Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 257 (2025) 834–841

Procedia Computer Science

www.elsevier.com/locate/procedia

The 8th International Conference on Emerging Data and Industry (EDI40), April 22-24, 2025, Patras, Greece

Automated UML Visualization of Software Ecosystems: Tracking Versions, Dependencies, and Security Updates

V. Kan[a], M. P. Lnu[a], S. Berhe[a,*], C. El Kari[a], M. Maynard[b], F. Khomh[c]

[a] *University of the Pacific, 3601 Pacific Ave, Stockton, CA 95211, USA*
[b] *Data Independence LLC, 23 Settlers Way. Ellington, CT 06029, USA*
[c] *Polytechnique Montréal, Montréal, QC, H3T 1J4, Canada*

Abstract

The growing complexity of software ecosystems—spanning multiple operating systems and their interconnected software components—poses significant challenges for documenting, visualizing, and maintaining these large systems over time. This complexity stems from the increasing number of components, their interdependencies, and rapid update cycles. In this work, we propose a release notes-driven approach that leverages Unified Modeling Language (UML), particularly component and package diagrams, to automate the visualization and monitoring of software architectures. Our method models peer relationships, stack dependencies, and hierarchical structures among components, addressing both architectural design clarity and cognitive scalability. By visually distinguishing critical information such as security updates (e.g., CVEs) and recent releases, our approach provides actionable information for software architects and engineers. We demonstrate practical use cases to highlight the effectiveness of our method in managing complex software ecosystems, enabling improved comprehension and decision-making within an ecosystem context.

Keywords: Software ecosystems; Unified Modeling Language (UML); Software architecture visualization; Release notes analysis; Component and package diagrams

1. Introduction

In the past decades, maintaining software architecture has primarily been considered an early phase software engineering task, largely confined to the design stage. It was traditionally treated as a static blueprint rather than a continuously evolving entity [1]. In recent years, faster update cycles, an increasing number of software components, their inter-dependencies, and security vulnerabilities have led to a change in the evolution of software architectures [2]. To date, maintaining an up-to-date software architecture essentially happens during all stages of the software

* Corresponding author. Tel.: +1-772-202-3743
E-mail address: sberhe@pacific.edu

10.1016/j.procs.2025.03.107

engineering process. Continuous updates, evolving dependencies, and security vulnerabilities make manual tracking impractical and error-prone [3]. This paper presents an approach to continously automate software architecture generation by integrating real-time version data and security updates [4]. By enriching architecture diagrams with the latest software releases and known vulnerabilities we aim to improve visibility and support the decision making across the software lifecycle. The work presented herein is a continuation of our previous efforts to improve the maintenance efforts of software ecosystems, specifically aiming the software update process. In 2020, our research explored how mining and clustering software updates data can be applied to support decision making regarding the timing of updates. Since then, our database has collected data on more than 80,000 software updates across more than 16,000 software components [5].

In 2023, we investigated methods to detect dependencies using a graph-based and tagging approach, highlighting the limitations of relying solely on software release notes for dependency detection [6]. This work led to the development of an impact-based approach in 2024, which leverages release notes classification. Users can define weighted impact attributes based on component type, update type, and other relevant attributes [7].

Our objective is to leverage component type information to understand and document the dependencies between operating system components and their respective installed components. We hypothesize that many software ecosystems are structured around architectures that include multiple interacting operating systems with various software components installed. Continuously documenting and maintaining such architectures is a challenging task, especially considering the frequency and independence of daily updates [8, 9]. Therefore, our goal is to automatically visualize current software architectural systems, including pending updates. This visualization aims to facilitate better documentation, engineering, reverse engineering, and maintenance practices, which is critical throughout the entire software engineering lifecycle. To achieve this, our approach emphasizes a UML-based visualization tool at various common stages of the software engineering process that presents a common use case of the Linux, Apache, MySQL, PHP (LAMP) stack application [10].

