2014, pp. 14–23.

[8] V. Bauer and B. Hauptmann, “Assessing Cross-Project Clones for Reuse Optimization,” in *2013 7th International Workshop on Software Clones (IWSC)*, 2013, pp. 60–61.

[9] A. Lotter, S. A. Licorish, B. T. R. Savarimuthu, and S. Meldrum, “Code Reuse in Stack Overflow and Popular Open Source Java Projects,” in *2018 25th Australasian Software Engineering Conference (ASWEC)*, 2018, pp. 141–150.

[10] S. Mahajan, N. Abolhassani, and M. R. Prasad, “Recommending Stack Overflow Posts for Fixing Runtime Exceptions Using Failure Scenario Matching,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE ’20, ACM, Nov. 2020, pp. 1052–1064.

[11] G. Melo, T. Oliveira, P. Alencar, and D. Cowan, “Knowledge Reuse in Software Projects: Retrieving Software Development QA Posts Based on Project Task Similarity,” *PLOS ONE*, vol. 15, no. 12, pp. 1–27, Dec. 2020.

[12] C. Ragkhitwetsagul and M. Paixao, “Recommending Code Improvements Based on Stack Overflow Answer Edits,” *Computing Research Repository (CoRR)*, vol. 2204, 2022. arXiv: 2204.06773 [cs.SE].

[13] R. Huang *et al.*, “How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and Opportunities,” *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 1243–1255, 2025.

[14] A. Koenzen, N. A. Ernst, and M. Storey, “Code Duplication and Reuse in Jupyter Notebooks,” *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–9, 2020.

[15] M. S. Siddik, H. Li, and C.-P. Bezemer, “A Systematic Literature Review of Software Engineering Research on Jupyter Notebook,” *Computing Research Repository (CoRR)*, vol. 2504, 2025. arXiv: 2504.16180 [cs.IR].

[16] B. van Oort, L. Cruz, M. Aniche, and A. van Deursen, “The Prevalence of Code Smells in Machine Learning projects,” in *1st IEEE/ACM Workshop on AI Engineering - Software Engineering for AI, WAIN@ICSE 2021, Madrid, Spain, May 30-31, 2021*, IEEE, 2021, pp. 35–42.

[17] M. Yang, Y. Zhou, B. Li, and Y. Tang, “On Code Reuse from StackOverflow: An Exploratory Study on Jupyter Notebook,” *Computing Research Repository (CoRR)*, vol. 2302, 2023. arXiv: 2302.11732 [cs.SE].

[18] M. Källén, U. Sigvardsson, and T. Wrigstad, “Jupyter Notebooks on GitHub: Characteristics and Code Clones,” *Computing Research Repository (CoRR)*, vol. 2007, 2020. arXiv: 2007.10146 [cs.SE].

[19] W. Epperson, A. Y. Wang, R. DeLine, and S. M. Drucker, “Strategies for Reuse and Sharing among Data Scientists in Software Teams,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’22, Pittsburgh, Pennsylvania: ACM, 2022, pp. 243–252.

[20] X. Li, Y. Wang, H. Wang, Y. Wang, and J. Zhao, “NBSearch: Semantic Search and Visual Exploration of Computational Notebooks,” in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’21, ACM, May 2021, pp. 1–14.

[21] M. Horiuchi, Y. Sasaki, C. Xiao, and M. Onizuka, “JupySim: Jupyter Notebook Similarity Search System.,” in *EDBT*, 2022, pp. 2–554.

[22] Elyra Team, *Code Snippets - Elyra 3.15.0 documentation*, https://elyra.readthedocs.io/en/v3.15.0/user_guide/code-snippets.html, [Acc. 03-Apr-2026], 2022.

[23] C. Ragkhitwetsagul *et al.*, “Typhon: Automatic Recommendation of Relevant Code Cells in Jupyter Notebooks,” *Computing Research Repository (CoRR)*, vol. 2405, 2024. arXiv: 2405.09075 [cs.SE].

[24] A. Watson, S. Bateman, and S. Ray, “PySnippet: Accelerating Exploratory Data Analysis in Jupyter Notebook through Facilitated Access to Example Code,” in *EDBT/ICDT Workshops*, 2019.

[25] X. Li, Y. Zhang, J. Leung, C. Sun, and J. Zhao, “EDAssistant: Supporting Exploratory Data Analysis in Computational Notebooks with In Situ Code Search and Recommendation,” *ACM Trans. Interact. Intell. Syst.*, vol. 13, no. 1, Mar. 2023, ISSN: 2160-6455.

[26] L. Li and J. Lv, “Unlocking Insights: Semantic Search in Jupyter Notebooks,” *Computing Research Repository (CoRR)*, vol. 2402, 2024. arXiv: 2402.13234 [cs.SE].

[27] E. Peixoto, D. Torres, D. Carneiro, B. Silva, and R. Marques, “Reusing ML Models in Dynamic Data Environments: Data Similarity-Based Approach for Efficient MLOps,” *Big Data and Cognitive Computing*, vol. 9, no. 2, 2025, ISSN: 2504-2289.

[28] F. Hutter, L. Kotthoff, and J. Vanschoren, *Automated Machine Learning: Methods, Systems, Challenges*. Springer Nature, 2019.

[29] R. Ahmad, N. N. Manne, and T. Malik, “Reproducible Notebook Containers using Application Virtualization,” in *2022 IEEE 18th International Conference on e-Science (e-Science)*, 2022, pp. 1–10.

[30] A. Rule *et al.*, “Ten Simple Rules for Reproducible Research in Jupyter Notebooks,” *Computing Research Repository (CoRR)*, vol. 1810, 2018. arXiv: 1810.08055 [cs.OH].

[31] J. Wang, T.-y. Kuo, L. Li, and A. Zeller, “Assessing and Restoring Reproducibility of Jupyter Notebooks,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’20, Virtual Event, Australia: ACM, 2021, pp. 138–149.

[32] P. Pirolli and S. Card, “Information Foraging.,” *Psychological Review*, vol. 106, no. 4, p. 643, 1999.

[33] O. Brown, N. Power, and J. Gore, “Cognitive task analysis: Eliciting expert cognition in context,” *Organizational Research Methods*, vol. 28, no. 3, pp. 375–404, 2025.

[34] V. Braun and V. Clarke, *Successful Qualitative Research: A Practical Guide for Beginners*. Sage Publications ltd, 2013.

[35] Delve, *Qualitative Data Analysis Software Delve*, https://delvetool.com, [Acc. 03-Apr-2026], 2025.

[36] S. Coban and P. Chrestin, *Artifacts from the Interview Study: Knowledge Reuse in ML Solution Prototyping*, Zenodo, Jan. 2026. DOI: 10.5281/zenodo.18196720. [Online]. Available: https://doi.org/10.5281/zenodo.18196720.

[37] T. Ten Berge and R. Van Hezewijk, “Procedural and Declarative Knowledge: An Evolutionary Perspective,” *Theory & Psychology*, vol. 9, no. 5, pp. 605–624, 1999.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. **ISBN: 978-1-68558-398-9**

Enhancing IoT Requirements Through Layered Contextual Information

Lasse Harjumaa
Kokkola University Consortium Chydenius
University of Jyväskylä
Kokkola, Finland
e-mail: lasse.m.harjumaa@jyu.fi

Abstract— **Internet of Things (IoT) systems operate in dynamic and heterogeneous environments where behavior depends on contextual assumptions such as connectivity, data quality, and operational intent. These assumptions often remain implicit, complicating validation and traceability in requirements engineering. This paper introduces Con², a lightweight context-aware approach that integrates intent-driven context modeling with executable behavioral specifications. Con² formalizes contextual assumptions as reusable context contracts linked to Gherkin scenarios. This separation of context and behavior improves explicitness, reduces duplication, and enables automated testing and runtime monitoring. A smart building lighting case study illustrates the approach, and an analytical comparison with a conventional use case highlights improvements in clarity and verifiability while maintaining compatibility with Agile practices.**

Keywords-Internet of Things; Requirements engineering; IoT Requirements; IoT development.

I. Introduction

IoT systems differ from conventional software systems in that they operate across heterogeneous infrastructures that connect cloud services with distributed networks of sensors and actuators. By integrating hardware and software across architectural layers, they create tight coupling between physical processes and digital logic. Requirements in IoT projects rarely remain stable; many emerge only during deployment and operational use. Development must therefore accommodate technological diversity while maintaining scalability and security, which calls for methods that capture requirements in a way that reflects the dynamic and context-dependent nature of IoT systems.

Contextual information forms the foundation of meaningful system behavior. Sensor measurements cannot be interpreted reliably without considering environmental conditions and user intent, and identical inputs may require different responses depending on operational circumstances. Context is thus essential rather than supplementary. Yet contextual assumptions often remain implicit and fragmented across architectural layers, with business meaning expressed at higher levels and sensing details confined to devices. This separation makes it difficult to relate functional requirements to the conditions under which they hold. To address this gap, we propose a context-aware requirements engineering framework that embeds contextual assumptions directly into executable behavioral specifications.

IoT systems operate in changing physical environments and evolving usage scenarios, which complicates the definition of behavior in advance. Requirements are refined iteratively as real-world feedback reveals new constraints and expectations. Such conditions favor lightweight, executable specifications that evolve with the system and can be validated continuously. Behavior-Driven Development (BDD) supports this need by expressing requirements as testable scenarios that function both as documentation and as automated verification artifacts [14].

The rest of this paper is organized as follows. Section II describes related work. Section III describes the importance of context in IoT. Section IV describes our approach for including context information in requirements. In Section V, constructs and usage of our solution are described. Section VI represents case study. Section VII provides evaluation of the approach, and Section VIII discusses the findings. Section IX concludes the paper.

II. Related Work

Research on context-aware computing provides foundational definitions that explain why context is essential for producing relevant system behavior. Dey's work [6] characterizes context-aware systems as those that use contextual information to deliver services aligned with a user's task. This notion of task-dependent relevance closely matches IoT scenarios, where identical measurements may lead to different interpretations or actions depending on operational purpose, such as monitoring or control. It also broadens the concept of context beyond sensor metadata by introducing an intent-driven perspective.

Within IoT, Perera et al. [11] survey context-aware computing for IoT and consolidate existing techniques and middleware support for managing context. They highlight the diversity of IoT environments and the complexity of context handling, but it focuses primarily on processing mechanisms rather than on how context should be elicited and structured as a requirement engineering artifact throughout the lifecycle.

Requirements engineering (RE) for IoT has been examined in several reviews that reveal both active research and fragmentation. A systematic mapping study by Aguilar-Calderón et al. [1] shows that most proposals address isolated aspects of RE and reflect the technical complexity of IoT systems. Similarly, da Silva Souza [5] identifies a range of RE processes and validation practices, suggesting a diverse landscape rather than a unified, end-to-end methodology.

Some efforts move toward more practical guidance. For example, RETIoT introduces structured scenarios and templates to support requirements work and reports encouraging results from project use [4]. Siakas et al. [12]

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

propose REFIoT, a framework that organizes IoT challenges across stakeholder and environmental dimensions to support systematic elicitation. Likewise, Boutot et al. [2] present UCM4IoT, a domain-specific modeling language that extends use cases to represent adaptive behavior across interconnected IoT components.

There are also examples of efforts to utilize Gherkin [3] language to systemize certain aspects of IoT development. Wang et al. [15] have proposed a lightweight enhancement for modeling concurrent and sequential behavior. Kannengiesser et al. [8] present an approach to address the challenges of testing Cyber-Physical Systems (CPS) by making abstractions of Gherkin test scenarios.

Despite this body of work, interest in IoT-specific RE has not significantly increased in recent years, and practice still relies largely on conventional techniques such as use cases or interviews [1]. Existing approaches remain fragmented and domain-bound, often targeting only selected phases of RE or specific applications. Given the inherently dynamic and context-dependent nature of IoT environments, more systematic methods are needed to capture contextual information and integrate it consistently into requirements across heterogeneous configurations.

In practice, context frequently remains implicit across architectural layers, which makes it difficult to connect high-level intent with low-level sensing and actuation behavior.

III. THE IMPORTANCE OF CONTEXT IN IOT

Perera et al. [11] define context as any information that characterizes an entity's situation. In software engineering, requirements are typically classified as functional or non-functional, with non-functional requirements shaping architectural and quality decisions throughout the lifecycle. In IoT systems, however, such qualities are rarely fixed. Properties like latency, reliability, energy use, or data freshness depend on operational conditions such as connectivity or device state, which makes many requirements inherently conditional rather than universally valid.

IoT applications involve diverse stakeholders who interact with different parts of a shared infrastructure. Because devices and services are interconnected across architectural layers, requirements may be interpreted differently depending on perspective. This variability increases the difficulty of specification and validation, particularly when contextual assumptions remain implicit.

When contextual knowledge is incomplete, data may become ambiguous at the application level. Although semantic metadata clarifies structure and relationships, contextual information is often only partially structured, limiting automated interpretation [13]. Misalignment between data producers and consumers can further distort meaning. In smart farming, for example, analytics-based recommendations may be disregarded when they conflict with established practices or experiential knowledge [7]. Resource-constrained devices add another layer of uncertainty by relying on external context sources that may be inconsistent or outdated [16]. Mouhim and Lachhab [10] survey contextual modeling approaches and highlights trade-offs between simplicity and expressiveness in different representation techniques.

More fundamentally, IoT requirements depend not only on inputs but also on situational context. In traditional software, identical inputs usually lead to identical outcomes. In IoT systems, the same input may produce different behavior when surrounding conditions change. Requirements therefore become adaptive and situational rather than stable and environment independent.

The layered architecture of IoT systems reinforces this dynamic. Contextual meaning emerges gradually as data moves upward through the system. A raw accelerometer reading at the device layer represents physical acceleration only; subsequent layers interpret and enrich this signal until it informs business-level decisions such as maintenance actions. Context thus accumulates across layers rather than existing as a single, fixed description. This accumulation is illustrated in Table I with an example of data from vibration sensor, which evolves from a plain acceleration value at the device level to a required action at the intent level.

TABLE I. ACCUMULATING CONTEXT

Layer	Data	Context Type	Meaning
Device	ax,ay, az	Physical	Acceleration values
Edge	RMS, FFT	Operational	Vibration level
Cloud	Anomaly score	Analytical	Equipment health
Application	Maintenance action	Business / intent	Recommended action

IV. CON^2 APPROACH

We adopt Gherkin and Behavior-Driven Development (BDD) as the foundation because they align with contemporary IoT practices and reflect the event-driven nature of such systems. Gherkin provides a lightweight, executable specification language in which requirements are expressed as Given–When–Then scenarios that function as both documentation and automated tests [3]. Unlike heavier modeling approaches that remain detached from implementation, Gherkin integrates naturally with Agile and DevOps workflows and supports continuous validation. Its executable scenarios allow requirements to evolve with the system and to be verified through testing or runtime monitoring, making it a practical basis for integrating explicit context and contract semantics.

However, while Gherkin captures discrete behaviors effectively, it does not explicitly represent the continuous and distributed contextual conditions typical of IoT systems. Our extension preserves standard syntax while introducing structured mechanisms to define and reuse contextual assumptions across scenarios. By separating context from behavior, the approach reduces duplication and enables conditional requirements to be specified under clearly defined operating conditions.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

A. *Intent-Driven Context Modeling*

The proposed context-aware requirements approach supports IoT development by making contextual knowledge explicit and structured. It begins with the systematic elicitation of operational intent and underlying assumptions so that critical dependencies do not remain implicit or fragmented across stakeholders. These are formalized as context contracts that describe the information and constraints associated with each architectural layer, reducing ambiguity and improving shared understanding. By linking contextual definitions to executable behavioral scenarios and runtime metrics, the approach establishes traceability from assumptions to observable system behavior and enables continuous validation as operating conditions evolve.

An Intent Context represents the operational purpose of the system from a stakeholder perspective. Rather than describing environmental conditions alone, it clarifies why the system operates and under which assumptions that purpose can be achieved. Intent defines which variables are relevant and what levels of performance or constraint are acceptable, thereby shaping how requirements are interpreted and how system behavior is evaluated.

Formally, we define an intent context C as the tuple

$$C = (I, S_r, S_e, Q, O, P)$$

where each element captures a distinct aspect of operational purpose and constraints. I denotes the intent statement, S_r denotes the set of required internal signals, S_e denotes the set of external inputs, Q denotes data quality constraints, O denotes operational or safety constraints, and P denotes optimization objectives or policies.

