

Deductive Software Architecture Recovery via Chain-of-thought Prompting

Satrio Adi Rukmono
Eindhoven University of Technology
Eindhoven, The Netherlands
Institut Teknologi Bandung
Bandung, Indonesia
s.a.rukmono@tue.nl

Lina Ochoa
Eindhoven University of Technology
Eindhoven, The Netherlands
l.m.ochoa.venegas@tue.nl

Michel R.V. Chaudron
Eindhoven University of Technology
Eindhoven, The Netherlands
m.r.v.chaudron@tue.nl

ABSTRACT

As software evolves, software architecture recovery techniques can help for effective maintenance. We envision a *deductive software architecture recovery* approach supported by Large Language Models (LLMs). Unlike existing *inductive* (bottom-up) recovery techniques, which reconstruct architecture by considering the properties observed at implementation level, our top-down approach starts with architectural properties and seeks their manifestations in the implementation. It employs a known Reference Architecture (RA) and involves two phases: RA definition and code units classification. A proof-of-concept with GPT-4 emulates deductive reasoning via chain-of-thought prompting. It demonstrates the deductive SAR approach, applying it to the Android application K-9 Mail and achieving a 70% accuracy in classifying 54 classes and 184 methods. The future plans focus on evaluating and refining the approach through ground-truth assessments, deeper exploration of reference architectures, and advancing toward automated human-like software architecture explanations. We highlight the potential for LLMs in achieving more comprehensive and explainable software architecture recovery.

CCS CONCEPTS

• Software and its engineering → Software maintenance tools.

KEYWORDS

software architecture, software architecture recovery, deductive SAR, chain-of-thought prompting

ACM Reference Format:

Satrio Adi Rukmono, Lina Ochoa, and Michel R.V. Chaudron. 2024. Deductive Software Architecture Recovery via Chain-of-thought Prompting. In *New Ideas and Emerging Results (ICSE-NIER'24)*, April 14–20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3639476.3639776>

1 INTRODUCTION

During the evolution of software, the source code typically changes in ways that violate the integrity of the architecture of a system. This results in degradation in the form of drift or erosion [21]. To

detect such degradation, one needs to compare a representation of the system's *intended* architecture to the implementation.

Link et al. define Software Architecture Recovery (SAR) as “the process of recovering a system's architecture from its implementation artifacts, such as its source code” [11]. SAR is of special importance when documentation about the system's architecture is outdated, incomplete, or simply lacking [11, 19]. To the best of our knowledge, current SAR techniques follow what we call an *inductive recovery*, where certain facts from the source code (e.g. software components, control- and data-flow dependencies, textual input) are extracted and clustered to recover the structure of the *current* architecture [1–3, 8, 16, 23]. However, we posit that to manage the deterioration of a system's architecture—and consequently rectify its evolution—one must follow a *deductive recovery* approach. That is, a known reference architecture [22] should be provided to assess how the current implementation deviates from the intended system architecture design.

In our approach, the *Reference Architecture* (RA) is an abstract representation of key aspects of an architectural design. Our RA considers two viewpoints: the *component viewpoint*, which is concerned with the structure and grouping of the elements, and the *interaction viewpoint*, which focuses on the interaction and coordination among components [22].

To illustrate the rationale for our deductive approach, consider the notion of layering in architecture. Existing techniques for recovering architectural layers involve analyzing a system's dependency graph to look for classes that no other class depends on, designating them as top-layer classes. However, when a human developer is getting acquainted with an unfamiliar system that employs a layered architecture, they often start by recognizing that the top layer in such system typically corresponds to the presentation layer, which comprises classes responsible for User Interface (UI) widgets such as buttons and text fields [22]. Hence, when a developer encounters a class that deals with UI buttons, they can deductively infer that the class belongs to the top layer. This approach thus relates to the observation that many existing recovery techniques “do not reflect the way engineers actually map entities to components” [8]. In particular, there is an aspect of the semantics of the implementation that current recovery techniques do not take into account.