By integrating visualization as a central feature into each stage of our methodology, our goal is to ensure a comprehensive approach to operating and maintaining complex software ecosystem architectures. Towards the objective, the remainder of this paper is organized as follows: Section 2 presents related work, Section 3 details our methodology, Section 4 presents the prototype implementation, Section 5 discusses our results, concluding with final remarks in Section 6.

2. Related Work

Much work has already been done with regard to visualizing software architectures and tracking versions. In this section, we review and contrast related work (see Table 1.).

Luján-Mora, Vassiliadis, and Trujillo's "Data Mapping Diagrams for Data Warehouse Design with UML" introduces data mapping diagrams to improve the data warehouse design by capturing attribute-level mappings and transformations in UML. Their approach improves the precision of ETL process documentation and integrates conceptual, logical, and physical design aspects [11]. Bali and Sawant's "SoftArchViz: An Automated Approach to Visualizing Software Architectures" (2007) presents an automated approach to visualizing software architectures using data derived from source code analysis and architectural patterns, with the visualization tool focusing on generating interactive and dynamic views of software structures [12]. C. Narawita and K. Vidanage's research focuses on automating the generation of Unified Modeling Language (UML) diagrams from analyzed requirement texts using Natural Language Processing (NLP). The proposed system efficiently and accurately extracts elements for use case and class diagrams, targeting the design phase of software development. The automation aims to provide a quick, reliable, and intelligent solution for generating UML-based documentation, thereby saving time and budget for users and system analysts [13]. Yacheslav Lyashenko, Amer Tahseen Abu-Jassar, Vladyslav Yevsieiev, and Svitlana Maksymova (2023) proposed a model for a real-time monitoring and visualization system tailored for cyber-physical production systems. The work emphasizes the hardware implementation and integration of monitoring systems into existing production processes, addressing challenges with legacy equipment. The proposed solution aims to be cost-effective and adaptable, focusing on the seamless integration of sensors and software to improve production control [13]. Gamage (2023) addresses the challenges associated with manual diagram creation, such as human error and time consumption, by proposing an automated software architecture diagram generator using Natural Language Processing (NLP). This ap-

Table 1. Comparison of Related Work

Work and Authors	Data Source	Real-Time Update	Scope	Objective
Bali et al. (2007)	Source code analysis	No	Commercial and open-source software	Visualizing software structures
Ghanam et al. (2008)	Survey of tools	No	Open-source visualization tools	Survey of visualization tools
Hamza et al. (2012)	Feature sets	No	Pervasive software systems	Generating pervasive systems architectures
Narawita et al. (2016)	Requirement text	No	Commercial software systems	Automated UML diagram generation
Lyashenko et al. (2023)	Production data	Yes	Embedded and hardware-linked software	Monitoring and visualization
Gamage (2023)	Requirement text	No	Open-source and custom software	Automated architecture diagram generation
Renovate (2024)	Software repositories	Yes	GitHub open-source software	Automating dependency updates in ecosystems like GitHub
Our Work	Release notes	Yes	Any commercial or open-source software	Automated documentation and monitoring

proach leverages a custom Named Entity Recognition Model, a Masked BERT Relation Classifier, and a knowledge graph to automate the generation of architecture diagrams, ensuring consistency and accuracy throughout the software development lifecycle [14]. Mostafa Hamza, Sherif Aly, and Hoda Hosny (2012) present a feature-driven approach to generate pervasive system architectures. The methodology involves automatically constructing architectures based on a predetermined set of architectural features, utilizing tools like the Feature Modeling Plug-in (FMP) with Eclipse and Visual Paradigm for UML. The system aggregates various components relevant to the selected features to create a configurable reference architecture for pervasive systems, supporting rapid development [15]. Lastly, existing software update tracking tools are specific to certain ecosystems. For example, GitHub updates can be tracked using tools such as Renovate [18]. However, such tools are limited to specific software repository ecosystems.

In contrast, our work continuously updates and automates the documentation of software architectures using enriched software version data and UML visualization, focusing on improving the software engineering process. By incorporating an operating system and its components approach, we effectively identify and illustrate common architectural dependencies. This method aims to improve the visibility of critical interactions within the ecosystem.