B. *Contracts*

Our approach draws inspiration from Design-by-Contract [9] to make assumptions and guarantees explicit and verifiable in IoT requirements. Specifications are structured around preconditions, postconditions, and invariants, which define the contextual assumptions under which behavior is valid, the outcomes the system must guarantee, and the properties that must always hold to ensure safety and correctness. In this framework, these elements are elevated from method-level assertions to system-level context contracts that define responsibilities across architectural layers. This contract-based view clarifies obligations across device, edge, and cloud components while enabling automated testing and runtime monitoring. By combining Design-by-Contract principles with executable behavioral scenarios, requirements become enforceable throughout development and operation rather than remaining purely descriptive.

Table II maps classical Design-by-Contract concepts to their counterparts in Con² and explains their role in IoT requirements engineering. Traditional elements are reinterpreted as explicit context assumptions, behavioral guarantees, and safety or quality constraints that govern when behavior is valid and what outcomes must be achieved. These constructs are organized into layered context contracts spanning architectural components to clarify responsibilities. Runtime assertion checking corresponds to continuous testing and monitoring, while contract refinement is reflected in context specialization for different operational intents.

TABLE II. CONTRACT CONCEPTS

Design-by-Contract concept	Traditional meaning	Corresponding element in Con²	Role in IoT requirements
Pre-conditions	Conditions that must hold before execution	Context assumptions / Preconditions in context contracts	Define when behavior or decisions are valid (e.g., data freshness, connectivity, calibration)
Post-conditions	Guarantees after execution	Then-clauses / Contract guarantees	Specify measurable outcomes and acceptance criteria
Invariants	Properties that always hold	Safety & quality constraints	Ensure continuous safety, reliability, and correctness
Contract	Formal agreement between caller and callee	Layered context contract (device/edge/ cloud)	Clarifies responsibilities between architectural layers
Assertion checking	Runtime verification of conditions	Tests & monitoring metrics	Continuous validation and runtime adaptation
Contract refine-ment	Stronger guarantees in subtypes	Context specialization per intent/layer	Different constraints under monitoring vs control vs optimization

A Context Contract defines the data and quality conditions associated with a given context by specifying the structure, semantics, and validity of the information on which a requirement depends. By making explicit which data must be available and trustworthy, it clarifies the assumptions underlying system behavior and addresses a common IoT issue in which data availability or quality constraints remain implicit.

A Contextual Scenario is a conventional Given–When–Then specification annotated with one or more intent contexts. The scenario inherits the context's assumptions and constraints, reducing repetition and strengthening traceability. Behavioral expectations are therefore interpreted only under explicitly defined operating conditions, which improves modularity and allows the same behavior to be evaluated consistently across alternative intents.

V. CONSTRUCTS AND USAGE

A. *Language Extensions*

Con² extends Gherkin with lightweight constructs that define reusable contexts and associate them with behavioral scenarios. Rather than altering the semantics of existing Feature or Scenario elements, the extension introduces additional structures that make contextual assumptions first-class and reusable. These separate the definition of operational intent and environmental constraints from behavioral descriptions, allowing requirements to be interpreted only under explicitly stated conditions.

The extension therefore augments, rather than replaces, conventional Given–When–Then scenarios. Context is defined once as structured, verifiable artifacts and then

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

referenced by scenarios that depend on it. This design reduces duplication, improves traceability between assumptions and behavior, and enables automated validation of both contextual conditions and functional outcomes. Together, the new constructs allow Gherkin specifications to capture not only what the system should do, but also under which contextual circumstances the behavior is valid.

A Context block declares reusable contextual assumptions and constraints that must hold for related scenarios to be valid. It encapsulates operational intent together with the required signals and quality conditions. The Intent field and contract-oriented sections - Preconditions, Invariants, Postconditions, and ExternalInputs - define contextual assumptions and guarantees outside of standard Given–When–Then scenarios.

The @context annotation associates a standard Gherkin Scenario with one or more previously defined contexts. The scenario inherits all assumptions and constraints from the referenced context. The following example illustrates the usage of these constructs.

```
Context: PredictiveMaintenance
  Intent: Early degradation detection with
minimal downtime

Preconditions:
- vibration_stream freshness <= 30s
- rpm_available OR rpm_quality >= 0.9
- calibration_status == valid
- sampling_rate_hz >= 5000

Invariants:
- alarm if rms_vibration_g >= 0.80 for >= 10s
- data_completeness >= 98% /day/asset
- units: acceleration=g, frequency=Hz
- anomaly_score uses approved_model_version

Postconditions:
- health_state every <= 5 min
- anomaly_score <= 10s per window
- maintenance_recommendation includes evidence

ExternalInputs:
- asset_registry
- maintenance_history
- operating_conditions
- spare_parts_status (optional)

@context(Monitoring)
Scenario: Excessive vibration alert
  Given vibration_rms > 12 mm/s
  When sustained for 5s
  Then maintenance alert generated
```

B. *Process Integration*

Con² integrates naturally into Agile/DevOps workflows without introducing heavy upfront modeling.

Step 1. Context elicitation. Stakeholders identify operational intents and derive corresponding contexts. For each intent, required signals, external inputs, and quality constraints are documented. This step makes implicit assumptions explicit and surfaces missing information sources.

Step 2. Scenario specification. Behavioral requirements are written as contextual scenarios. Scenarios focus on decision logic and observable outcomes rather than repeating environmental details.

Step 3. Validation and testing. Contexts are automatically checked for completeness, while scenarios are executed using BDD frameworks. Context contracts guide the creation of simulation data and monitoring rules.

Step 4. Runtime monitoring. Quality constraints defined in contexts (e.g., freshness or completeness) are monitored in production, enabling continuous validation of requirements satisfaction.

C. *Diverse stakeholder concerns*

By aligning context contracts with both architectural layers and stakeholder perspectives, the approach provides a structured way to elicit and document requirements from multiple viewpoints without losing coherence across the system. Each layer captures the goals that are meaningful to its respective stakeholders, allowing device-level concerns such as sensing accuracy or connectivity to be specified independently from operational processing constraints or business objectives. This separation enables stakeholders to articulate requirements using familiar terminology, without needing detailed knowledge of the entire technical stack. At the same time, explicit contracts connect these viewpoints through clearly defined dependencies, ensuring that higher-level expectations remain grounded in the conditions provided by lower layers. We believe that this reduces ambiguity and improves traceability of requirements. Furthermore, it systematically integrates diverse stakeholder concerns while maintaining consistency across the IoT architecture.

VI. Case Study: Smart Building Lighting Control

To demonstrate the applicability of Con² in a realistic setting, we apply it to a smart building lighting control scenario derived from our ongoing project, where IoT technology is used during construction and later to monitor living conditions. The case illustrates how contextual assumptions can be elicited, formalized as contracts, and linked to executable behavioral specifications. Although compact, it reflects key IoT characteristics such as heterogeneous devices, distributed control, multiple stakeholders, and context-dependent behavior.

The system provides voice-controlled indoor lighting. Users can switch individual lights or groups through spoken commands. The solution integrates voice assistants, home automation middleware, and KNX-based building automation, spanning device, edge, and application layers.

Three stakeholder roles interact with the system: residents, presenters, and maintainers. While sharing the same infrastructure, they differ in expectations regarding responsiveness and reliability, which highlights the need for explicit contextual assumptions.

Although switching lights appears straightforward, correct operation depends on contextual conditions such as user presence, connectivity, device availability, command resolution, and timely response. In the original project documentation, these aspects were captured in a traditional use case where assumptions remained partly implicit and scattered, making validation difficult.

We therefore elicit the operational purpose as an intent context, defined here as providing reliable, low-latency voice-based lighting control inside the building. The resulting

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

LightingControl context is specified using Con² contract elements to make these assumptions explicit..

```
Context: LightingControl
  Intent: Provide reliable and low-latency voice-
based lighting control inside

Preconditions:
  - user_location == "inside"
  - network.connected == true
  - voice_service.available == true
  - each_light.has_unique_identifier == true

Invariants:
  - response_time <= 1s
  - device_availability >= 99%
  - authorization_valid == true

Postconditions:
  - selected_lights.state == requested_state

ExternalInputs:
  - voice_command
  - device_registry
  - KNX network
```

These elements formalize requirements that would otherwise remain implicit. For example, response time is defined as a measurable invariant, while connectivity and authorization become explicit, verifiable assumptions. Structuring requirements in this way clarifies responsibilities across architectural layers, assigning device state, command routing, and authorization control to their respective components.

Behavior is then specified using Gherkin scenarios annotated with the defined context, keeping behavioral logic concise while automatically inheriting contextual assumptions. Additional scenarios describe alternative behaviors and error conditions.

```
@context(LightingControl)
Scenario: Turn on living room ceiling light
  Given the user says "Turn on the living room
ceiling light"
  When the command is recognized
  Then the living room ceiling light shall be ON

@context(LightingControl)
Scenario: Unknown light name
  Given the user says "Turn on the red hallway lamp"
  When the device cannot be resolved
  Then an error message shall be provided

@context(LightingControl)
Scenario: Device unavailable
  Given the requested light is unreachable
  When a control command is issued
  Then the system shall report a failure
```

These scenarios focus on observable behavior, while contextual assumptions such as connectivity and latency are inherited from the context contract. This separation reduces duplication and improves maintainability, since shared assumptions are defined once. Because the scenarios follow the BDD style, they can be executed as automated tests. During development, simulated device states and voice commands verify postconditions, and in operation the same contract elements are monitored at runtime. Response time, device availability, connectivity, and command success rates are continuously tracked, allowing invariant violations to trigger alerts or fallback behavior. Requirements thus evolve from static documentation into enforceable runtime checks, supporting continuous validation within DevOps workflows.

Applying Con² to the lighting scenario demonstrates practical benefits. Contextual assumptions become explicit and testable, separation of context and behavior reduces repetition, layered contracts clarify responsibilities, and executable specifications enable automatic validation of both functional and quality requirements. Even in this compact example, the dependency of behavior on contextual factors becomes evident, reinforcing the value of making context explicit to improve clarity and maintainability.

VII. EVALUATION

We evaluate the proposed Con² approach using criteria relevant to IoT requirements engineering: explicitness of assumptions, traceability between context and behavior, and executability. Rather than conducting a large-scale empirical study, the goal is to analytically assess whether the approach improves clarity, structure, and verifiability compared with the conventional format used for the smart building use case.

The comparison contrasts two representations of the same functionality, Lighting-1: the original textual use case and the Con²-based specification built from intent contexts, contracts, and contextual scenarios. Although both describe voice-controlled lighting, they differ in how contextual knowledge is represented and validated. The analysis examines which assumptions are implicit or explicit, and how clearly responsibilities are assigned across architectural layers.

In the traditional use case, preconditions, postconditions, and interaction steps are presented narratively, leaving many contextual aspects informal. Network availability, device readiness, response time, and authorization appear as textual assumptions without structured support for validation or reuse, which increases the risk of being overlooked. In contrast, the Con² specification separates context from behavior by formalizing assumptions and quality constraints as contracts while scenarios describe observable outcomes. This structure enables reuse across scenarios and supports automated testing and runtime monitoring, turning contextual statements into verifiable system properties.

Table III summarizes the differences using concrete elements drawn directly from the original use case. The comparison suggests three main benefits. First, Con² improves explicitness by converting informal assumptions into structured, contract-based elements. Second, it enhances traceability by linking contextual conditions directly to executable scenarios. Third, it increases verifiability by enabling automated testing and runtime monitoring of both functional and non-functional requirements.

VIII. DISCUSSION

Con² augments behavioral specification with contextual semantics. With structured yet lightweight constructs for context and contracts, it enhances explicitness without imposing substantial overhead. Improved explicitness is the central benefit observed in the case study, which also improves traceability. Each contextual scenario links high-level intent to an executable behavior, making it possible to reason systematically from stakeholder goals to concrete tests.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

In IoT projects, requirements frequently emerge only after deployment and operational feedback. Contexts and scenarios can evolve alongside the system, enabling gradual refinement rather than heavy upfront modeling. Traditional requirements techniques often assume stable environments and may struggle to capture the dynamic behavior of IoT systems.

TABLE III. TABLE TYPE STYLES

Aspect from the original use case	Conventional use case representation	Con² representation	Practical effect
User must be inside building	Informal precondition text	Explicit precondition in context contract	Can be validated using location sensing
Network must be operational	Mentioned as assumption	Context invariant (connectivity == true)	Enables automatic monitoring and alerts
Devices must be ready	Narrative statement	Device availability invariant	Clear responsibility for device layer
Unique light identifiers	Implicit design constraint	Required internal signal	Improves device registry traceability
Response time $\leq$ 1 s	Non-functional note	Measurable quality constraint	Testable in CI and runtime
Authorization required	Textual rule	Operational constraint	Enforceable and auditable
Voice command behavior	Step-by-step scenario	Gherkin contextual scenario	Executable automated test
Error handling	Listed exceptions	Separate contextual scenarios	Testable failure paths
Shared assumptions across scenarios	Repeated or implied	Defined once in context block	Reduces duplication
Responsibility across layers	Not explicit	Layered context contracts	Clear device /edge/cloud roles

Although demonstrated using a smart building scenario, the principles underlying Con² are not domain specific. Any IoT system where behavior depends on environmental or operational conditions may have advantage from explicit context modeling. However, for small or static systems, the potential modeling overhead may not be justified. The benefits become more pronounced as the system complexity, heterogeneity, or stakeholder diversity increases.

The current evaluation remains qualitative and limited to a single illustrative real-world case, which limits the ability to generalize the findings across diverse domains and project scales. Furthermore, the assessment of clarity and traceability relies on analytical comparison rather than strictly controlled empirical measurements, which may introduce subjective interpretation. Broader empirical studies are required to assess usability of the approach, and measurable impacts on clarity, traceability, and validation of requirements in IoT.

IX. CONCLUSION AND FUTURE WORK

IoT systems operate in dynamic and context-dependent environments where behavior depends not only on functional logic but also on implicit environmental and operational assumptions. This paper introduced Con², a lightweight context-aware requirements engineering approach that makes such assumptions explicit through intent-driven context contracts and executable behavioral scenarios. By integrating contextual modeling with Gherkin-based specifications, Con² improves explicitness while remaining compatible with Agile and DevOps practices. The exemplar case study indicates that even simple IoT application scenarios benefit from separation of concerns, supporting the argument that context should be treated as a first-class element in IoT requirements engineering. Future work will focus on empirical evaluation and tool support in larger industrial deployments.

REFERENCES

[1] J.-A. Aguilar-Calderón, C. Tripp-Barba, A. Zaldívar-Colado, and P.-A. Aguilar-Calderón, "Requirements engineering for Internet of Things (IoT) software systems development: A systematic mapping study," IEEE Access, vol. 12, no. 15, p. 7582, 2022.

[2] P. Boutot, M. R. Tabassum, A. Abedin, and S. Mustafiz, "Requirements development for IoT systems with UCM4IoT," Journal of Computer Languages, vol. 78, p. 101251, 2024.

[3] Cucumber Ltd., "Gherkin Reference Documentation," Accessed: Feb. 14, 2026. [Online]. Available: https://cucumber.io/docs/gherkin/.

[4] D. V. da Silva, B. P. de Souza, T. G. Gonçalves, and G. H. Travassos, "A requirements engineering technology for IoT software systems," arXiv, 2021.

[5] L. da Silva Souza, F. B. Ayres, P. H. T. Costa, and E. D. Canedo, "Requirements engineering processes in the context of IoT and requirements validation techniques," in Proc. WER, 2022.

[6] A. K. Dey, "Understanding and using context," Personal and Ubiquitous Computing, vol. 5, no. 1, pp. 4–7, 2001.

[7] A. Kamilaris, A. Kartakoullis, and F. X. Prenafeta-Boldú, "A review on the practice of big data analysis in agriculture," Computers and Electronics in Agriculture, vol. 143, pp. 23–37, 2017.

[8] U. Kannengiesser, F. Krenn, and C. Stary, "A behaviour-driven development approach for cyber-physical production systems," in Proc. 2020 IEEE Conf. on Industrial Cyberphysical Systems (ICPS), 2020, pp. 179–184.

[9] B. Meyer, "Design by Contract," in Advances in Object-Oriented Software Engineering, Prentice Hall, 1991, pp. 1–50.

[10] S. Mouhim and F. Lachhab, "Towards a context awareness system using IoT, AI, and big data technologies," IEEE Access, vol. 13, pp. 40302–40315, 2025.

[11] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the Internet of Things: A survey," IEEE Communications Surveys & Tutorials, vol. 16, no. 1, pp. 414–454, 2014.