To bridge the aforementioned gap, we present in Section 2 our vision of *deductive software architecture recovery*, an approach that classifies code units (e.g., methods, classes) according to an RA. We demonstrate our vision via a proof of concept in Section 3 that utilizes transformer-based language models and chain-of-thought prompting to emulate deductive reasoning. We apply the deductive



This work licensed under Creative Commons Attribution-NonCommercial International 4.0 License.

ICSE-NIER'24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0500-7/24/04.
<https://doi.org/10.1145/3639476.3639776>

SAR approach to the Android application K-9 Mail to obtain preliminary results on the feasibility of the approach in Section 4. Future research directions and conclusions are elaborated in Section 5 and Section 6.

2 VISION: DEDUCTIVE SOFTWARE ARCHITECTURE RECOVERY

In this section, we describe the general approach for deductive software architecture recovery. The approach is depicted in Figure 1. It is structured into two phases: the *RA definition* phase and the *code units classification* phase. These phases, along with their corresponding steps, are described below.

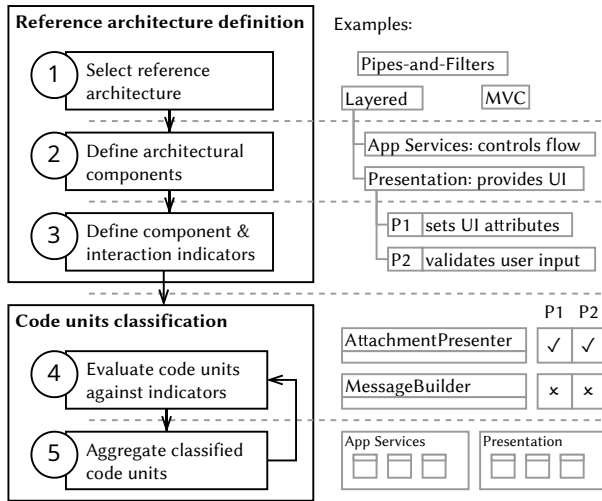


Figure 1: The general approach for deductive SAR.

2.1 Phase I: Reference Architecture Definition

The first phase of deductive SAR consists of defining the RA to be used in the recovery process. The RA provides a blueprint to classify and organize the architectural elements recovered from the system's source code. We emphasize that by reference architecture, we do not mean an actual instance of architecture with specific components—rather, it is an abstract description of how architectural elements are organized.

Step 1: Select the reference architecture. The RA that applies to a particular software system is intended to come from one of the system's architects or engineers. In case this is not available, it may suffice to select one of the more common RA—e.g., layered architecture, pipes-and-filters [22], and model-view-controller (MVC) [5]. The approach applies broadly to any reference architecture, as long as one can describe it in terms of components and interactions between components, as elaborated in the next steps.

Step 2: Define the architectural components. This step focuses on the *component viewpoint* of the RA definition. The natural course of action is, therefore, to define the distinct architectural components that are to be used as a reference for later code unit classification. Note that the *component* terminology pertains to the general organization of elements as suggested by Wirfs-Brock et al. [22]. For

example, in a layered architecture, the components include, e.g., *presentation layer* and *application services layer*, while in an MVC architecture, the components are *models*, *views*, and *controllers*.

Step 3: Define component- & interaction-indicators. To complete the definition of the RA, this step focuses on how architectural components and interactions between them are manifested in source code. In particular, each architectural component (defined in *Step 2*) holds a specific responsibility that also defines how components interact with each other. This step, thus, involves breaking down these responsibilities and interactions into a set of specific indicators that source code units (e.g., methods, classes) corresponding to the particular architectural component may exhibit. The indicators may come from best practices of the particular RA. They can also depend on the specific technology stack used to implement the system.