3. Methodology

In this paper, we apply a three-step methodology to automate the generation of UML diagrams for software ecosystem architecture visualization via software release notes. Initially, the release notes API enables users to select software components of interest. This selection process can be automated for reverse-engineering existing ecosystems or conducted manually for new designs. Next, these components are transmitted to the diagram generator API, which utilizes the standard PlantUML language. This diagram generation is continuously updated to reflect real-time software updates, aiming that the UML diagrams are current [17]. Finally, the resultant UML diagram depicting the ecosystem architecture is displayed through the visualization tool (see Fig. 1).

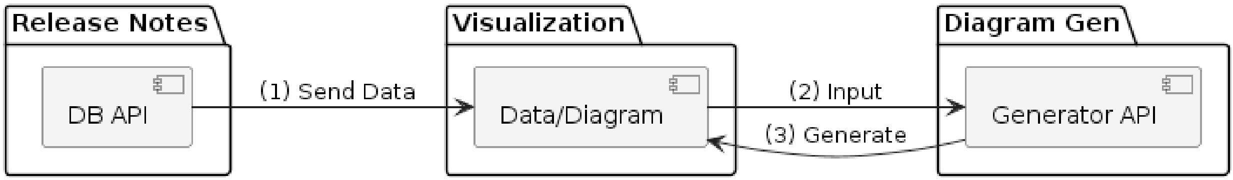

Fig. 1. Architecture of the Software Ecosystem Visualization System

3.1. Design

Unified Modeling Language (UML) is our suitable choice for our visualization needs, primarily because it is an industry standard. Its widespread acceptance ensures consistency and compatibility across different platforms and tools. Furthermore, UML supports APIs that allow for the dynamic generation of diagrams from textual descriptions, supporting our ability to update and modify visualizations in real-time as system requirements evolve. Additionally, UML is extensible, enabling us to customize its capabilities to meet specific visualization requirements throughout various software engineering phases of our use case application.

3.2. Data and Algorithm

In this work, we utilize software release notes data to visualize dependencies within software ecosystems. Specifically, our aim is to describe the relationships between operating system components and their installed software components. Since 2020, our database has queried data on over 80,000 software updates across more than 16,000 software components from sources such as GitHub, Wikipedia, DistroWatch, and MITRE. This extensive dataset significantly improves our ability to document and analyze the intricate interactions and dependencies within these systems. Leveraging detailed attributes from release notes, our approach provides clear and structured visualizations by incorporating details such as release dates, component names, version numbers, component types, and distribution channels. This structured approach not only supports the precise documentation of software ecosystems but also improves the understanding of their architectural evolution and maintenance challenges. In this work, the algorithm implements an hierarchical, stacked, and peer representation of operating system components and associated software components within a UML framework. Our visualization dynamically maps operating system components to an outer UML package, which encapsulates both the operating system itself and all related non-operating system software components (see Fig. 3).

Mapping of Version Data to UML Diagram

Software Component `<<Component>>`
Models distinct software units (e.g., servers, databases) as UML `<<Component>>`, enabling modular representation.

Version, Date, Channel `<<Attribute>>` Encodes version details, release dates, and channels as `<<Attribute>>`, with emphasis on major and patch versions in bold.

OS, Stacked Components `<<Package>>`
Groups related components into cohesive `<<Package>>` structures to represent stack-based or layered architectures.

Hierarchical, Peer Relationships `<<Component>>`
Captures hierarchical dependencies and peer relationships through nested and interconnected `<<Component>>` elements.

This structured approach not only supports the precise documentation of software ecosystems but also improves the understanding of their evolutionary dynamics, facilitated by the visual emphasis on critical updates.