[12] E. Siakas et al., "REFIoT: A framework to combat requirements engineering in IoT applications and systems," in Systems, Software and Services Process Improvement, vol. 2179, Springer, 2024, pp. 80–96.

[13] M. Simpson et al. "A platform for the analysis of qualitative and quantitative data about the built environment and its users," in Proc. 2017 IEEE 13th Int. Conf. on e-Science, 2017, pp. 228–237.

[14] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in Proc. 37th EUROMICRO Conf. on Software Engineering and Advanced Applications (SEAA), Aug. 2011, pp. 383–387.

[15] B.-Y. Wang, Y.-C. Yen, and Y.-C. Cheng, "Specifying Internet of Things behaviors in behavior-driven development: Concurrency enhancement and tool support," Applied Sciences, vol. 13, no. 2, p. 787, 2023.

[16] N. Zubair, N. A., K. Hebbar, and Y. Simmhan, "Characterizing IoT data and its quality for use," arXiv, 2019.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Fine-tuned Random Forest-based Software Defect Prediction via Metaheuristics

Tadashi Dohi, Jingchi Wu, Junjun Zheng and Hiroyuki Okamura

Graduate School of Advanced Science and Engineering, Hiroshima University
Higashi-Hiroshima 739-8527, Japan
email: {dohi, kure, jzheng, okamu}@hiroshima-u.ac.jp

***Abstract*—Software defect prediction is one of the most fundamental issues to identify defect-prone modules in module testing, and enables to reduce effectively the module testing cost. While several machine learning techniques are applied to the software defect prediction, it is well known that ensemble methods such as random forest are quite useful to detect the defect-prone modules with high predictive performance and low computation cost. Since the Random Forest (RF) contains the hyperparameters, which have to be carefully tuned in advance, the prediction accuracy on the defect-proneness strongly depends on the tuning results in the random forest. In this paper, we propose to apply three metaheuristic algorithms: Latin Hypercube Sampling (LHS) algorithm, Artificial Bee Colony (ABC) algorithm, and Parameter free Genetic Algorithm (PfGA), to search the optimal hyperparameters in the RF-based software defect prediction, and investigate the predictive performance of defect-prone modules. In experiments with six actual software development projects, we compare our fine-tuned RF algorithms with the existing machine learning approaches. It is empirically shown that the fine tuning via metaheuristics could provide better predictive performances on F-score in software defect prediction.**

Keywords-software defect prediction; defect-prone module; random forest; hyperparameters; metaheuristics; fine tuning; search-based software engineering.

I. INTRODUCTION

In almost all software development projects, it is necessary to spend a large amount of testing time and cost, because software systems are implicitly expected to be defect-free after releasing to the users or market. On one hand, it is known that the requirement of defect-free software is rather stringent and cannot be achieved in almost all the development projects, since the software testing period is limited but the testing cost is rather expensive. Hence, the practical and ad hoc approach would be to shorten the testing period by improving the efficiency of software test. Many experiences suggest that software defects should be fixed timely and exactly from the defect-prone modules, if they can be identified. If sufficient testing resources, including testing personnel for review and highly advanced test case generation algorithms etc., can be allocated to the software module testing, it may be possible to detect/fix an enough number of defects even in the module testing level.

However, the defect-prone modules in large-scaled software systems with a number of modules tend to be sparse, testing/reviewing all the modules is not cost-effective. *Software defect prediction* is one of the most fundamental issues to identify the defect-prone modules before the review/test, and enables to reduce effectively the module testing cost. It can be defined as a *statistical discriminant problem* and be characterized by *software defect-prone module probability*, which is defined as the probability that a module contains at least one software defect. More specifically, suppose that the binary data that each module was defect-prone (1) or defect-free (0) are available as training data, and that feature data, called *software metrics*, involving the module size, complexity and development effort, are observed for all the modules. Based on the training data and the feature data, one predicts the defect-prone module probabilities for the remaining modules that have not been reviewed yet, and judges whether they are defect-prone or not.

During almost the last four decades, several machine learning techniques were applied to the software defect prediction. In early phase, standard discriminate analysis techniques such as logistic regressions, multi-layer perceptron neural networks, naive Bayes, etc. have been utilized for the software defect prediction. There are classified into two approaches: deep neural network approach [1] and ensemble approach [29]. In the former approach, the deep neural networks [2] [3] [26] are commonly used techniques to relate software metrics data to the binary discriminant problem. The latter approach is based on boosting algorithms, such as XGBoost [8] and AdaBoost [12], and bagging algorithms [5]. Laradji et al. [17] combined feature selection and ensemble learning on the performance of defect classification, where a new two-variant ensemble learning method is proposed with and without feature selection. Tong et al. [23] applied stacked denoising autoencoders for feature learning in software defect prediction, and combined the deep learning with two-stage ensemble. Riaz et al. [22] developed a novel method by combining rough set theory, K-nearest neighbor rule and noise-filter technique to deal with imbalanced classes for software defect prediction.

In this paper, we focus on random forest (RF) -based software defect prediction. RF proposed by Breiman [6] is a representative ensemble method by extending the well-known decision tree algorithms, and can be also used in software defect prediction with high predictive performance and low computation cost. Similar to the other machine learning methods, RF contains hyperparameters, which have to be carefully tuned in advance [21]. Malhotra and Khanna [20] used four strategies of ensemble learning to predict change prone classes by combining seven individual Particle Swarm Optimization (PSO)-based classifiers as constituents of ensembles and aggregating them using weighted voting. Turabieh et al. [25] employed three metaheuristics: Binary Genetic Algorithm

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

(BGA), binary PSO, and Binary Ant Colony Optimization (BACO), for feature selection problem in software defect prediction. Alsghaier and Akour [4] proposed an approach by integrating GA with support vector machine classifier and PSO for software defect prediction. Zhang et al. [28] considered a novel defect prediction model based on stacked sparse denoising autoencoders in [23], extreme deep learning machine optimized by PSO and another complementary gravitational search algorithm. Khan et al. [14] investigated software defect prediction models with seven machine learning classifiers in conjunction with the artificial immune network by optimizing their hyperparameters. Recently, Thomas and Kaliraj [24] used a fine-tuned RF classifier to optimize hyperparameters in addition to applying an oversampling technique, and enhanced predictive accuracy in software defect prediction.

We consider RF-based software defect prediction, which differs from [14] and [24] in that the hyperparameters of RF are tuned by multiple metaheuristic algorithms. Note that the original RF by Breiman [6] consists of the depth of trees, the number of trees and the number of features, which are regarded as hyperparameters. The commonly used technique to determine the hyperparameters is based on the preliminary experiments to tune them. In fact, several ensemble methods involving RF are available in free tools, where the default values of the hyperparameters are set up. It is worth mentioning that these default values must be tuned carefully or optimized, if possible, for specific problems, but some authors seem to use the default values without doubt in many cases. The most well-known technique to determine the hyperparameters in RF would be grid search [21]. However, the grid search is also a heuristic algorithm, and does not guarantee the globally optimal hyperparameters.

The main challenge of this paper is to apply three metaheuristic algorithms: Latin Hypercube Sampling (LHS) algorithm, Artificial Bee Colony (ABC) algorithm, and Parameter free Genetic Algorithm (PfGA), to search the optimal hyperparameters in the RF-based software defect prediction, LHS algorithm, which was developed in Los Alamos National Laboratory [19], is a classical heuristic method for generating a near-random sample of parameter values from a multidimensional distribution. ABC algorithm is an optimization algorithm inspired by the intelligent foraging behavior of honey bee swarm [13]. Kondo and Asanuma [15] applied the ABC algorithm to optimize the hyperparameters in RF. PfGA is an intuitive but most fundamental technique to determine the hyperparameters in ensemble methods. We tune the RF-based software defect prediction models by the above three metaheuristics, and compare them with the grid search approach and the other boosting-based software defect prediction models, such as AdaBoost [12], XGBoost [8] and Light Gradient Boosting Machine (GBM) developed by Microsoft [16].

The remaining part of this paper is organized as follows. In Section II, we formulate the software defect prediction as a statistical discriminant analysis. Section III overviews the RF. Section IV introduces three metaheuristic algorithms: LHS, ABC and PfGA. The experiments with six NASA data sets: CM1, KC1, KC3, MW1, MC1 and MC2, are presented in Section V, where we compare our fine-tuned RF-based software defect prediction models with the common grid search as well as the existing boosting models: AdaBoost, XGBoost, Light GBM, and common Multi-Layer Perceptron (MLP) neural network. It is empirically shown that the fine tuning via metaheuristics could provide better predictive performances in terms of F-score.

II. SOFTWARE DEFECT PREDICTION: FORMULATION

In software development projects, detecting and fixing software defects in limited testing resources are rather expensive, but play a significant role to assure software reliability. There are two major problems to make the software test effective: software defect localization and software defect prediction. The purpose of software defect localization is to identify the defect position on a software program. The software defect prediction focuses on the identification of defect-prone modules before reviewing/testing software program in the module testing. Identification of software defect-prone modules enables us to carry out software module test effectively.

Suppose that a software system under the module test consists of N modules. Let $\boldsymbol{x}_i = (x_{i1}, x_{i2}, \dots, x_{in})$ denote the feature vector of the $i\ (= 1, 2, \dots, N)$-th software module, where n types of features, called *software metrics*, are available in the coding phase. Define the binary random variable:

$$Y_i = \begin{cases} 1, & \text{if } i-\text{th-module contains software bugs,} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Let $P_i(\boldsymbol{x}_i) = \mathrm{P}\{Y_i = 1 \mid \boldsymbol{x}_i\}$ be the defect-prone module probability which denotes a probability that the i-th module contains any software bug, conditioned that the software metrics $\boldsymbol{x}_i$ are given. Since Y_i is a Bernoulli random variable for fixed i, it is evident that $\mathrm{E}[Y_i] = P_i(\boldsymbol{x}_i)$.

For the simplest example, consider the logistic regression model:

$$P_i(\boldsymbol{x}_i) = P_i(\boldsymbol{x}_i \mid \boldsymbol{\beta}, \beta_0) = \frac{\exp(\boldsymbol{z}_i)}{1 + \exp(\boldsymbol{z}_i)}, \tag{2}$$

where $\exp(\boldsymbol{z}_i) = \boldsymbol{\beta}^T \boldsymbol{x}_i + \beta_0$ denotes the random variables representing the tendency of bug presence, $\boldsymbol{\beta} = (\beta_1, \beta_2, \dots, \beta_n)$ is the regression coefficient vector, β_0 is a scalar constant, and T is the transpose. Since the defect-prone module probability $P_i(\boldsymbol{x}_i)$ is given by a function of $\boldsymbol{x}_i$, it turns out that the dependent variable becomes a monotonic function with respect to each module with $\boldsymbol{x}_i$. Define the parameter vector $\boldsymbol{\theta} = (\boldsymbol{\beta}, \beta_0)$. For realizations y_i of the binary random variable Y_i with feature vector $\boldsymbol{x}_i$, the log likelihood function is given by

$$\begin{aligned} \ln L(\boldsymbol{\theta}) \quad = \quad & \sum_{i=1}^{N} \Big\{ y_i \ln P_i(\boldsymbol{x}_i \mid \boldsymbol{\beta}, \beta_0) \\ & + (1 - y_i) \ln[1 - P_i(\boldsymbol{x}_i \mid \boldsymbol{\beta}, \beta_0)] \Big\}. \end{aligned} \tag{3}$$

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

By maximizing $\ln L(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$, we get the maximum likelihood estimate of the parameter $\boldsymbol{\theta} = (\boldsymbol{\beta}, \beta_0)$, and predict the defect-prone module probabilities for the remaining modules that have not been reviewed. Recently, Dohi et al. [10] generalized the classical software defect prediction by introducing the semi-definite logistic regression. From the above example, it can be easily seen that software defect prediction is reduced to a statistical discriminant problem and that a number of machine learning algorithms containing deep neural network and ensemble methods can be applied.

We refer to a static software defect prediction. Suppose that there are $N = k + m$ software modules, where k modules were already reviewed and identified whether they were defect-prone or defect-free. Let $(y_i, \boldsymbol{x}_i)$ $(i = 1, 2, \ldots, k)$ be the training data. Based on the training data, we estimate $P_j(\boldsymbol{x}_j)$ $(j = k + 1, k + 2, \ldots, k + m)$, where $(y_j, \boldsymbol{x}_j)$ are the validation data. Let ξ $(0 < \xi < 1)$ denote the discriminant threshold, which is a cut off value to identify whether the j-th module is defect-prone or not. If $P_j(\boldsymbol{x}_j)$ is greater than ξ, then we identify that j-th module is defect-prone, otherwise, defect-free. In practice, only defect-prone modules will be reviewed in the module test. To evaluate the predictive performances of the machine learning models, we compare our prediction results for m modules with the validation data y_j $(j = k + 1, k + 2, \ldots, k + m)$ and obtain the confusion matrix which consists of True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN). TP and FP imply the respective numbers of defect-prone modules predicted correctly and incorrectly, TN and FN mean the numbers of defect-free modules predicted correctly and incorrectly. Define the following three prediction metrics:

$$\mathrm{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}, \quad (4)$$

$$\mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \quad (5)$$

$$\mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}. \quad (6)$$

Finally, by taking the harmonic mean of Precision and Recall, we obtain the F-score:

$$\mathrm{F-score} = \frac{2 \times \mathrm{Precision} \times \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}. \quad (7)$$

The larger F-score implies the higher predictive performance in software defect prediction. It should be noted that F-score is not a unique prediction measure, because it also has some limitations [9] [18] [27]. Nevertheless, we focus on effects of fine-tuned RF in terms of the resulting F-scores, because it is still useful for general discriminant problems but not for only software bug prediction.

III. RANDOM FORESTS

RF [6] is a representative of the state-of-the-art ensemble methods and a lightweight machine learning technique in terms of computation cost. It is a substantial extension of bagging (bootstrap aggregating) [5], where the major difference from bagging is to incorporate the randomized feature selection. For better understanding, we quickly overview the RF. Consider a decision tree, which is an algorithm that searches for the most matching leaves by tracing conditional branches from a root. During the construction of a component decision tree, at each step of split selection, RF randomly selects a subset of features, and then carries out the conventional split selection procedure within a selected feature subset. The combination of features and threshold that could reduce the impurity within each node is selected using the entropy. For the current node s, define the entropy:

$$H(s) = -\sum_{c \in \{0,1\}} p(c|s) \log p(c|s), \quad (8)$$

where

$$\Delta I(s) = H(s_a) - \sum_{i \in \{L,R\}} \frac{n_i}{k} H_i(s_b) \quad (9)$$

is the information gain, $p(c|s) = n_c/k$, c is the class of the objective variable, k is the number of training data, n_i is the number of data belonging to the class, s_a is the node before the branch, s_b is the node after the branch, $i = L$ and $i = R$ are the left and right nodes after the branch, respectively. The above equations are used to split the data and to find the optimal splitting condition at each node. We decide whether the end condition is satisfied or not, so if the end condition was satisfied, then we integrate the results. This procedure is repeated recursively until a specified number of times to create multiple decision trees is achieved. For software defect prediction, the results of each decision tree are integrated by means of a majority voting.

In our classification problem, the final prediction value becomes the output, so the class of the objective variable is set to 1 if it contains at least one software defect, otherwise, set to 0. The posterior probability $p(c|s)$ in (8) can be expressed as the total number of software modules containing defects in the leaf node at that time. The combination of thresholds that minimize the posterior probabilities is selected based on the entropy in (8) and the information gain in (9).

Hence, it is obvious to see that we encounter problems to determine the number of trees and the depth of each tree, and a problem which feature should be applied for the discriminant analysis. Note that the parameter of features controls to incorporate the randomness. If all the numbers of features equal, then RF is reduced to a deterministic decision tree. If the parameter of features is given by 1, then one feature is selected randomly. It is known that the depth of the decision tree used in a weak learner is a significant parameter in terms of prediction accuracy. To this end, the predictive performance in RF varies significantly depending on the depth of trees. Also, since RF creates a specified number of trees, the final output by a majority voting with all the trees, and the resulting prediction accuracy depends on the predictive performance. Unfortunately, almost all RF tools request to tune the number of trees, the depth of each tree and the number

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

of features in advance. In the following section, we introduce three metaheuristics to determine these hyperparameters.

IV. Metaheuristic Algorithms

In this section, we summarize three metaheuristic algorithms employed in the paper.