2.2 Phase II: Code Units Classification

In this phase, source code units are evaluated and classified into an architectural components-type as defined in the *RA definition* phase. The results are then aggregated, and this process cascades to code units defined at a higher level of abstraction until the system's architecture is recovered:

Step 4: Evaluate source code units against indicators. In this phase, low-level source code units are classified into the component-types specified in *Step 2*. A *low-level source code unit* is defined as a source code construct at the lowest level of abstraction for analytical purposes. Instances of low-level units can include statements, methods, or classes. To perform the classification, the approach evaluates whether the syntax and semantics of these units align with the *indicators* defined for each architectural component. This shows the *deductive* nature of our recovery approach: we start with a set of architectural properties and try to find evidences within the implementation, rather than starting with properties exhibited by the implementation. We leave the details of the compliance analysis non-specific here, firstly, because the details may vary depending on each indicator and RA, and secondly, to keep it open to the application of diverse techniques. However, as we demonstrate in the coming sections, the idea of deductive SAR is sparked and made effective by transformer-based language models, specifically via the chain-of-thought prompting technique, which entices deductive pseudo-reasoning in LLMs via step-by-step “thinking” process.

Step 5: Aggregate the classification results. After evaluating and classifying individual source code units, the subsequent task involves aggregating this information to classify more abstract compositions of source code units. For instance, if the preceding step involved classifying methods, this current step entails aggregating that information to classify classes. Subsequently, this aggregation process can extend to higher abstraction levels, facilitating the construction of a comprehensive depiction of the system's architecture.

In the following section, we showcase an instance of the deductive SAR approach and its application to an open-source system.

3 PROOF OF CONCEPT

Recently, Large Language Models (LLMs) have been applied to support different coding-based tasks in the software engineering practice. Examples of such tasks include defect and clone detection, code comprehension, code summarization, among others [13].

However, software engineering is not limited to coding and this technology can be leveraged to perform effective software design and maintenance. In this section, we demonstrate how we leverage GPT4, a transformer-based language model, to emulate deductive reasoning in deductive SAR, and support the RA definition using natural language.

In this proof of concept, we aim to recover the architecture of the Android email application, K-9 Mail. We first select a *layered architecture* (**Step 1**). We follow a description of layered architecture [22] that includes the following architectural components and their responsibilities (**Step 2**): (1) the *presentation layer* [Pr] provides interfaces to interact with the user; (2) the *application services layer* [Ap] controls flow and coordinates responses to events; (3) the *domain services layer* [Do] provides information and services related to the system’s problem domain, and; (4) the *technical services layer* [Te] connects to external devices and programs.

For **Step 3**, we elect to define the architectural control indicators at the *method level*, i.e., each indicator describes a characteristic of a typical *Java method in an Android application* for an architectural component. To illustrate this, an excerpt of the indicators we use for the proof-of-concept is shown in Table 1. Thanks to the LLM, the indicators can be specified in natural language without the need of depending on any sort of domain specific language. However, the results of the recovery process might be hindered if the RA is not properly defined.

Table 1: Sample indicators for components of layered architecture in an Android email client.

№	Indicator (A method in this layer ...)
<i>Presentation Layer</i> [Pr]	
Pr ₁	... sets the attributes of UI components, e.g., sets the text of a TextView.
Pr ₂	... notifies listeners about user events, such as button clicks or list item selections.
Pr ₃	... transforms domain objects into visual representations.
Pr ₄	... performs validation on user input.
<i>Technical Services Layer</i> [Te]	
Te ₁	... interacts with databases or other persistence services, e.g., inserts a row into an SQLite database.
Te ₂	... performs network operations, e.g., sends a request to an email server.
Te ₃	... interacts with the file system, e.g., saves an email attachment to disk.
Te ₄	... uses Android’s hardware-related APIs, e.g., checks if the device is connected to the internet.

As we move on to the code units classification phase, we digress from the details of the steps to discuss the role of static analysis and LLMs for achieving the classification. In classical SAR techniques, analysis of software units is performed based on information extracted from static analysis—such as metrics and dependency graphs. In our approach, indicators are phrased in natural language and can be either generic or application-specific.

Relying on static analysis for our classification would lead into roadblocks. For example, checking for code statements that write

data to a file [Te₃] is possible (e.g., in Java: statements that call the methods of `BufferedWriter`), but some indicators are harder to implement as static analysis—e.g., there can be countless ways that domain objects can be transformed into visual representations [Pe₃]. Even with the indicators that can be implemented as static analysis, there can be variations, and they may depend a lot on the specific implementation technology, which may prevent good recall. The Te₃ example above will not work, for instance, when the class in question depends on another class that wraps `BufferedWriter` instead of using it directly.