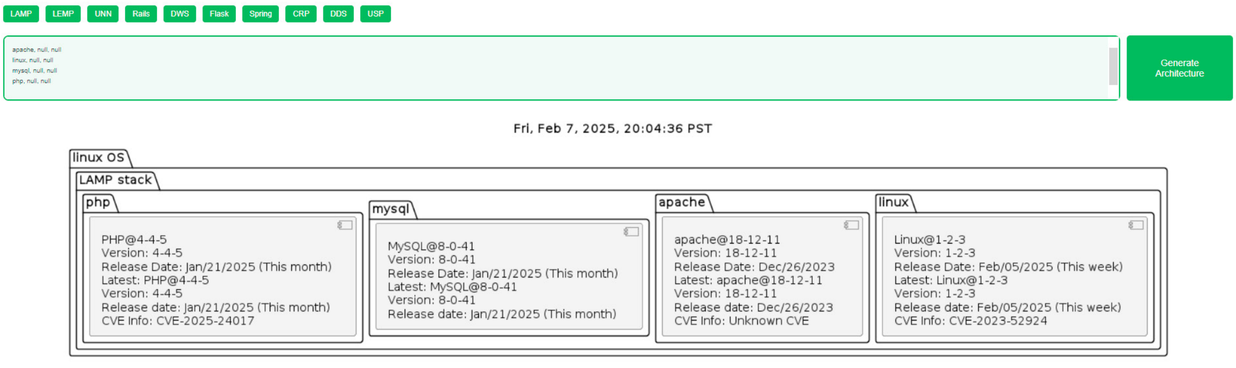

Fig. 2. Prototype: Linux, Apache, MySQL, PHP Ecosystem Architecture Design

4. Implementation

This section details the web-based implementation of our application prototype, focusing on utilizing the LAMP stack (Linux, Apache, MySQL, PHP) and its integration into the software engineering lifecycle.

4.1. Requirements

During the requirements phase, teams commonly evaluate the compatibility of the LAMP stack with our domain's requirements. This assessment includes a critical analysis of the current architecture's support for the LAMP stack components. A key activity includes users searching for the components individually—Linux, Apache, MySQL, and PHP—to reverse engineer and visualize the current technology stack installed within the ecosystem. However, the initial diagram generated indicates that while Linux and Apache are installed and visualized, MySQL and PHP components are missing.

4.2. Design Phase

During the design phase, it is required to evaluate which MySQL and PHP version to install to be compatible with the current Linux and Apache software. After the review and installation, the visualization tool is updated including the entire LAMP stack.

4.3. Implementation

During the implementation phase, the ecosystems team is usually tasked with continuously monitoring security patches as a crucial part of maintaining system integrity. The visualization tool plays a central role in this process by dynamically displaying security patches as soon as they are published. This capability is particularly vital for critical components of the LAMP stack, such as the database or the operating system. For example, an Apache security update notes an outstanding issue. Figure 4 illustrates an update scenario where an Apache-related security patch becomes available, emphasizing the tool's utility in providing real-time, actionable information to the team (see Fig. 2).

4.4. Testing Phase and Maintenance Phase

During the testing phase, the use of a well-defined software baseline is critical for the clear communication and execution of test cases. This baseline, which includes specific versions of the software components installed (not including security updates) within the LAMP stack, serves as a reference that ensures all testing activities are consistent and replicable while enabling efficient detection of deviations or potential regressions in updated components. After software release, the maintenance phase ensures system operation and security. The ecosystems team monitors updates, focusing on critical security patches impacting stability. Proactive monitoring of the automatically updated