A. Latin Hypercube Sampling Algorithm

LHS is a method for sampling the search space uniformly within a specified number of iterations [19]. First, we specify the number of iterations α, and divide the search space into α intervals for each feature. We draw one sample randomly from each interval, and take place an experiment by combining it with other features. For RF, we use the above LHS algorithm to find a combination of tree depths, number of trees, and number of features. It is assumed in our study that each depth ranges in 2∼50, the number of trees in 10∼1000, and the number of features in 1∼38. We repeat these random selections 1000 times, and prepare 1000 sets of hyperparameters. Let ϕ be a uniformly distributed pseudo random number. For the hyperparameters $\boldsymbol{h} = (h_1, \ldots, h_{1000})$, we find the maximum value, $\max(\boldsymbol{h})$ and the minimum value, $\min(\boldsymbol{h})$. Then the LHS-based hyperparameters are given by $\phi\{\max(\boldsymbol{h}) - \min(\boldsymbol{h})\} + \min(\boldsymbol{h})$.

B. ABC Algorithm

The ABC algorithm is an optimization algorithm that imitates the behavior of honey bee swarm and is divided into three phases [13]. The first phase called the *harvesting bee phase* is to update each candidate solution. In the second phase called the *bee phase*, the updated candidate solution is selected probabilistically in accordance with the degree of adaptation of each candidate solution. In the final *reconnaissance bee phase*, the candidate solutions that have not been updated within a pre-determined number of times are randomly generated, and are replaced by new ones. In updating the solution, the (i, j)-element of the candidate solution, denoted by i-th row and j-th column vector, is given by $\boldsymbol{v}_{i,j} = \boldsymbol{x}_{i,j} + \phi(\boldsymbol{x}_{i,j} - \boldsymbol{x}_{k,j})$. Then we compare with the newly generated candidate solution and replace it by the one with the higher adaptivity $1/\{1 + f(\boldsymbol{x}_i)\}$ with any adaptive function $f(\cdot)$, where ϕ is a uniformly distributed pseudo random number in the interval $[0, 1]$. Following Kondo and Asanuma [15], we use the ABC algorithm to find a combination of tree depth, number of trees, and number of features. Set 2∼50 for the depth, 10∼1000 for the number of trees, and 1∼38 for the number of features as well. Next, we prepare 1000 candidates of hyperparameters. Substituting these hyperparameters into $x_{i,j}$ yields the updated hyperparameters.

C. Parameter Free GA

PfGA is a genetic algorithm that eliminates the need to select parameters with crossover rate, mutation rate and crossover method. In the crossover step, two individuals are randomly selected from the local population to perform a multipoint crossover with random number and random location. Next, two individuals are randomly selected from the local population and are mutated at random numbers and positions. In the selection, based on the evaluation points of these four individuals of the family, we perform the selection procedure and return the operation to the local population. If both the offspring are better than the two parents, then we form a local population from the two offspring and the better off parent. If the two offspring are worse than the two parents, then we form a local population with the better parent. If only one of the two parents is better than the two children, then the better parent and the better child form a local population. If only one of the two offspring is better than the two parental offspring, then a local population is formed with the better offspring and a randomly selected individual from the entire search space. In our software defect prediction problem, each parameter is used as genetic information to perform the operation. For the hyperparameters in RF, we prepare two random combinations of parameters and put them into the above algorithm.

TABLE I. DATA SETS.

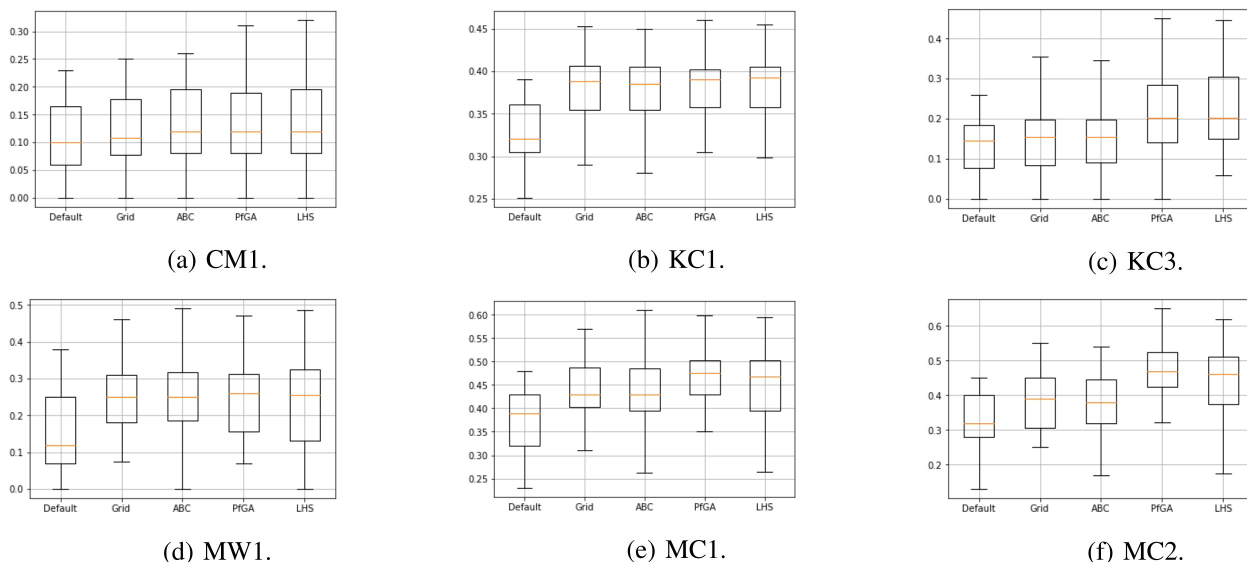
	No. modules	Defect proneness (%)	No. metrics
CM1	327	12.80	38
KC1	2108	9.96	38
KC3	458	9.17	38
MW1	403	7.70	38
MC1	9466	7.18	38
MC2	161	32.20	38

V. Experiments

The well-known benchmark data sets for software defect prediction come from NASA MPD. In Table 1, we present six software programs: CM1, KC1, KC3, MW1, MC1, MC2 with each 37 software metrics. In our experiments, we compare the predictive performances of RFs tuned by grid-search (Grid), LHS, ABC and PfGA with the RF with default hyperparameters (Default). We also compare our refined RFs with four conventional machine learning techniques: MLP with three layers, Adaboost, XGBoost, and LightGBM. Through experiments, we set the discriminant threshold as $\xi = 0.5$. Although the argument to control the discriminant threshold itself exists, we emphasize that the software defect-prone module probability is symmetric to justify our assumption. It is also well-known that the predictive performance in software defect prediction strongly depends on the sampled module, where k modules are sampled as the training data. In order to reduce the prediction bias, we applied the holdout cross-validation and selected randomly 50% of the total number of modules 100 times in each data set.

In Figure 1, we give the box plots of F-scores predicted by RFs and compare the predictive performances of RFs with metaheuristics (Grid, LHS, ABC, PfGA). In all data sets, it is seen that the metaheuristic approaches worked significantly better than the default case. So, we could observe that the tuning of hyperparameters is effective to improve the predictive performance in software defect prediction. Comparing

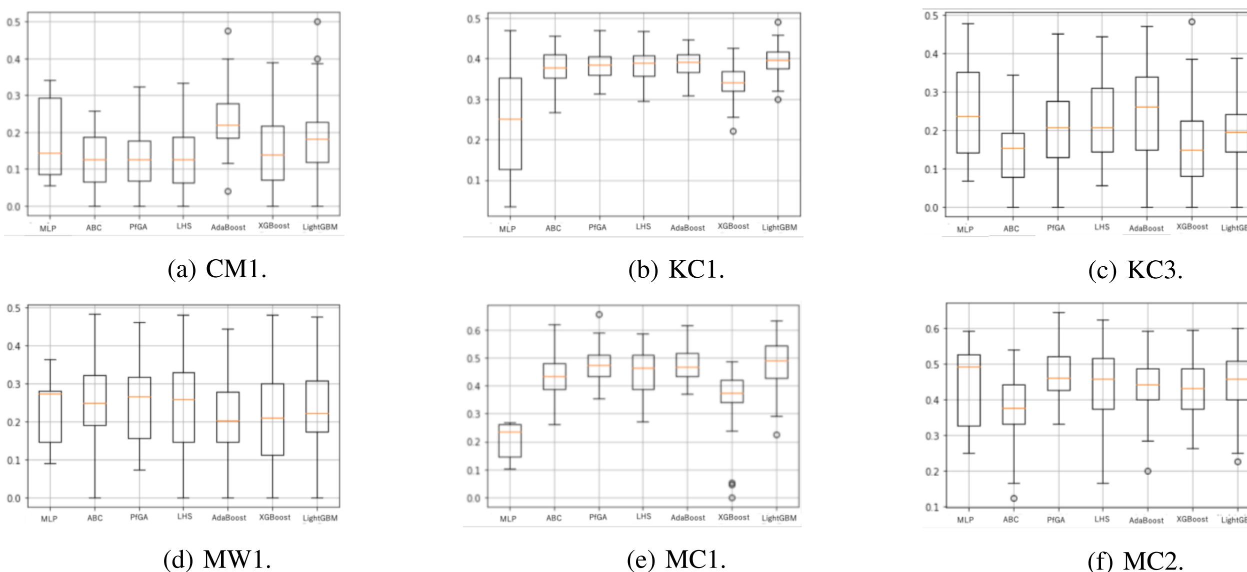
Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org



(a) CM1. (b) KC1. (c) KC3.

(d) MW1. (e) MC1. (f) MC2.

Figure 1: Comparison of RFs with four metaheuristics in terms of F-score.



(a) CM1. (b) KC1. (c) KC3.

(d) MW1. (e) MC1. (f) MC2.

Figure 2: Comparison with other machine learning algorithms in terms of F-score.

with the grid search-based RF, it can be seen that ABC could not show the larger median values than Grid in KC1, KC3, MW1, MC1 and MC2. However, we find that LHS and PfGA provided the larger median values than Grid in all data sets. In the comparison between LHS and PfGA, we can see that PfGA gave slightly better prediction results than LHS. From these results, we can conclude that the refinement with metaheuristics in RFs could improve the F-scores in software defect prediction, and could outperform the baseline RF with default hyperparameters. However, it should be noted that the lengths from lower quartile to upper quartile in LHS and ABC are longer than those in Default and Grid in CM1, MW1, MC1 and MC2. This implies that the tuning of hyperparameters with LHS and ABC tends to show the optimistic prediction with higher median and variance.

Next we concern the comparison of our fine-tuned RFs with the other boosting algorithms (Adaboost, XGBoost, LightBGM) and the classical MLP. Figure 2 gives the box plots in terms of the F-scores with seven machine learning techniques. In CM1 and KC3, AdaBoost gave the highest F-scores among others. In KC1 and MC1, LightGBM was the best predictor. In MC2 and MW1, MLP could provide the highest F-scores. In this way, as seen in the existing works, the machine learning technique with the highest predictive performance depends on the kind of data sets. However, it should be noted that our refined RFs could be the second and/or third best predictors

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

in many cases except CM1 and KC3, and could provide rather stable prediction results. Especially, it can be shown that the variance of PfGA is smaller than that of LHS expect in KC3. From these results, we agree that more recent boosting algorithms, such as XGBoost and LightBGM tended to give better prediction results, but conclude that the refinement of hyperparameters in the classical RF could improve the predictive performances effectively.

VI. Conclusion and Future Work

In this paper, we have improved the RFt-based software defect prediction by applying three metaheuristic algorithms; LHS algorithm, ABC and PfGA, to search the optimal hyperparameters. Through experiments with six NASA MPD data sets, we have investigated the predictive performances of software defect-prone modules, and compared our refined random forests with the default RF approach as well as the existing machine learning approaches. It has been shown that the RF with metaheuristics could provide stable prediction results in many cases.

In the future, we will apply the other metaheuristics, such as PSO, BGA, binary PSO, and BACO, for further potential improvement of RF-based software defect prediction. Also, effects of fine tuning should be investigated in different prediction measures from F-score [9] [18] [27]. Another direction is to improve the predictive performance by tuning the hyperparameters and preprocessing the imbalanced classes in software modules. In general, the number of defect-prone modules is sparse in all the software modules. For such imbalanced data, almost all machine learning models cannot work to guarantee satisfactory F-scores. Then, the so-called oversampling algorithms will be used to generate the training data, where SMOTE [7] and ADASYN [11] are the most well-known algorithms. Actually, Riaz et al. [22] improved the software defect prediction by combining the machine learning algorithm with the oversampling algorithm. We will also study the efficiency by combining the fine-tuning of the hyperparameters with the oversampling algorithm in the forthcoming article.

References

[1] C. C. Aggarwal, *Neural Networks and Deep Learning: A Textbook*, Springer, 2023.

[2] Q. Al-Qasem, M. Akour, and M. Alenezi, “The influence of deep learning algorithms factors in software fault prediction”, *IEEE Access*, vol. 8. pp. 63945–63960, 2020.

[3] E. N. Akimova et al., “A survey on software defect prediction using deep learning”, *MDPI Mathematics*, vol. 9, article no. p. e1180, 2021.

[4] H. Alsghaier and M. Akour, “Software fault prediction using particle swarm algorithm with genetic algorithm and support vector machine classifier”, *Software: Practice and Experience*, vol. 50, no. 4, pp. 407–427, 2020.

[5] L. Breiman, “Bagging predictors”, *Machine Learning*, vol. 24, pp. 123–140, 1996.

[6] L. Breiman, “Random forests”, *Machine Learning*, vol. 45, pp. 5–32, 2001.

[7] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: synthetic minority over-sampling technique”, *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[8] T. Chen and C. Guestrin, “XGBoost: A scalable tree boosting system”, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2016)*, pp. 785–794, ACM ICPS, 2016.

[9] D. Chicco and G. Jurman, “The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation”, *BMC Genomics*, vol. 21, no. 6, pp. 1-–13, 2020.

[10] T. Dohi, J. Wu, and H. Okamura, “Software bug prediction based on semi-definite logistic regression model”, *Proceedings of The 10th International Conference on Advances and Trends in Software Engineering (SOFTENG 2024)*, pp. 11–16, IARIA Press, 2024.

[11] H. He, Y. Bai, E. A. Garcia, and S. Li, “ADASYN: Adaptive synthetic sampling approach for imbalanced learning”, *Proceedings of 2008 IEEE International Joint Conference on Neural Networks*, pp. 1322–1328, IEEE CPS, 2008.

[12] Y. Freund and R. E. Schapire, “A decision-theoretic generalization of on-line learning and an application to boosting”, *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[13] D. Karaboga,“ An idea based on honey bee swarm for numerical optimization”, *Technical Report-TR06*, Department of Computer Engineering, Engineering Faculty, Erciyes University, 2005.

[14] F. Khan, S. Kanwal, S. Alamri, and B. Mumtaz, “Hyper-parameter optimization of classifiers, using an artificial immune network and its application to software bug prediction”, *IEEE Access*, vol. 8, pp. 20954–20964, 2020.

[15] H. Kondo and Y. Asanuma, “Optimization of hyperparameters and feature selection for random forests and support vector machines by artificial bee colony algorithm (in Japanese)”, *Journal of the Japanese Society for Artificial Intelligence*, vol. 34, no. 2, pp. G-136: 1–11, 2019.

[16] L. Kopitar, P. Kocbek, L. Cilar, A. Sheikh, and G. Stiglic, “Early detection of type 2 diabetes mellitus using machine learning-based prediction models”, *Scientific Reports*, vol. 10, p. e11981, 2020.

[17] I. H. Laradji, M. Alshayeb, and L. Ghouti, “Software defect prediction using ensemble learning on selected features”, *Information and Software Technology*, vol. 58, pp. 388—402, 2015.

[18] L. Lavazza and S. Morasca, “Comparing ϕ and the F-measure as performance metrics for software-related classifications, *Empirical Software Engineering*, vol. 27, no. 185, 2022.

[19] M. D. McKay, R. J. Beckman, and W. J. Conover, “A comparison of three methods for selecting values of input variables in the analysis of output from a computer code”, *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979.

[20] R. Malhotra and M. Khanna, “Particle swarm optimization-based ensemble learning for software change prediction”, *Information and Software Technology*, vol. 102, pp. 65–84, 2018.

[21] P. Probst, M. N. Wright, and A.-L. Boulesteix, “Hyperparameters and tuning strategies for random forest”, *WIREs Data Mining and Knowledge Discovery*, vol. 9, no. 3, p. e1301 (2019).

[22] S. Riaz, A. Arshad, and L. Jiao, “Rough noise-filtered easy ensemble for software fault prediction”, *IEEE Access*, vol. 6, pp. 46886–46899, 2018.