On the other hand, LLMs are trained with vast amounts of data—including source code from various programming languages, frameworks, and problem domains. This makes it work particularly well for classification and clustering tasks [6], providing opportunities for LLM utilization to evaluate source code units against the sets of indicators. In particular, our approach benefits from the ‘semantic clustering’ in LLMs that manifests from the notion of similarity in embedding models that they utilize [7, 12].

We return to the classification phase in our proof of concept. For **Step 4**, we perform the classification of Java methods with the help of OpenAI’s GPT-4 model [14]. We construct LLM prompts asking to evaluate Java methods against each control indicator in a chain-of-thought manner [20]. Every single prompt includes: (1) information about the system and its problem domain, (2) the qualified name of the class that contains the method, (3) the method’s source code, (4) the architectural layer being evaluated, including a description of its responsibility, and (5) all control indicators for the architectural layer. The prompt continues with a request to check whether the method satisfies each indicator and specify the reasoning. For ease of output parsing, we ask the LLM to end its response with a boolean list corresponding to the “verdict” for each indicator. The prompt template that we use can be found in Figure 2. We ask the LLM to end its response in a specific format for ease of parsing.

In a layered software architecture, one of the layers is the *(layer_name)* layer, which *(layer_responsibility)*. Consider the context of an Android Java project “*(project_name)*”: *(project_domain_description)*

Here are some indicators that a Java method in the project may belong to a class in the *(layer_name)* layer:

(layer_indicators)

The class ‘*(class_name)*’ contains the method ‘*(method_name)*’:

(method_source_code)

Check whether this method satisfies each indicator above. Mention the specific line of code that supports your reason. At the very last line, write the boolean verdicts separated by a comma, e.g., ‘true, true, false, true’. If indeterminate, say ‘false’.

Figure 2: Prompt template for method classification.

Finally, our proof of concept ends with **Step 5**, in which we aggregate the method evaluation results at the class level. We do this by counting how many times each indicator appear in a class. We then decide the layer for each class based on which indicated layer dominates the class.

4 PRELIMINARY RESULTS AND ANALYSIS

To initiate the evaluation of our approach, we implemented the technique on a subset of K-9 Mail, version 5.304¹. We randomly selected 54 out of 779 classes to apply deductive SAR. We commenced by defining our own reference layered architecture for each selected class through manual inspection of their source code. Then, we extracted the source code of the public methods from each class, totaling 184 methods. Subsequently, we executed the LLM request for each method and architectural layer. We then tallied the number of times an indicator is exhibited by the methods of each class and assign the class to the architectural layer with the most indicator occurrences. Notice that in different implementations, engineers can set the classification thresholds based on their own criteria.

In the K-9 Mail case, we found that each Java package contains classes either from a single layer, or two adjacent layers. This suggests a good source code organizational choice. The relative comparison between our automated classification and the manual one is presented in Figure 3. It exhibits an overall accuracy—fraction of corrected classified cases—of 70%, with additional classification performance metrics provided in Table 2. In cases with tied aggregates, i.e., code units that are classified into two different layers, we consider them to be true positives as long as one of the predicted label matches the manually classified layer.

		Automated classification				Total
		Pr	Ap	Do	Te	
Manual classif.	Pr	8	1	1	1	11
	Ap	4	7	2	0	13
	Do	0	0	13	0	13
	Te	1	2	4	10	17

Figure 3: Confusion matrix for sampled classes of K-9 Mail.

Table 2: Summary of the preliminary experiment results.

	Average	Layer			
		Pr	Ap	Do	Te
Precision	72%	62%	70%	65%	91%
Recall	71%	73%	54%	100%	59%
F1-score	69%	67%	61%	79%	71%

5 FUTURE PLANS

To reach our vision, we have defined a research roadmap that focuses on the following three main aspects.