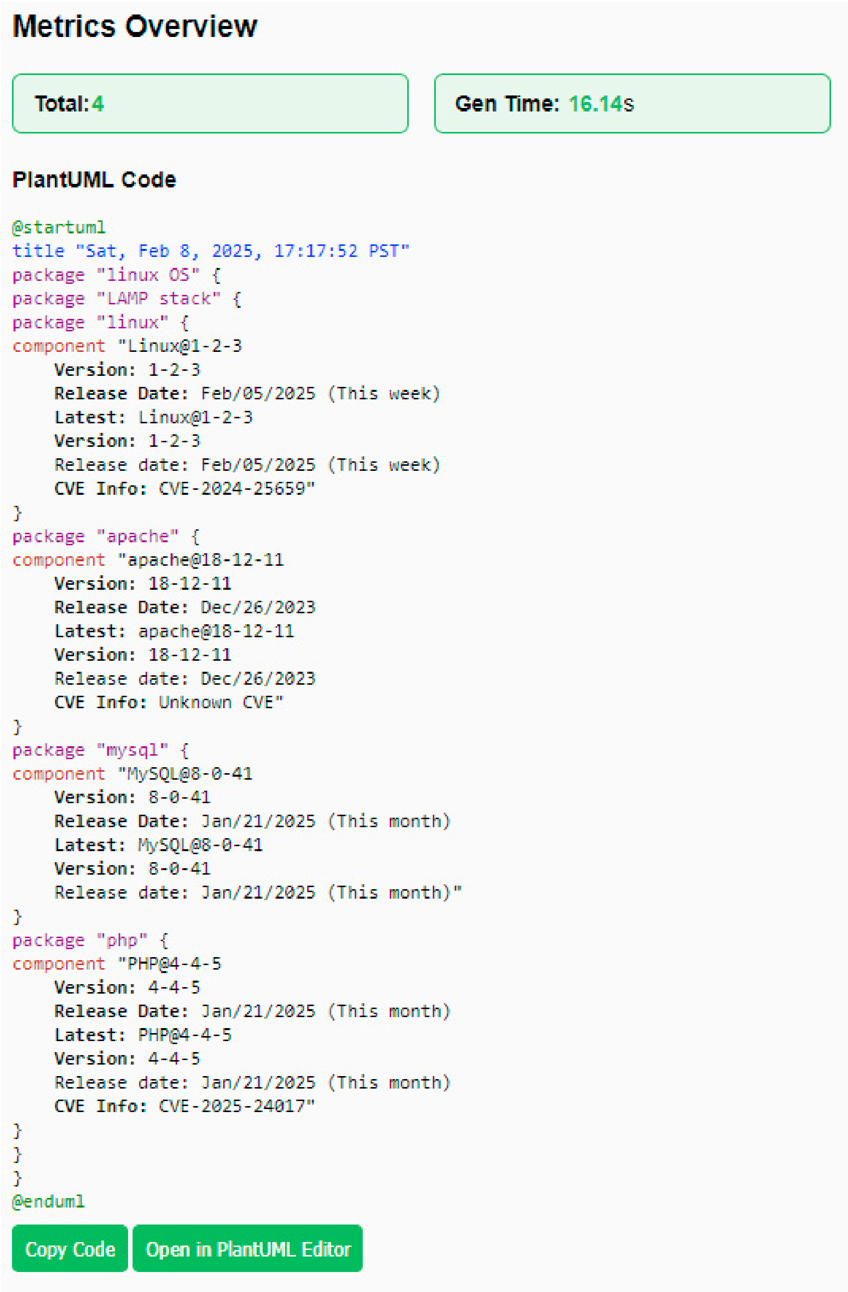

Fig. 3. Prototype: Metrics and Generated PlantUML Code

architectural diagram minimizes downtime and adapts to technological changes, allowing fast identification and resolution of potential vulnerabilities by leveraging visualized dependencies to prioritize critical updates.

5. Results and Discussion

Our experience with deploying the Unified Modeling Language (UML) for the automated visualization of software ecosystems has yielded several key insights [1]. Firstly, UML's status as a standardized visualization tool is improved by a robust suite of customization tools, which improved our ability to adapt the diagrams to various specifications. Technically, our system has demonstrated its capability to generate UML architectural diagrams automatically, which underscores the efficacy of the underlying algorithms and software design. The prototype has shown consistent performance across different web browsers including Safari, Chrome, and Firefox, highlighting its robustness and cross-platform compatibility. Furthermore, the ability to sort operating system components and associated software components by impact attributes, such as 'major' and 'recent', has proven effective. This sorting mechanism aims to improve the visual representation of components based on their relevance and impact. Its aim is to improve the utility and interpretability of the generated diagrams. Next, the section will discuss the three categories, scalability, usability, and limitations, in more detail.