[23] H. Tong, B. Liu, and S. Wang, “Software defect prediction using stacked demising auto encoders and two-stage ensemble learning”, *Information and Software Technology*, vol. 96, pp. 94–111, 2018.

[24] N. S. Thomas and S. Kaliraj, “An improved and optimized random forest based approach to predict the software faults”, *SN Computer Science*, vol. 5, p. e530, 2024.

[25] H. Turabieh, M. Mafarja, and X. Li, “Iterated feature selection algorithms with layered recurrent neural network for software fault prediction”, *Expert Systems with Applications*, vol. 122, pp. 227–42, 2019.

[26] Z. Xu et al., “LDFR: Learning deep feature representation for software defect prediction”, *Journal of Systems and Software*, vol. 158, p. e11040, 2019.

[27] J. Yao and M. J. Shepperd, “The impact of using biased performance metrics on software defect prediction research”, *Infomation and Software Technology*, vol. 139, p. e106664, 2021.

[28] N. Zhang, S. Ying, K. Zhu, and D. Zhu, “Software defect prediction based on stacked sparse denoising autoencoders and enhanced extreme learning machine”, *IET Software*, pp. 1–19, 2021.

[29] Z.-H. Zhou, *Ensemble Methods: Foundation and Algorithm*, CRC Press, 2012.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. **ISBN: 978-1-68558-398-9**

Towards Trust Engineering in Open Data Systems: A Layered Conceptual Framework Integrating Quality Assurance and Governance Perspectives

Luciano Santos Pinheiro, Cristiane de Holanda de Barros e Silva,
Vitor Barros Aquino, Thays Maria da Conceição Silva Carvalho,
Cristiano Vale do Rego Barros Filho, Washington Henrique Carvalho Almeida
Cesar School
Recife, Brazil
email: {lsp3, chbs, vba, tmcsc, cvrbf, whca}@cesar.school

***Abstract*—Open data systems face persistent trust deficits due to silent quality regressions, schema drift, and inadequate provenance tracking. While existing frameworks address either data quality measurement or governance structures in isolation, a unified conceptual model integrating quality assurance mechanisms with governance principles remains absent. This paper proposes a five-layer conceptual framework for trust engineering in open data systems, synthesizing insights from data quality theory, data trust models, and software engineering validation practices. The framework organizes quality assurance mechanisms into a hierarchical pyramid—from structural contracts to semantic policy checks, anomaly monitoring, and observability—with each layer addressing distinct quality dimensions while collectively building trust through transparency and accountability. We position this framework within existing theoretical landscapes, including Findability, Accessibility, Interoperability, and Reusability (FAIR) principles, data trust governance, and International Organization for Standardization (ISO) quality standards, demonstrating how it extends current models by explicitly linking quality dimensions to executable validation mechanisms and publication governance decisions. Through comparative analysis of existing frameworks, we identify gaps in operationalizing trust through continuous validation and propose testable propositions for future empirical investigation. This work contributes a conceptual foundation for engineering trustworthy open data systems that balances transparency, risk management, and stakeholder accountability.**

***Index Terms*—data quality; trust engineering; open data; data governance; FAIR principles.**

I. Introduction

We present a conceptual framework for trust engineering in open data systems. Throughout this paper, *trust engineering* refers to the systematic application of engineering practices to produce verifiable trust properties in data systems; *trust-building* refers to the organizational and social processes through which stakeholders develop confidence in those systems; and *trustworthy systems* refers to the resulting artefacts whose properties have been engineered and verified. These three concepts are complementary and collectively necessary for open data trust.

A. Motivation and Problem Statement

Open data systems have become critical infrastructure for democratic governance, policy-making, and civic participation. However, persistent trust deficits undermine their potential value. Silent quality regressions—including schema drift, unstable identifiers, distribution shifts, and missing provenance—erode user confidence and limit data reuse [1], [2]. Empirical studies in software engineering corroborate this: Wu et al. demonstrate that silent label-quality errors in software datasets propagate undetected through automated pipelines, causing systematic downstream failures [41]. Unlike traditional software systems, where test-first methodologies have proven effective for quality assurance, open data ecosystems lack comparable conceptual frameworks that integrate continuous validation with governance structures.

Open data portals maintained by public agencies face additional challenges: heterogeneous consumer populations with varying technical expertise, absence of service-level agreements, and regulatory transparency requirements that demand auditability of quality decisions. The concept of *data trust*—defined here as the justified belief that a dataset accurately represents the phenomenon it purports to describe, and that its provenance and transformation history are transparent and auditable—is an engineered property produced by verifiable processes, not assumed by goodwill.

Existing theoretical work addresses data quality measurement [3], [4] and governance models [5]–[7], [29], [30] in isolation, but fails to provide an integrated conceptual foundation for engineering trust through systematic quality assurance. Quality frameworks such as ISO 25012 [3], the W3C Data Quality Vocabulary (DQV) [37], and the FAIR principles [16] enumerate desirable properties but stop short of prescribing the engineering mechanisms through which those properties are achieved and maintained over time. Data trust models emphasize participatory governance and stakeholder engagement [5], [8], while quality frameworks focus on dimensional assessment [3], [4], [9]. This fragmentation leaves practitioners without clear guidance on how quality assurance mechanisms

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

relate to trust-building practices and governance decisions.

B. Research Gap and Contribution

This paper addresses the gap between quality measurement and trust governance by proposing a five-layer conceptual framework that integrates quality assurance mechanisms with governance principles. Our framework synthesizes insights from software engineering validation practices [10], data quality theory [3], [9], and data trust models [5], [6] to provide a unified conceptual foundation for engineering trustworthy open data systems.

We contribute: (1) a five-layer conceptual framework integrating quality assurance with governance, with each layer mapped to specific quality dimensions, implementation technologies [22], [38], [40] and governance responsibilities; (2) a terminological clarification distinguishing verification ("Did we build the system right?") from validation ("Did we build the right system?") [20], [21] applied to data quality assurance; (3) a comparative analysis positioning the framework against ISO 25012, FAIR, W3C DQV, and Apache Deequ; (4) an illustrative application to the Brazilian Institute of Environment and Renewable Natural Resources (IBAMA) pesticide sales dataset [24]; and (5) six testable propositions linking framework adoption to trust outcomes.

The remainder of this paper is structured as follows. Section II reviews background and theoretical foundations. Section III describes the five-layer framework. Section IV presents the IBAMA illustrative application. Section V presents comparative analysis. Section VI presents six research propositions for empirical validation. Section VII discusses implications and limitations. Section VIII concludes.

II. Background and Theoretical Foundations

We organize related work along four dimensions: (i) data quality measurement frameworks and standards; (ii) data trust and governance models; (iii) software engineering validation practices; and (iv) FAIR principles and data stewardship. We then identify the gap that our framework addresses.

A. Data Quality Dimensions and Standards

Data quality research has established multidimensional frameworks for assessing fitness for use. Wang and Strong's seminal work identified fifteen quality dimensions organized into intrinsic, contextual, representational, and accessibility categories [9]. ISO/IEC 25012 standardized quality characteristics including accuracy, completeness, consistency, and credibility [3]. These frameworks provide taxonomies for quality assessment but lack operational guidance on implementing continuous validation mechanisms.

At the implementation level, several open-source tools operationalize quality measurement. Great Expectations [38] introduces *expectations*—declarative assertions about data properties evaluated at runtime. Soda Core [39] adopts a domain-specific language (SodaCL) for defining checks embeddable in orchestration pipelines. The dbt testing framework [40] integrates schema and referential integrity tests directly into transformation workflows. Apache Deequ provides a Scala/Spark-based library for automated quality monitoring at scale. Recent work extends dimensional models to open data contexts. Vetrò et al. proposed quality metrics tailored to open government data, emphasizing timeliness, accuracy, and accessibility [4]. Gong et al. confirm that completeness, consistency, and timeliness remain the most operationally critical dimensions while identifying the absence of integrated governance mechanisms as a persistent gap [42].

B. Data Trust and Governance Models

Data trust frameworks emphasize governance structures that enable stakeholder participation and accountability. Milne and Brayne's data trust model proposes independent stewardship, participatory governance, and transparent decision-making as trust-building mechanisms [5]. Radosevic et al. extend this model to spatial data infrastructures [6]. Artyushina's civic data trust framework highlights transparency, accountability, and community participation [8]. The UK Food Standards Agency's food data trust initiative further demonstrates how sector-specific governance structures can operationalize data stewardship principles in practice [32].

Data mesh architectures [35] distribute governance responsibility to domain teams, introducing data products with embedded quality contracts. DataOps [31] applies continuous integration principles to data pipelines. While these approaches advance practice, they do not provide a unified theoretical model mapping specific quality dimensions to specific governance decisions. The present framework addresses this gap by making the governance decision the mandatory final step in the publication pipeline.

C. Software Engineering Validation Practices

The distinction between verification and validation, introduced by Boehm [20] and formalized in IEEE Std 1012 [21], is central to the framework's design. Verification asks: "Are we building the product right?"—it checks conformance to a specification (e.g., does the dataset schema match the published contract?). Validation asks: "Are we building the right product?"—it checks fitness for the intended use (e.g., do the pesticide sales figures accurately reflect market reality?). In the context of data quality assurance, Layers 1–3 of our framework are primarily verification activities; Layer 4 supports both; Layer 5 is a validation activity in which human stewards exercise judgment about fitness for publication.

Software engineering has developed mature practices for continuous validation. Test-driven development establishes executable specifications that prevent regressions [10]. Design by contract formalizes preconditions, postconditions, and invariants as enforceable constraints [11]. Observability engineering provides runtime visibility into system behavior through structured logging, metrics, and tracing [12].

These practices have proven effective for maintaining software quality but have not been systematically adapted to open

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

data contexts. Our framework bridges this gap by translating software validation concepts to data quality assurance.

D. FAIR Principles and Data Stewardship

The FAIR principles—Findability, Accessibility, Interoperability, and Reusability—provide foundational guidelines for scientific data management [16]. Nicholson et al. demonstrate that FAIR compliance does not imply quality: a dataset can be fully FAIR-compliant yet contain semantic errors, distributional anomalies, or outdated provenance records [17]. This observation motivates the explicit inclusion of a quality assurance layer beneath the governance layer: FAIR addresses discoverability and accessibility, whereas our framework addresses fitness for use and the engineering processes that sustain it.

Quality frameworks such as ISO 25012 [3], the W3C Data Quality Vocabulary (DQV) [37], and the FAIR principles [16] each address a subset of the quality-governance space but remain high-level guidelines requiring operationalization through specific technical mechanisms.

III. Five-Layer Conceptual Framework

Our framework organizes quality assurance mechanisms into five hierarchical layers, each addressing distinct quality dimensions while collectively building trust through continuous validation and transparent reporting. The layers progress from low-level syntactic contracts to high-level semantic and organizational controls, mirroring the Open Systems Interconnection (OSI) network model's principle of layered abstraction. Each layer addresses a distinct class of quality failures; higher layers assume the guarantees provided by lower layers. Conflicts are resolved by the governance layer (Layer 5), which has authority to halt publication pending remediation. Alternative decompositions—by quality dimension, stakeholder role, or data lifecycle stage—were considered but rejected in favour of the hierarchical technical-to-governance ordering, which reflects natural implementation dependencies and supports incremental adoption: organisations can implement lower layers first and gain immediate value before adding higher layers.

Figure 1 illustrates both the framework architecture and the validation pipeline: the pyramid layers represent the five trust levels (L1–L5), and the Remediation Queue panel on the left shows how failed checks at each layer are routed for corrective action before re-ingestion.

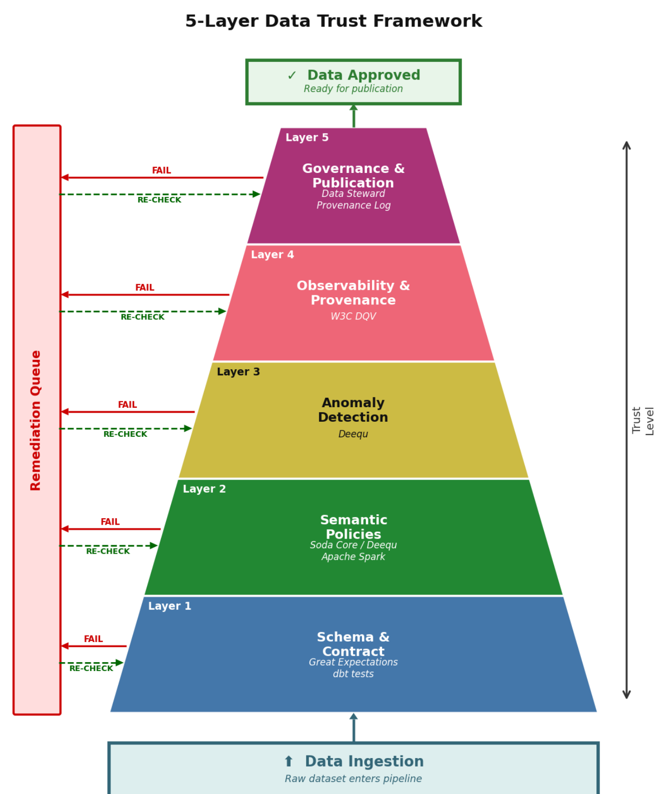


Fig. 1. Five-layer trust engineering framework with integrated remediation pipeline. Trust accumulates from structural contracts (Layer 1) to governance decisions (Layer 5); failed checks at each layer route to the Remediation Queue (left panel) for automated or manual correction before re-ingestion. Implementation tools are shown per layer.

A. Layer 1: Structural Contracts

Structural contracts establish foundational guarantees about data schema, types, and required fields. Drawing from design-by-contract principles [11], this layer defines machine-readable specifications that prevent schema drift and ensure structural consistency. Structural contracts address the accuracy, completeness, and consistency dimensions by enforcing type constraints, nullability rules, and referential integrity.

Implementation technologies include: Great Expectations [38] `ExpectationSuite` objects serialised as JSON (versionable alongside the dataset); OpenAPI [25] schemas for API-delivered data; and dbt [40] schema YAML files defining column-level constraints evaluated on every pipeline run. These contracts serve as executable documentation, enabling automated validation at ingestion and publication boundaries.

B. Layer 2: Semantic Policies

Semantic policies enforce domain-specific business rules and logical constraints that transcend structural validation. This layer addresses semantic accuracy and logical consistency by validating relationships between fields, enforcing domain constraints, and detecting logical inconsistencies. Semantic policies translate domain knowledge into executable rules that prevent semantically invalid data from propagating through systems.

Examples include range constraints (e.g., pesticide sales volumes must be non-negative for final annual records), cross-field validations (e.g., end date must follow start date), entity resolution rules [22], and controlled-vocabulary conformance. Soda Core [39] SodaCL checks express semantic rules in a human-readable domain-specific language evaluated at runtime; Apache Deequ `Check` objects support constraint verification at scale on Apache Spark. Semantic policies require domain expertise to define but provide critical protection against logically inconsistent data that passes structural validation.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

This layer bridges the gap between syntactic correctness and semantic validity.

C. Layer 3: Anomaly Detection

Anomaly detection monitors distributional properties and temporal patterns to identify unexpected changes in data characteristics. This layer addresses the timeliness, consistency, and credibility dimensions by detecting distribution shifts, outliers, and temporal anomalies that may indicate quality degradation or upstream process changes [26], [27].

The framework employs a two-stage approach: (i) automated statistical monitoring using control charts or Z-score thresholds to flag candidate anomalies, and (ii) steward review to classify flagged observations as natural behavior (document and accept), data error (remediate), or genuine anomaly (quarantine and investigate). This design prevents the framework from suppressing real signals while ensuring genuine errors are not published. Apache Deequ provides automated anomaly detection; the MOA framework [28] supports concept-drift detection for streaming data. Thresholds are determined per column using a rolling baseline window (default: 12 periods) and flagging observations that deviate by more than $k\sigma$ from the baseline mean, where k is set by the data steward based on acceptable false-positive rates (recommended $k = 3$ for initial deployment). Concept drift—a sustained shift in the underlying data distribution rather than an isolated anomaly—is distinguished from point errors by applying the Page-Hinkley test [28] over the same rolling window; a confirmed drift triggers a schema evolution review rather than a remediation action. Anomaly detection complements rule-based validation by identifying quality issues that cannot be anticipated through explicit constraints. This layer provides early warning of quality degradation before downstream impacts occur.

D. Layer 4: Observability and Provenance

Observability mechanisms provide transparency into data lineage, transformation history, and quality metrics. Drawing from observability engineering [12] and provenance research [13]–[15], this layer addresses the credibility, traceability, and understandability dimensions by documenting data origins, transformations, and quality assessments.