Evaluation of the current approach. As stated by Garcia et al., SAR techniques usually suffer from inaccuracies that are hardly detected due to the lack of reference architectures [9]. Reflecting the exact intended architecture of a system, as envisioned by its architect(s), is a daunting task. Garcia et al. proposed to recover the so-called ground-truth architectures. These architectures are obtained by performing a SAR technique that extracts an *authoritative*

architecture—an architecture generated without the involvement of the system’s experts—and then putting it under the scrutiny of the system’s engineers to correct or complete it. We aim to reuse the existing ground-truth architectures of four open-source systems (i.e., Apache Hadoop, Bash, ArchStudio, and Apache OODT) [9] to perform an accuracy evaluation of the output of our approach. Additionally, we plan to conduct industrial case studies. We aim for qualitative studies in the form of *field experiments* [18] where the systems’ experts are consulted to validate the recovered architecture and the followed process. If the studied companies consent, these recovered architectures can be fed back into the set of ground-truth architectures available for future research.

Research on reference architectures. Our proposed technique relies on the definition of a reference architecture. In the future, we plan to dive deeper into this area and design a “gold standard” RA—including its components, interactions, and indicators—that works reasonably well for general-purpose software architecture recovery. The existence of such reference can then eliminate phase I for the majority of cases. We intuit that layered architecture is a good base for a gold standard, because, regardless of the specific architecture, software systems can always be divided into *parts that communicate with clients* (e.g., users or other systems) and *parts that communicate with the hardware*.

Software architecture explanation. The ultimate step in this research line is to be able to provide software architecture explanations in a human-like manner. This approach aims at addressing the well-known problem of incomplete, dispersed, or dated documentation of a system [2, 4, 17]. We advocate extending the *on-demand developer documentation* proposed by Robillard et al. [15] to an *on-demand software architecture explanation*. We speculate that such explanations should maintain the question-answering style that arises naturally in certain software engineering scenarios, such as *onboarding processes*, where an engineer needs to quickly become familiar with the system to be productive [10], or during an *architecture conformance check*, where the alignment of the current architecture with the intended design decisions is subject to examination.

6 CONCLUSIONS

We have presented our vision for deductive software architecture recovery and its concretization using LLMs. LLMs favor the definition of a reference architecture using natural language and help exploiting both syntactic and semantic aspects of the system’s implementation to recover its architecture. The deductive SAR approach shows promising results after applying it to a sample of 54 classes in the K-9 Mail application spanning 184 Java methods. Our approach exhibits a 70% accuracy for this specific case. However, further evaluation needs to be performed using ground-truth- and industrial-architectures. We, therefore, expect our approach to benefit the field of software architecture recovery by aligning better with how software is designed, and allowing explainable software architecture recovery: when deciding if a source code unit complies with a particular architectural component, our indicators written using natural language are used to explain *why*. This is a step forward in the quest of providing software architecture explanations in a human-like manner.