Provenance tracking captures who produced data, when, through what processes, and with what quality characteristics. Provenance records include: source system identification, ingestion timestamps, transformation steps with input/output checksums, quality check results from Layers 1–3, and steward decision records from Layer 5. The W3C PROV-DM standard [15] provides vocabulary for representing provenance as a directed acyclic graph of entities, activities, and agents. The W3C DQV [37] provides complementary vocabulary for publishing quality metadata as linked data. A minimal provenance record for each dataset version includes the following mandatory fields: `source_uri` (origin endpoint), `ingest_timestamp` (ISO 8601), `checksum_sha256` (structural fingerprint), `l1_pass`/`l2_pass`/`l3_pass` (boolean layer results), `anomaly_flags` (flagged column–period pairs), and `steward_decision` (approve, quarantine, or remediate). The observability dashboard exposes these fields as a time-series quality log, enabling consumers to inspect quality history and compare metrics across releases. Observability transforms opaque data pipelines into transparent, auditable systems.

E. Layer 5: Governance and Publication Decisions

The governance layer integrates quality signals from lower layers into publication and access control decisions. This layer addresses accountability, compliance, and risk management by establishing thresholds for publication, defining stakeholder roles, and implementing feedback mechanisms. Governance policies translate quality assessments into actionable decisions about data release, access restrictions, and quality warnings.

The governance model specifies: (i) a *data steward* role responsible for reviewing quality check results and making publication decisions; (ii) a *data owner* role responsible for defining quality policies and accepting residual risk; (iii) a *data consumer* role with the right to access quality metadata and provenance records; and (iv) a change management process for schema and policy updates. Publication decisions are recorded in the provenance log (Layer 4), creating an auditable governance trail. The primary advantage of integrating quality measurement (Layers 1–4) with governance (Layer 5) is the elimination of the accountability gap: no accountable actor can bypass the quality evidence review before publication. Publication eligibility is determined by a composite Global Quality Score $Q_s = w_1L_1 + w_2L_2 + w_3L_3 + w_4L_4$, where $L_i \in [0, 1]$ is the pass rate for layer i and weights w_i sum to 1. Default weights ($w_1 = 0.3, w_2 = 0.3, w_3 = 0.2, w_4 = 0.2$) reflect the foundational importance of structural and semantic layers; the data owner has authority to adjust weights to reflect domain-specific risk tolerance. A dataset is eligible for publication if $Q_s \geq \theta$, where the publication threshold θ (default $\theta = 0.85$) is set by the data owner and reviewed annually or following any governance incident.

This layer operationalizes data trust principles [5], [6] by connecting technical quality mechanisms to organizational accountability structures. Governance frameworks define who can publish data, under what quality conditions, with what transparency requirements, and through what stakeholder engagement processes. This layer closes the loop between quality assurance and trust governance.

IV. Illustrative Application: IBAMA Pesticide Sales Dataset

To demonstrate practical applicability, we apply the framework to the IBAMA pesticide sales dataset [24], a publicly available open data artefact published by the Brazilian federal environmental agency. The dataset contains 124,245 records covering 584 active ingredients across 19 years (2007–2026), distributed across 27 Brazilian states, encoded in UTF-8, semicolon-delimited, with 33 columns including

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

`Ingrediente_ativo`, `Ano`, `Semestre`, and 27 state-level sales columns measured in tonnes of commercial product.

Layer 1 (Structural Contracts): A Great Expectations `ExpectationSuite` was defined specifying 33 columns in fixed order, UTF-8 encoding, semicolon delimiter, and integer type for `Ano` (range 2007–2026). All structural checks passed, confirming encoding and schema integrity.

Layer 2 (Semantic Policies): Semantic checks revealed 10,959 records with negative sales values in state columns—semantically anomalous values representing returns or corrections. A Soda Core check would flag these for steward review. The policy decision requires domain expertise and is escalated to Layer 5.

Layer 3 (Anomaly Detection): Statistical monitoring identified a 15.6% decline in glyphosate sales between 2022 (614,329 tonnes) and 2023 (517,983 tonnes). Steward review classified this as natural behavior attributable to documented regulatory changes. A structural change was also identified: two semesters per year for 2007–2021 but only one from 2022 onwards—a schema evolution not documented in the metadata, representing a provenance gap flagged by Layer 4.

Layers 4–5 (Provenance and Governance): The IBAMA portal records a last-update timestamp but does not publish a transformation lineage or quality check history. The framework prescribes three governance decisions requiring steward action: (i) classification of 10,959 negative-value records; (ii) documentation of the 2022 semester-structure change; and (iii) establishment of a provenance publication policy for future updates.

V. Comparative Analysis

We position our framework within existing theoretical landscapes by comparing it to established models across three dimensions: quality focus, governance integration, and operational specificity.

A. Comparison with Quality Frameworks

ISO/IEC 25012 [3] and Wang and Strong's framework [9] provide comprehensive quality taxonomies but lack operational guidance on implementing continuous validation. Our framework extends these models by mapping quality dimensions to specific validation mechanisms organized hierarchically. Where ISO 25012 defines accuracy as a quality characteristic, our framework specifies structural contracts, semantic policies, and anomaly detection as complementary mechanisms for ensuring accuracy at different levels of abstraction.

Vetrò et al.'s open data quality metrics [4] emphasize measurement but do not address prevention or continuous monitoring. Our framework integrates measurement with proactive validation, shifting from reactive quality assessment to preventive quality engineering.

B. Comparison with Data Trust Models

Data trust frameworks [5], [6], [8] emphasize governance structures, stakeholder participation, and transparency but provide limited technical specificity regarding quality assurance mechanisms. Our framework operationalizes trust principles by connecting governance decisions to concrete quality validation layers. Where Milne and Brayne emphasize independent stewardship and transparent decision-making [5], our framework specifies how observability and provenance mechanisms enable transparency, and how governance layers translate quality signals into publication decisions.

TABLE I. Comparison of the Proposed Framework Against Related Models.

Dimension	Proposed	ISO 25012	FAIR	W3C DQV	Deequ
Quality dims.	Yes	Yes	Partial	Partial	Yes
Exec. mechs.	Yes	No	No	No	Yes
Governance	Yes	No	No	No	No
Provenance	Yes	No	Partial	Yes	No
Anomaly det.	Yes	No	No	No	Yes
Open data	Yes	No	Yes	Yes	No

Recent governance frameworks [7], [29], [30] propose organizational structures and policy guidelines but lack integration with technical quality mechanisms. Our framework bridges this gap by explicitly linking governance decisions to quality assurance outputs.

C. Comparison with FAIR Principles

FAIR principles [16] provide high-level guidelines for data stewardship but require operationalization through specific mechanisms. Our framework operationalizes FAIR principles: structural contracts ensure interoperability through standardized schemas, provenance tracking enhances findability and reusability, and observability mechanisms support accessibility through transparent quality reporting. Our framework extends FAIR by adding continuous validation and anomaly detection, addressing temporal quality dimensions not explicitly covered by FAIR principles. This extension is critical for open data systems where quality degrades over time through schema drift and distributional shifts.

D. Gaps in Existing Frameworks

Comparative analysis reveals three critical gaps: (1) fragmentation between quality measurement and governance structures; (2) lack of operational guidance on implementing continuous validation; and (3) insufficient attention to temporal quality dimensions and quality degradation over time. Our framework addresses these gaps by integrating quality assurance with governance, providing hierarchical organization of validation mechanisms, and emphasizing continuous monitoring through anomaly detection and observability. Table I summarises the comparison; our framework is the only model that simultaneously addresses all six dimensions.

VI. Research Propositions

We articulate six testable propositions linking framework adoption to trust outcomes, quality improvements, and organizational practices. These propositions guide future empirical investigation.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

P1 (Trust and Transparency): *Open data systems implementing observability and provenance mechanisms (Layer 4) will exhibit higher stakeholder trust compared to systems without such mechanisms, mediated by perceived transparency.*
This proposition draws from data trust literature emphasizing transparency as a trust antecedent [5], [8]. Empirical testing requires measuring stakeholder trust before and after implementing observability mechanisms, controlling for data quality levels.

P2 (Quality and Validation Layers): *Open data systems implementing multiple validation layers will demonstrate fewer quality defects in production compared to systems implementing single-layer validation, with diminishing returns beyond three layers.*
This proposition reflects the hierarchical nature of quality assurance, where each layer addresses distinct defect types. Empirical testing requires longitudinal tracking of defect rates across systems with varying numbers of validation layers.

P3 (Governance and Quality Signals): *Open data systems integrating quality signals into publication decisions (Layer 5) will exhibit more consistent quality levels over time compared to systems with decoupled quality assessment and publication processes.*
This proposition addresses the gap between quality measurement and governance action. Testing requires comparing quality variance over time between systems with integrated versus decoupled governance.

P4 (Anomaly Detection and Timeliness): *Open data systems implementing continuous anomaly detection (Layer 3) will identify quality degradation earlier than systems relying solely on rule-based validation, reducing mean time to detection by at least 50%.*
This proposition emphasizes the value of behavioral monitoring beyond static rules. Testing requires measuring the time elapsed between quality degradation onset and detection across different validation approaches.

P5 (Semantic Policies and Domain Expertise): *The effectiveness of semantic policy layers (Layer 2) in preventing quality defects is positively moderated by the level of domain expertise involved in policy definition.*
This proposition recognizes that semantic validation quality depends on domain knowledge. Testing requires comparing defect rates across systems with varying levels of domain expert involvement in policy definition.

P6 (Framework Adoption and Organizational Maturity): *Organizations with higher data governance maturity will adopt framework layers in hierarchical order (Layers 1–5), while organizations with lower maturity will adopt layers opportunistically, resulting in lower overall effectiveness.*
This proposition addresses implementation pathways and organizational readiness. Testing requires longitudinal case studies tracking adoption patterns and effectiveness across organizations with varying maturity levels.

VII. DISCUSSION

This section examines the theoretical, practical, and boundary implications of the proposed framework, situating its contributions within the broader literature and identifying conditions that shape its applicability.

A. Theoretical Implications

Our framework contributes to data quality theory by integrating quality dimensions with executable validation mechanisms, thereby addressing the gap between measurement and engineering. By organizing mechanisms hierarchically, we provide conceptual clarity about how different validation approaches complement one another. The framework extends software engineering validation practices to open data contexts, demonstrating how test-driven development, design by contract, and observability principles apply to data quality assurance.

The framework also contributes a theoretical account of why existing integrations—DAMA-DMBOK, DataOps literature, and data mesh architectures—fall short for open data systems. DAMA-DMBOK [36] addresses process maturity but not the technical architecture of validation layers. DataOps focuses on pipeline velocity rather than publication governance. Data mesh distributes ownership but does not specify inter-domain quality contracts. Our framework addresses these gaps by providing an explicit mapping from quality signals to governance decisions.

The framework also contributes to trust theory by demonstrating that trust is a systemic property emerging from the interaction of technical mechanisms (Layers 1–4) and organizational processes (Layer 5). Trust cannot be achieved by technical means alone; it requires governance structures that translate quality evidence into accountable decisions. This account bridges the gap between socio-technical trust models and technical quality practices.

The framework also contributes to data trust theory by operationalizing trust principles through technical mechanisms. Where existing trust models emphasize governance structures and stakeholder participation, our framework specifies how quality assurance mechanisms enable transparency, accountability, and informed trust decisions. This integration bridges the gap between socio-technical trust models and technical quality practices.

The framework can be mapped to the three trust dimensions identified by Mayer et al. [43]: *competence* trust (the belief that the trustee has the ability to perform as expected), *integrity* trust (adherence to acceptable principles), and *benevolence* trust (acting in the trustor's interest). Layers 1–3 address competence trust by providing verifiable evidence of correct quality-check performance. Layer 4 addresses integrity trust by making transformation history and quality decisions transparent and auditable. Layer 5 addresses benevolence trust by establishing accountable roles and publication policies that demonstrate the data owner acts in consumers' interests—confirming the framework addresses all three trust dimensions, not merely technical quality assurance.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

B. Practical Implications

For practitioners, the framework provides a roadmap for implementing quality assurance in open data systems. The hierarchical organization suggests implementation priorities: establish structural contracts first, then add semantic policies, followed by anomaly detection and observability. This staged approach enables incremental adoption aligned with organizational maturity and resource constraints.

The framework also guides tool selection and architectural decisions. Each layer maps to specific technology categories: schema validation tools for structural contracts [23], rule engines for semantic policies, statistical monitoring for anomaly detection, and provenance systems for observability. This mapping helps practitioners translate the conceptual framework into concrete implementations.

C. Limitations and Boundary Conditions

Our framework is conceptual and requires empirical validation through case studies and controlled experiments. The propositions articulated in Section VI provide starting points for such validation but remain untested. The framework emphasizes technical quality mechanisms and may underweight social and organizational factors that influence trust in practice. Trust is fundamentally socio-technical, emerging from interactions among technical systems, organizational practices, and stakeholder relationships. While our framework addresses the technical mechanisms that enable transparency and accountability, it does not fully specify the organizational processes and stakeholder engagement practices required for trust building.

The framework assumes that organizations have sufficient technical capacity to implement all validation layers. For resource-constrained organizations, full implementation may be infeasible. Future work should investigate lightweight implementations and identify minimum viable configurations for different organizational contexts. Trade-offs between validation rigor and computational cost require investigation, particularly for high-volume data streams where comprehensive validation may introduce unacceptable latency [19]. Incremental adoption pathways that deliver value at each stage while building toward comprehensive implementation need further specification.

The framework focuses on structured and semi-structured data [18], with limited applicability to unstructured data (text, images, video). Extending the framework to unstructured data contexts requires additional conceptual development, particularly for semantic validation and anomaly detection layers. Quality dimensions for unstructured data differ from those for structured data, emphasizing aspects such as relevance, coherence, and contextual appropriateness that resist formal specification. Machine learning-based quality assessment for unstructured data introduces additional challenges, including model bias and interpretability.

The framework does not explicitly address adversarial scenarios in which data producers intentionally manipulate quality metrics or validation mechanisms. Security considerations—including data integrity verification, tamper detection, and audit trail protection—require integration with the framework. The relationship between data quality assurance and data security practices merits further investigation.

VIII. Conclusion and Future Work

This paper proposed a five-layer conceptual framework for trust engineering in open data systems, integrating quality assurance mechanisms with governance principles. The framework organizes validation mechanisms hierarchically—from structural contracts to semantic policies, anomaly detection, observability, and governance—with each layer addressing distinct quality dimensions while collectively building trust through continuous validation and transparent reporting.

Through comparative analysis, we positioned the framework within existing theoretical landscapes, including ISO quality standards, data trust models, and FAIR principles, demonstrating how it extends current models by explicitly linking quality dimensions to executable mechanisms and governance decisions. We identified three critical gaps in existing frameworks: fragmentation between quality measurement and governance, lack of operational guidance on continuous validation, and insufficient attention to temporal quality dimensions.

We articulated six testable propositions linking framework adoption to trust outcomes, quality improvements, and organizational practices, providing a research agenda for empirical validation. The framework contributes a conceptual foundation for engineering trustworthy open data systems that balances transparency, risk management, and stakeholder accountability, bridging the gap between data quality theory, data trust governance, and software engineering validation practices.

Future research should pursue four directions. First, empirical validation through longitudinal case studies tracking framework adoption, implementation challenges, and trust outcomes across diverse organizational contexts. Such studies should test the six propositions in Section VI, examining both successful implementations and failed adoption attempts to identify critical success factors, and compare adaptations across domains (government, healthcare, scientific research).

Second, development of a reference implementation as an open-source pipeline integrating Great Expectations [38], Soda Core [39], dbt [40], and a W3C PROV-DM provenance store, deployed against the IBAMA dataset as a reproducible benchmark. Reference architectures for common technology stacks (cloud platforms, data lakes, data meshes [35]) would provide concrete implementation guidance.

Third, extension of the framework to emerging data contexts, including real-time streaming data, federated data systems, and artificial intelligence training datasets. Streaming contexts require adaptation of anomaly detection mechanisms to handle concept drift and temporal dependencies. Federated systems require distributed validation and consistency protocols across organizational boundaries. AI training datasets require additional validation layers addressing bias, represen-

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

tativeness, and fairness [33], including training-serving skew and model decay [34].

Fourth, investigation of organizational factors influencing framework adoption and effectiveness. Research should examine how organizational culture, governance maturity, resource constraints, and stakeholder diversity affect implementation success and trust outcomes. The relationship between framework adoption and broader DataOps practices [31] merits exploration, as does integration with data mesh architectures emphasizing domain ownership and federated governance [35].

References

[1] B. W. Wirtz, J. C. Weyerer, and C. Geyer, "Artificial Intelligence and the Public Sector—Applications and Challenges," *International Journal of Public Administration*, vol. 42, no. 7, pp. 596–615, 2019.

[2] J. Attard, F. Orlandi, S. Scerri, and S. Auer, "A systematic review of open government data initiatives," *Government Information Quarterly*, vol. 32, no. 4, pp. 399–418, 2015.

[3] ISO/IEC, *ISO/IEC 25012:2008—Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Data Quality Model*. Geneva: International Organization for Standardization, 2008.

[4] A. Vetrò, L. Canova, M. Torchiano, C. O. Minotas, R. Iemma, and F. Morando, "Open data quality measurement framework: Definition and application to open government data," *Government Information Quarterly*, vol. 33, no. 2, pp. 325–337, 2016.

[5] R. Milne, J. Morley, and H. S. Howard, "Data trusts and the governance of health data," *Nature Medicine*, vol. 28, pp. 2218–2220, 2022.

[6] D. Radosevic, M. Cetl, and V. Cetl, "Spatial Data Trust Framework," *ISPRS International Journal of Geo-Information*, vol. 12, no. 11, p. 456, 2023.

[7] B. C. Stahl, D. Wright, and M. Wakunuma, "Ethics of AI and big data: Governance frameworks and their implications," *AI & Society*, vol. 40, pp. 1–15, 2025.

[8] A. Artyushina, "The civic data trust: A new model for data stewardship," *Data & Policy*, vol. 2, e7, 2020.

[9] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," *Journal of Management Information Systems*, vol. 12, no. 4, pp. 5–33, 1996.

[10] K. Beck, *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley, 2003.

[11] B. Meyer, "Applying 'Design by Contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[12] C. Majors, L. Fong-Jones, and G. Sheffield, *Observability Engineering*. Sebastopol, CA: O'Reilly Media, 2022.

[13] J. Cheney, L. Chiticariu, and W. C. Tan, "Provenance in databases: Why, how, and where," *Foundations and Trends in Databases*, vol. 1, no. 4, pp. 379–474, 2009.

[14] Y. L. Simmhan, B. Plale, and D. J. Gannon, "A survey of data provenance in e-Science," *ACM SIGMOD Record*, vol. 34, no. 3, pp. 31–36, 2005.

[15] P. Groth and L. Moreau, "PROV-Overview: An Overview of the PROV Family of Documents," W3C Working Group Note, Apr. 2013. [Online]. Available: https://www.w3.org/TR/prov-overview/ [retrieved: Apr. 2026].

[16] M. D. Wilkinson *et al.*, "The FAIR Guiding Principles for scientific data management and stewardship," *Scientific Data*, vol. 3, p. 160018, 2016.

[17] N. Nicholson, R. N. Carvalho, and I. Štotl, "A FAIR Perspective on Data Quality Frameworks," *Data*, vol. 10, no. 9, p. 136, Aug. 2025.

[18] C. J. Date, *Database Design and Relational Theory*, 2nd ed. Sebastopol, CA: O'Reilly Media, 2019.

[19] M. Kleppmann, *Designing Data-Intensive Applications*. Sebastopol, CA: O'Reilly Media, 2017.

[20] B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE Software*, vol. 1, no. 1, pp. 75–88, Jan. 1984.

[21] IEEE, *IEEE Standard for System, Software, and Hardware Verification and Validation*, IEEE Std 1012-2016. New York, NY: IEEE, 2017.

[22] L. Getoor and A. Machanavajjhala, "Entity resolution: Theory, practice & open challenges," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 2018–2019, 2012.

[23] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger, "Automating large-scale data quality verification," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1781–1794, 2018.

[24] Brazilian Institute of Environment and Renewable Natural Resources (IBAMA), "Comercialização de Agrotóxicos por Unidade da Federação," Open Data Portal, 2024. [Online]. Available: https://dadosabertos.ibama.gov.br/dados/AGROTX/produtosestado_csv.zip [retrieved: Apr. 2026].

[25] OpenAPI Initiative, "OpenAPI Specification v3.2.0," 2025. [Online]. Available: https://spec.openapis.org/oas/latest.html [retrieved: Apr. 2026].

[26] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 1–58, 2009.

[27] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd ed. Waltham, MA: Morgan Kaufmann, 2011.

[28] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: Massive online analysis," *Journal of Machine Learning Research*, vol. 11, pp. 1601–1604, 2010.

[29] A. Sarran, R. Ramnarain-Seetohul, and S. Cadersaib, "Towards a Data Governance Model for Enhanced Data Quality Management," *International Journal of Information Technology, Research and Applications*, vol. 3, no. 4, p. 110, 2024.

[30] O. Adeyemi and C. Okonkwo, "A conceptual framework for data governance in big data and cloud environments," *International Journal of Science and Research Archive*, vol. 12, no. 2, pp. 1177–1195, 2024.

[31] A. Reis and R. Housley, "DataOps: Towards a Definition," in *Proc. 13th Int. Conf. Software Technologies (ICSOFT)*, Porto, Portugal, 2018, pp. 104–111.

[32] Food Standards Agency, "Food Data Trust: Legal, Structuring and Governance," *FSA Research and Evidence*, London, UK, 2021.

[33] S. Barocas, M. Hardt, and A. Narayanan, *Fairness and Machine Learning: Limitations and Opportunities*. Cambridge, MA: MIT Press, 2023.

[34] D. Sculley *et al.*, "Hidden technical debt in machine learning systems," in *Proc. 29th Conf. Neural Information Processing Systems (NIPS)*, Montreal, Canada, 2015, pp. 2503–2511.

[35] Z. Dehghani, *Data Mesh: Delivering Data-Driven Value at Scale*. Sebastopol, CA: O'Reilly Media, 2022.

[36] DAMA International, *DAMA-DMBOK: Data Management Body of Knowledge*, 2nd ed. Basking Ridge, NJ: Technics Publications, 2017.

[37] R. Albertoni and A. Isaac, "Data on the Web Best Practices: Data Quality Vocabulary," W3C Working Group Note, Dec. 2016. [Online]. Available: https://www.w3.org/TR/vocab-dqv/ [retrieved: Apr. 2026].

[38] Great Expectations, "Great Expectations: Data Validation Framework," 2023. [Online]. Available: https://greatexpectations.io [retrieved: Apr. 2026].

[39] Soda, "Soda Core: Open-Source Data Quality Framework," 2023. [Online]. Available: https://github.com/sodadata/soda-core [retrieved: Apr. 2026].

[40] dbt Labs, "dbt: Data Build Tool," 2023. [Online]. Available: https://www.getdbt.com [retrieved: Apr. 2026].

[41] J. Wu, Y. Tian, F. Thung, D. Lo, and C. Tantithamthavorn, "Data Quality Matters: A Case Study on Data Label Correctness for Security Bug Report Prediction," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2541–2556, Jul. 2021.

[42] X. Gong, J. Liu, Y. Zhao, and H. Wang, "A survey on dataset quality in machine learning," *Information and Software Technology*, vol. 162, p. 107268, Oct. 2023.

[43] R. C. Mayer, J. H. Davis, and F. D. Schoorman, "An integrative model of organizational trust," *Academy of Management Review*, vol. 20, no. 3, pp. 709–734, 1995.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. **ISBN: 978-1-68558-398-9**

Towards the Static Detection of Compromised Software Components by Topological Anomalies

Oscar Z. de Paiva, Wilson V. Ruggiero, Marcos A. Simplicio Jr.
Computer Engineering Department
Polytechnic School – Universidade de São Paulo
São Paulo, Brazil
e-mail: {ozpaiva | wilson | msimplicio}@larc.usp.br

***Abstract*—Open-source components are critical assets for reducing software development costs and fostering innovation throughout the globe. However, they introduced serious attack venues in modern software supply-chains, as attackers have been able to inject malicious code into these components with relative ease through various means. Such attacks have increased over the past five years and remain frequent despite the various protection mechanisms proposed to prevent or mitigate them. To address this issue, this paper presents a novel protection mechanism aimed at detecting malicious code injected into open-source components and applications via supply-chain attacks. Due to the relevance of technology in corporate environments, we selected Java as the focus of our study. The proposed mechanism is based on a topological static analysis approach: it extracts topological features of uncontaminated programs, and associates these features to normal (i.e., non-malicious) connections of security-critical methods to program parts, and with each other via these parts. These associations are then used to identify the presence of malicious code as topological anomalies in compromised programs. Preliminary results, obtained from a partial implementation of the technique, indicate high detection accuracy (ranging from 86.3% to 99.98% in a publicly-available dataset), and a small number of false-positive classifications – illustrating that the proposed technique is robust and useful in practice. The paper also discusses some potential limitations of our proposal, as well as possible improvements to address them.**

Keywords-Open-Source Security; Supply Chain Attacks; Malware Detection

I. Introduction

Most of the code of modern software applications consists of direct or indirect third-party dependencies, especially Open-Source Components (OSCs) [1]. This demonstrates how essential OSCs have become in reducing software development costs and promoting innovation around the world. However, the ubiquity of these components has unfortunately opened new attack venues for malicious actors to compromise systems via the Software Supply Chains (SSCs) on which these depend. Indeed, in the last decade, attackers have been able to infiltrate communities involved in OSC development, or compromise systems used by them, to inject malicious code and "trojanize" them with relative ease [2]. In 2025, more than one million compromised OSCs were detected by a major vendor [3].

The occurrence of such attacks has not gone unnoticed by cybersecurity specialists, which began to propose various solutions and also mandate/encourage their adoption [4]. Despite those advances, the risk associated with SSC attacks in OSCs remains largely unmitigated.

To address this issue, this article proposes a novel static analysis technique for detecting the types of malicious code most commonly embedded in OSCs through SSC attacks. The forms and behaviors of these types of code were extracted from the "Backstabber's Knife Collection" dataset [5], the most important collection of code disseminated through SSC attacks to date. Essentially, the code corresponds to small excerpts (of no more than a dozen statements). They are also self-contained, i.e., do not exchange data with benign code, and call one or more security-critical methods (mostly related to network access, file I/O and command interpretation) from default APIs of the execution environment. These excerpts implement behaviors typical of *data exfiltrators*, *droppers*, *backdoors*, and *reverse shells*. Hence, they are all associated with the goal of either directly exfiltrating data from a computer system or penetrating it as the first step of an attack.

Our technique is based on a topological approach to static analysis, through which high-level features of a program's control-flow are inspected. These features, while not providing formal guarantees about the program's security properties, enable the identification of topological anomalies that may indicate the presence of malicious code excerpts in it. The analysis begins with a division of the program's call graph into parts, each associated with a cohesive component. We then extract local and global features, i.e., features related to the parts themselves and to how they are connected, respectively. Connections of security-critical methods to the parts, and between the methods through the parts, are then used to select features that may indicate anomalies.

Motivated by the widespread adoption of Java in high-risk enterprise applications, we tested this approach on a synthetic set of Java programs built using XCorpus as basis [6]. Our preliminary results, comprising the evaluation of a detector based solely on local features, indicate that high accuracy is achievable with our technique. Moreover, some of the features were also designed to increase the cost of evasion attacks, which would require the insertion of spurious code that, at least in principle, may be identified by algorithms of dead-code detection. This may also enable the design of threat models based on the percentage of the SSC controlled by the attacker.

The paper is organized as follows. Section II describes our technique, and preliminary results are presented in Section III. Section IV discusses aspects related to the technique's resistance to evasion. Related works are summarized in section

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

V. Section VI concludes the discussion and presents ideas for future works.

II. Proposed Technique

Our technique relies primarily on a two-step division of the input program's call graph into hierarchically-related parts. Each part produced in the first step corresponds to one of the *packages* to which belong the classes that constitute the program. In the second step, each of these parts is then mapped to an element of the set of *package groups* that constitute the program. For simplicity, those groups are defined in this work by proximity relations between package names, leveraging the naming conventions usually adopted by Java developers.

The partitioning of the call graph into package and package groups provides natural guidance to analyze the topology of the program, following the definitions provided in its code and in the code of its direct and indirect dependencies. Each part contains methods belonging to a single library or to a group of related libraries and, thus, is expected to exhibit characteristics that distinguish it from other parts. Moreover, the relations between parts express how the program relies on its dependencies to operate, and how these dependencies rely on other dependencies, and so on, transitively.

To detect the types of malicious code snippets, we analyze how security-critical methods from standard Java APIs are normally connected to parts of the program and through them. The definition of "normalcy" in this instance relies on features of the program parts and of the relations between them, such that a random insertion of a malicious code snippet into a part that does not exhibit the expected feature values, can be detected as an anomaly. If connections to and between security critical methods are associated with sufficiently rare feature values (and our preliminary results indicate that this is the case), high detection accuracy can be achieved. Moreover, together, these features may contribute to increase the cost of attacks for the difficulty in forging their values – as that would require inserting spurious method calls into the program in a way as to match expected feature values associated with the connection(s) introduced into the parts by the malicious snippet.

To characterize the types of connections to and between security-critical methods associated with the malicious excerpts, it is necessary to establish the distinction between two cases. In the first one (which we denote as Jx), malicious behaviors are implemented with methods from more than one category of functionality from standard APIs. For example, a data exfiltrator can be implemented with methods from the file and network access categories, while a backdoor can be implemented with methods from the network and command interpretation categories. The second case (denoted as NJ) refers to behaviors implemented by a single category of functionality – namely, command interpretation or reflection – that provides access to any other functionality at runtime.

For the Jx type of malicious code snippet, we defined additional subcategories that further characterize the nature of connections between security-critical methods. These definitions are built on the notion of *junction tree*, which is in turn defined as a tree that is contained in the call graph and has, as leafs, the nodes representing security-critical methods from a specific set under consideration. The root of such a tree we call *junction*. Informally, it is the meeting point for the paths starting from each security-critical method, following the edges in reverse order using a breadth-first search algorithm (some examples can be found in the Figure 1). Depending on the height of the tree and the number of distinct package and package groups to which their methods belong (characteristics that denote the topological nature of each junction), four classes of junctions can be defined as follows:

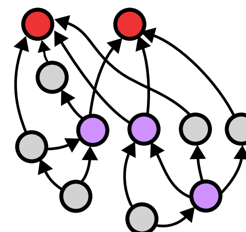


Figure 1. A simple graph illustrating the definitions of junction: relatively to the security-critical methods in red, the vertices in purple represent junctions.

- **J0**: A tree of unitary height, corresponding to malicious functionality implemented inside a single method, from which all calls to security-critical methods are made.
- **J1**: A tree of height greater than one, but whose nodes all belong to a single package containing the calls to the security-critical methods.
- **J2**: A tree of height greater than one whose nodes belong to a single package group but not to a single package. This corresponds to malicious functionality implemented in a single component, from which the calls to security-critical methods are made.
- **J+**: A tree of height greater than one, whose nodes do not belong to a single package group. This corresponds to malicious functionalities implemented by the use of different components, possibly from different libraries or frameworks, from which the calls to security-critical methods are made.

For characterizing program parts and the connections between them, we employ the features defined below. Figure 2 provides examples of all the features. The first four features are, in nature, local (i.e., characterize individual program parts), while the last two are aimed at characterizing more high-level, global aspects of a program.