¹<https://github.com/thundernest/k-9/tree/5.304>

REFERENCES

- [1] Alvine Boaye Belle, Ghizlane El Boussaidi, and Sègla Kpodjedo. 2016. Combining Lexical and Structural Information to Reconstruct Software Layers. *Information and Software Technology* 74 (2016), 1–16. <https://doi.org/10.1016/j.infsof.2016.01.008>
- [2] Tingting Bi, Peng Liang, Antony Tang, and Chen Yang. 2018. A Systematic Mapping Study on Text Analysis Techniques in Software Architecture. *Journal of Systems and Software* 144 (2018), 533–558. <https://doi.org/10.1016/j.jss.2018.07.055>
- [3] Ghizlane El Boussaidi, Alvine Boaye Belle, Stéphane Vaucher, and Hafedh Mili. 2012. Reconstructing Architectural Views from Legacy Systems. In *2012 19th Working Conference on Reverse Engineering*. 345–354. <https://doi.org/10.1109/WCRE.2012.44>
- [4] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 2016. 10 Years of Software Architecture Knowledge Management: Practice and Future. *Journal of Systems and Software* 116 (2016), 191–205. <https://doi.org/10.1016/j.jss.2015.08.054>
- [5] John Deacon. 2009. Model-view-controller (MVC) Architecture. *Online*[Citado em: 10 de março de 2006.] <http://www.jdl.co.uk/briefings/MVC.pdf> 28 (2009).
- [6] Bosheng Ding, Chengwei Qin, Linlin Liu, Lidong Bing, Shafiq Joty, and Boyang Li. 2022. Is GPT-3 a good data annotator? *arXiv preprint arXiv:2212.10450* (2022).
- [7] Ábel Elekes, Martin Schäler, and Klemens Böhm. 2017. On the various semantics of similarity in word embedding models. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, 1–10.
- [8] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. 2013. A Comparative Analysis of Software Architecture Recovery Techniques. In *28th IEEE/ACM International Conference on Automated Software Engineering*. 486–496. <https://doi.org/10.1109/ASE.2013.6693106>
- [9] Joshua Garcia, Ivo Krka, Chris Mattmann, and Nenad Medvidovic. 2013. Obtaining Ground-truth Software Architectures. In *35th International Conference on Software Engineering*. 901–910. <https://doi.org/10.1109/ICSE.2013.6606639>
- [10] An Ju, Hitesh Sajjani, Scot Kelly, and Kim Herzig. 2021. A Case Study of Onboarding in Software Teams: Tasks and Strategies. In *IEEE/ACM 43rd International Conference on Software Engineering*. 613–623. <https://doi.org/10.1109/ICSE43902.2021.00063>
- [11] Daniel Link, Pooyan Behnamghader, Ramin Moazeni, and Barry Boehm. 2019. The Value of Software Architecture Recovery for Maintenance. In *Proceedings of the 12th Innovations on Software Engineering Conference (Formerly Known as India Software Engineering Conference)*. Association for Computing Machinery, New York, Article 17, 10 pages. <https://doi.org/10.1145/3299771.3299787>
- [12] Claudio Lucchese, Giorgia Minello, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, and Alberto Veneri. 2023. Can Embeddings Analysis Explain Large Language Model Ranking?. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 4150–4154.
- [13] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-trained Models of Source Code. *arXiv* (2023). <https://doi.org/10.48550/arXiv.2302.04026>
- [14] OpenAI. 2023. *GPT-4 technical report*. Technical Report. OpenAI. <https://doi.org/10.48550/arXiv.2303.08774>
- [15] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, Gail C. Murphy, Laura Moreno, David Shepherd, and Edmund Wong. 2017. On-demand Developer Documentation. In *IEEE International Conference on Software Maintenance and Evolution*. 479–483. <https://doi.org/10.1109/ICSME.2017.17>
- [16] Amir M. Saeidi, Jurriaan Hage, Ravi Khadka, and Slinger Jansen. 2015. A Search-based Approach to Multi-view Clustering of Software Systems. In *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*. 429–438. <https://doi.org/10.1109/SANER.2015.7081853>
- [17] Christoph Johann Stettina and Werner Heijstek. 2011. Necessary and Neglected? An Empirical Study of Internal Documentation in Agile Software Development Teams. In *Proceedings of the 29th ACM International Conference on Design of Communication*. Association for Computing Machinery, New York, 159–166. <https://doi.org/10.1145/2038476.2038509>
- [18] Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of Software Engineering Research. *ACM Trans. Softw. Eng. Methodol.* 27, 3, Article 11 (sep 2018), 51 pages. <https://doi.org/10.1145/3241743>
- [19] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. 2009. *Software architecture: foundations, theory, and practice*. Wiley Publishing.
- [20] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837.
- [21] Erik Whiting and Sharon Andrews. 2020. Drift and Erosion in Software Architecture: Summary and Prevention Strategies. In *4th International Conference on Information System and Data Mining*. Association for Computing Machinery, New York, 132–138. <https://doi.org/10.1145/3404663.3404665>
- [22] Rebecca Wirfs-Brock, Alan McKean, Ivar Jacobson, and John Vlissides. 2002. *Object Design: Roles, Responsibilities, and Collaborations*. Pearson Education.
- [23] Tianfu Yang, Zhiyong Jiang, Yanhong Shang, and Monire Norouzi. 2021. Systematic Review on Next-generation Web-based Software Architecture Clustering Models. *Computer Communications* 167 (2021), 63–74. <https://doi.org/10.1016/j.comcom.2020.12.022>

Received 14 September 2023; accepted 22 November 2023