1) **Incidence**: the number of calls made from a part to methods of a specific Java standard API, expressed both as absolute or, to normalize the feature in relation to the size of the part, relative numbers. Figure 2 (a) shows, relatively to the `net` API (which provides network access functionalities), the incidence of the depicted program part onto the API: 4 in absolute terms, or 0.5 (4/8) in relative terms.
2) **Dispersion**: defined as the number of methods that, within a program part, make calls to a specific standard API. It may also be expressed in absolute or relative terms. Figure 2 (a) shows the dispersion, within a program part, of the

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9

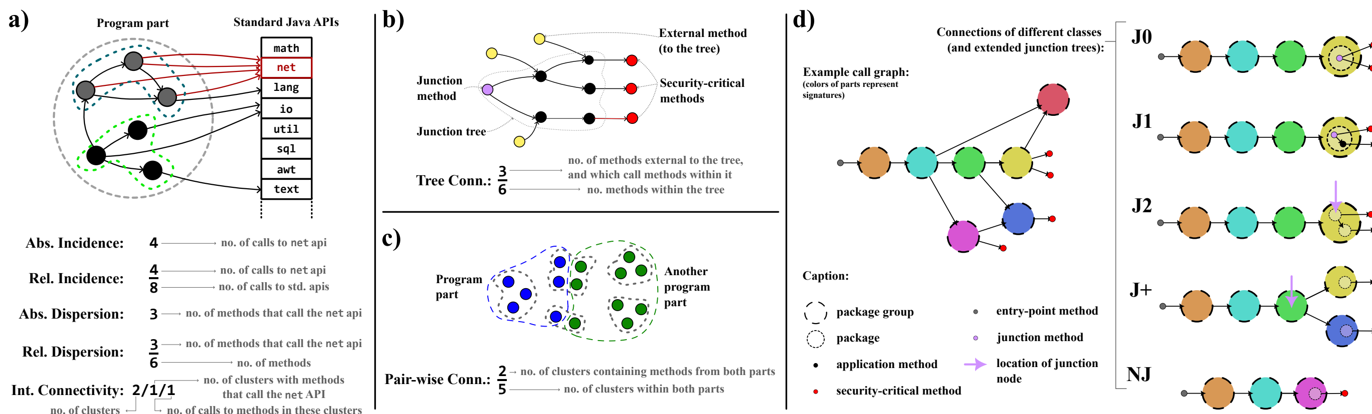


Figure 2. Examples of topological features used to characterize connections of security-critical methods to program parts, and with each other via these parts.

methods that call the `net` API (nodes in gray): in absolute terms, 3, or 0.5 (3/6), in relative terms.

3) **Internal Connectivity**: defined as an indicator of association from a given set of methods in a program part to the rest of the part. It can be obtained by applying a clustering algorithm to the part and then combining: (1) the number of clusters that contain methods from the set; (2) the number of connections between clusters that contain the methods and clusters that do not. Figure 2 (a) shows the internal connectivity of the set of methods that use the `net` API within a program part, expressed as 2/1/1: two clusters in total, one cluster (as defined by the dashed blue line) containing all the methods from the set, and one call to the latter cluster from the other cluster (as defined by the dashed green line).
4) **Tree Connectivity**: expressed as the number of calls made to the methods belonging to a junction tree from methods not in it. Figure 2 (b) shows that, for the depicted junction tree, whose root is the node in purple and which connects the three security-critical methods in red, the tree connectivity is 3, expressing the number of methods that do not belong to the tree (in yellow) but call methods that do. Relatively to the number of elements in the tree, the value of its connectivity is 0.5.
5) **Pair-Wise Connectivity**: expresses how much two program parts are coupled. This feature is defined as a function of both the number of connections between the parts (in absolute or relative numbers), and, once a clustering algorithm is applied to the nodes of both parts, the number of clusters that contain nodes from both parts. Figure 2 (c) shows the pair-wise connectivity between two program parts (depicted in green and blue), which can be expressed as 0.4 (2/5): the number of clusters that contain nodes from both parts (2), among the total number of clusters (5).
6) **Incidence Signatures**: defining the signature of a program part as its set of incidences onto every standard API, a sequence of signatures can be associated with a junction tree as a function of the packages and package groups to which the tree's vertices belong. Likewise, extending the tree backwards into the call-graph, up to the point where the program's expected entry-points are reached, similar signatures can be extracted from the program parts to which the tree is connected (even for the NJ category, if the method that calls the security-critical method is taken as the root of a "virtual" junction tree). In the Figure 2 (d), for an arbitrary graph as depicted, the pictures on the right illustrate, for all possible categories of connections, the path from an entry-point to the security critical methods, and where the junction is located (except for the NJ category). The colors of the program parts represent their incidence signatures, and the typical connections between parts of different signatures represent the most high-level topological property used in our technique.

To build a malicious code snippet detector based on these features, providing both accuracy and robustness against evasion, we rely on the following hypotheses (one for each feature) regarding the normal occurrence of calls to security-critical methods within program parts:

1) The calls are made from parts that have minimal values of incidence onto the APIs containing security-critical methods. In other words, these calls are not exceptions in the parts in which they are made.
2) In these parts, there is a minimum dispersion value associated with calls to the APIs containing the security-critical methods, i.e., there must be at least a few methods that make such calls.
3) There are minimum values of connectivity, within a program part, between the sub-parts in which calls to security-critical methods are made and other sub-parts. In other words, these calls are not made from isolated subparts.
4) For calls associated with junction trees of non-unitary height (i.e., J1, J2 and J+), there are minimum values of connectivity associated with them – i.e., the trees are not isolated in the program parts that contain them.
5) In the paths that begin at the programs' entry-points and lead to a part in which calls to security-critical methods are made, there are minimum values of pair-wise connectivity between each consecutive part involved in the path. This implies that, in each of the parts, a call to the next one in the path is not an exception.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

6) From the paths that begin at the programs' entry-points to the parts from which calls to security-critical methods are made, there are expected sequences of signatures among the parts involved in the path. Those sequences can be recognized by Graph Learning Algorithms, such as Graph Neural Networks (GNNs).

A detector combining these features should put an extra burden on SSC attack attempts, which would have to obey high-level topological properties of programs to go undetected. In other words, even though the proposed technique is unlikely to completely prevent the insertion of the malicious excerpts considered in this work (derived from [5]), it is expected to increase the cost of SSC attacks by requiring the adoption of more advanced evasion techniques. In particular, assuming a white-box model (i.e., that attackers know the expected feature values in the program), evasion may be possible by inserting spurious calls into the target parts, having to take extra care to ensure that these insertions are not flagged by dead-code detection techniques. Attackers may also graft, into the call graph of the attacked component, a whole part that naturally meets the expected feature values – for example, by extracting it from some legitimate library and contaminating it with the malicious excerpt. In that case, however, the model of expected signature sequences for analyzing more high-level program features may be helpful in identifying such a malicious implant. Ultimately, threat models that consider the percentage that an attacker controls of the call graph of a program may be defined to analyze security guarantees provided by our method.

III. Preliminary Results

So far, we have implemented a basic version of our detection technique. It was designed to detect only the Jx case of malicious code excerpts, specifically the J0, J1 and J2 cases. The detection of those cases provides validation for the most basic hypotheses underlying our technique: the correlation between incidence and the presence of junctions within the program parts. Moreover, as those cases represent 98% of the instances found in the BKC dataset, they can be considered the most natural choice as the first detection target.

Our implementation relies on the Wala framework [7] to build the call graph of an input program from its bytecode. The framework was configured in a way that reduces computational costs (therefore, possibly not using the most precise static analysis algorithms available in WALA). Two graph processing programs were developed to analyze the call graph: the first extracts from packages and package groups the values of incidences onto standard APIs; and the second searches for natural occurrence of junctions connecting different combinations of security-critical methods, each from three possible package groups from native Java APIs: (1) `net`, which aggregates methods related to network access; (2) `io`, containing methods related to file input and output; and (3) `lang`, which contains methods related to command interpretation and reflection. The combinations were generated in such a way as to represent the largest possible number of implementation possibilities for the malicious behaviors examined in this work, considering Java 1.6 APIs (the latest version compatible with our validation dataset). The data processing step was performed manually, with the help of common spreadsheet software. All the software and data are available in our public repository [8].

The dataset used in our test was XCorpus [6]. Despite being relatively old, this choice was motivated by the fact that it is a curated dataset of compilable and executable programs and program components, which enables the construction of more complete call graphs (as the dependencies of all instances have been resolved). The dataset contains 6694 unique packages, which were manually grouped into 372 package groups by name similarity. The values of minimal incidence were adjusted manually, aiming to maximize detection accuracy considering a scenario of random contamination of packages and package groups. Table I displays the resulting values of minimal incidence associated with junctions connecting security-critical methods within program parts (**J0** or **J0+J1** within packages, or **J0+J1+J2** in package groups), each method belonging to the `io`, `lang` or `net` APIs. The symbols $\theta_I^{(1)}$,$\theta_R^{(1)}$ and $\theta_I^{(2)}$,$\theta_R^{(2)}$ denote the adjusted values of absolute and relative minimal incidences within packages and package groups, respectively. Table I also displays the detector's accuracy and average F_1 score ($\overline{F_1}$) across different synthetic contamination scenarios.

In our tests, we found that the natural occurrence of junctions for classes **J0**, **J1** and **J2** is itself rare. The incidence values displayed in Table I show that, in most cases, junctions of class **J0** are linked to higher relative incidences (and also, to a lesser extent, absolute incidences) than junctions of class **J1** are. These adjusted parameters also show that relative incidences in the packages are generally higher than in package groups, indicating, as expected, that bigger program parts contain code implementing more diverse functionalities. Lastly, these parameters also show that junctions of classes **J2** are associated, with some exceptions, to lower values of relative incidences in comparison to junctions of classes **J1** and **J0**.

The detection accuracies achieved were very high, ranging from 86.3% to 99.98%, and $\overline{F_1}$ ranged between 0.806 and 1. The false positive rate did not exceed 12% in the worst case scenario, indicating the suitability of the method for real-world settings – in which it is very important not to overwhelm human analysts with false alerts. We also performed a simple ablation study, and found that no single parameter contributes to more than 8% to the accuracy in any setting.

IV. Limitations and Resistance to Evasion

Beyond the evasion tactics discussed in Section II, attackers may also employ other tactics, somewhat orthogonal to the former ones, to try to avoid detection. One category of strategies may exploit the limited accuracy of static analysis methods, which causes the construction of incomplete call graphs. An exceptionally thorny case involves Java frameworks, which make heavy use of reflection and annotations to link components at runtime, whereas static analysis tools are not inherently capable of inferring those links. Moreover, as some API implementations are not distributed with applications (for example, because they are part of containers, which serve as

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

TABLE I. PARAMETERS OF MINIMAL INCIDENCES ASSOCIATED WITH THE PRESENCE OF JUNCTIONS WITHIN PROGRAM PARTS, AND THE DETECTOR'S MINIMAL ACCURACY AND $\overline{F_1}$.

	io+net combinations						lang+net combinations						lang+io+net combinations								
	J0		**J0+J1**		**J0+J1+J2**		**J0**		**J0+J1**		**J0+J1+J2**		**J0**			**J0+J1**			**J0+J1+J2**		
	io	net	io	net	io	net	lang	net	lang	net	lang	net	lang	net	io	lang	net	io	lang	net	io
$\theta_I^{(1)}$	14	7	26	6	–	–	48	13	48	10	–	–	144	24	28	144	24	28	–	–	–
$\theta_R^{(1)}$	4.6%	1.6%	0.6%	1.65%	–	–	8.5%	0.5%	4.2%	0.5%	–	–	33%	2.5%	6.8%	11%	0.3%	1.73%	–	–	–
$\theta_I^{(2)}$	41	17	41	6	62	39	1554	70	583	22	1329	68	11224	725	1618	1329	88	312	1329	40	189
$\theta_R^{(2)}$	2.2%	0.6%	0.1%	1.94%	1.7%	0.35%	9.8%	0.4%	7.8%	0.2%	10%	0.17%	18.6%	1.1%	2.4%	13.6%	0.3%	2.4%	9.1%	0.3%	1%
Acc.	99.2%		97.8%		88.1%		98.3%		95.8%		86.3%		99.98%			99.6%			86.5%		
$\overline{F_1}$	0.993		0.986		0.902		0.985		0.960		0.806		1.00			0.995			0.880		

additional middleware layers), the links established through these APIs are missed by standard static analysis tools. To circumvent these limitations, our technique may be equipped with static representations of frameworks, such as the one found in [9], to allow for more complete call graphs. Another alternative would be to analyze features at runtime, once all connections between program parts are resolved.

Another possible category of evasion tactic involves the obfuscation of package names. In this case, our analysis would require more robust methods to group packages, using not their names but the topological features themselves. Graph similarity techniques may also be used to identify known instances from a dataset of components.

One last evasion technique category of involves aspects of Adversarial Machine Learning, specifically for the possible use of GNNs to identify high-level topological features. In this case, adversarial training and other techniques may be used to make the detection more robust [10].

V. RELATED WORKS

Many methods in the literature to detect malicious code in Java programs and related technologies also guide the analysis by the usage of security-critical methods. The earliest one, to the best of our knowledge, is Jarhead [11], aimed at the detection of malicious applets. For the detection of malicious code in OSCs, [12] investigates simple possible indicators of malicious behavior in JARs. One indicator that the authors conclude to perform is the usage of security-critical methods in a single method. Another related work is Shear [13], which employs program slicing techniques guided by paths between security-critical methods. The slices are then used to train a supervised deep-learning detector. Dapasa [14], in turn, aims to identify piggybacked Android software. The solution relies on noticeable differences between malicious and piggybacked applications in the usage of security-sensitive APIs, guiding their analysis by the concept of sensitive subgraphs. Finally, [15] employs Class-Dependency Graphs to identify disjoint dependency regions in the applications under analysis, noting that isolated regions that make singular use of security-critical methods may be identified as malicious.

Despite sharing similarities with our work, none of these studies analyze high-level topological features of programs, as we do, to guide an unsupervised detection approach.

VI. CONCLUSION AND FUTURE WORK

We present a novel technique for the detection of malicious code embedded in (Java) OSCs through SSC attacks. We follow an anomaly detection approach, using local and high-level topological features to distinguish normal occurrences of connections between security-critical methods and program parts, and between those methods through program parts. The technique was implemented partially, and the tests conducted so far indicate that it can achieve high accuracy considering common code excerpts observed in SSC attacks [5].

As future work, we intend to implement the technique in its entirety, and validate it with a dataset containing more recent versions of Java programs. We also plan to implement and test some of the robustness improvements discussed in Section IV.

REFERENCES

[1] The Linux Foundation, "A summary of census ii: Open source software application libraries the world depends on", 2022, Accessed: 2026-04-16. [Online]. Available: https://www.linuxfoundation.org/blog/blog/a-summary-of-census-ii-open-source-software-application-libraries-the-world-depends-on.

[2] Synopsys Inc., "2025 open source security and risk analysis", 2025, Accessed: 2026-02-19. [Online]. Available: https://www.blackduck.com/resources/analyst-reports/open-source-security-risk-analysis.html.

[3] Sonatype Inc., "Open source malware at the gate: The evolving software supply chain attack surface", 2026, Accessed: 2026-04-16. [Online]. Available: https://www.sonatype.com/state-of-the-software-supply-chain/2026/open-source-malware.

[4] M. Ohm and C. Stuke, "Sok: Practical detection of software supply chain attacks", in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES '23, Benevento, Italy: Association for Computing Machinery, 2023.

[5] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks", in *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cham: Springer International Publishing, 2020, pp. 23–43.

[6] J. Dietrich, H. Schole, L. Sui, and E. Tempero, "Xcorpus - an executable corpus of java programs", *Journal of Object Technology*, vol. 16, no. 4, 1:1–24, Aug. 2017.

[7] J. C. S. Santos and J. Dolby, "Program analysis using wala (tutorial)", in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, p. 1819.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

[8] O. Paiva, W. Ruggiero, and M. Simplicio, “Towards the static detection of compromised software components by topological anomalies (repository)”, 2026, Accessed: 2026-04-21. [Online]. Available: https://github.com/ozpaiva/static-topol-analysis.

[9] M. Chen, T. Tu, H. Zhang, Q. Wen, and W. Wang, “Jasmine: A static analysis framework for spring core technologies”, in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22, Rochester, MI, USA: Association for Computing Machinery, 2023.

[10] E. Tabassi, K. J. Burns, M. Hadjimichael, A. D. Molina-Markham, and J. T. Sexton, “A taxonomy and terminology of adversarial machine learning”, *NIST IR*, vol. 2019, no. 1-29, p. 1, 2019.

[11] J. Schlumberger, C. Kruegel, and G. Vigna, “Jarhead analysis and detection of malicious java applets”, ser. ACSAC ’12, Orlando, Florida, USA: Association for Computing Machinery, 2012, pp. 249–257.

[12] P. Ladisa, H. Plate, M. Martinez, O. Barais, and S. E. Ponta, “Towards the detection of malicious java packages”, in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED’22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 63–72.

[13] Y. Zeng et al., “Wolf in sheep’s clothing: Shearing the camouflage of malicious java components in maven”, *IEEE Transactions on Software Engineering*, vol. 51, no. 10, pp. 2847–2863, 2025.

[14] M. Fan et al., “Dapasa: Detecting android piggybacked apps through sensitive subgraph analysis”, *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 8, pp. 1772–1785, 2017.

[15] K. Tian, D. Yao, B. G. Ryder, G. Tan, and G. Peng, “Detection of repackaged android malware with code-heterogeneity features”, *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 1, pp. 64–77, 2020.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-398-9