5.1. Scalability

Scalability remains a significant challenge in our automated UML visualization system, particularly in terms of rendering times and display capacity. Currently, the system requires approximately 2 seconds per image for rendering operating system diagrams, which means that visualizing large datasets is not instantaneous and can lead to delays in the user experience. This limitation becomes more pronounced as the number of operating system components increases, as the system is challenged to efficiently display multiple components on a single screen or paper page. The limited scalability with regard to the number of OS components that can be effectively visualized simultaneously restricts the tool's utility in scenarios where comprehensive, real-time visualization of extensive software ecosystems is required.

5.2. Usability

The usability aspects of our automated UML visualization system for software ecosystems revealed both strengths and areas for improvement. To improve readability and utility, we created a single image for each operating system component; this approach was necessary to maintain clarity and prevent overcrowding in one row of the architectures. However, the system currently lacks support for making line breaks within UML diagrams, which can limit the ability to clearly separate and delineate complex information. Moreover, the diagrams are read-only, and the system does not support click events on single attributes, restricting interactive exploration and detailed examination of specific components directly through the diagrams. Regarding print output, the system's performance on A4 paper was moderate; although the resolution was generally satisfactory, diagrams occasionally did not float correctly, affecting the layout in printed documentation. On a positive note, our ability to apply different font decorations based on impact attributes—such as marking critical vulnerabilities (CVEs) in bold— aiming to improve the visual effectiveness of the diagrams, making it easier to identify and focus on high-impact components quickly.

5.3. Limitations

Our method for visualizing software ecosystems, though effective, encounters several technical limitations that can impact the accuracy and scalability of the diagrams generated. Firstly, our system is restricted to depicting read-only images of operating system component, which reduces the ability for interaction. Additionally, not all standard UML properties are supported, including features like line breaks within component descriptions, which limits the expressiveness of the visualization. The scaling of components also presents challenges, as the representation does not scale well between singular and multiple components, often leading to cluttered or unreadable diagrams if number of components varies too much. Moreover, during the tool implementation phase, the need to clear the browser cache after each update to prevent data inconsistency and the potential for incorrect architecture representations due to

[1] https://releasetrain.io/arch

erroneous data inputs further complicate the reliability. The use of an online service for API application also imposes dependency on external service availability and response times. The PlantUML tool, which we rely on for diagram generation, requires very exact spacing, quotation, and line breaks, which can be error-prone. Additionally, the API error messages provide limited information, complicating debugging efforts and slowing down both implementation and testing phases. Furthermore, not all characters are supported by the tool, for example, the dot (.) character can cause issues in the rendering process.

6. Conclusion

Maintaining software architectures is an arduous task, particularly given the increasing complexity and interdependencies of modern software ecosystems. In summary, our automated UML visualization system for software ecosystems, driven by software release notes as its primary data source, has demonstrated potential in improving the understanding of complex software architectures throughout all main software engineering phases. Addressing the root problem of managing these complexities, the system leverages the standardized features of UML to dynamically represent intricate relationships between operating system components and associated software components. By automating the visualization process and integrating detailed release note data, we provide a clearer, more structured view of these relationships. While the system effectively applies visual distinctions such as bold-text coding to emphasize critical updates, its current limitations include a lack of interactive functionalities (mainly read-only image) and challenges in handling large-scale visualizations without reducing clarity. Future work will focus on improving interactive capabilities and optimizing the scalability of the system to better accommodate larger software ecosystems. Overall, despite these challenges, the prototype system offers a foundation for further development and refinement in the field of software architecture visualization.

References

[1] L. Cavique, M. Cavique, A. Mendes, and M. Cavique, "Improving Information System Design: Using UML and Axiomatic Design," *Computers in Industry*, vol. 135, p. 103569, 2022. doi: 10.1016/j.compind.2021.103569.

[2] D. Sobhy, R. Bahsoon, L. Minku, and R. Kazman, "Evaluation of Software Architectures under Uncertainty: A Systematic Literature Review," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 4, article 51, pp. 1–50, Aug. 2021. doi: 10.1145/3464305.

[3] F. Tian, T. Wang, P. Liang, C. Wang, A. A. Khan, and M. A. Babar, "The Impact of Traceability on Software Maintenance and Evolution: A Mapping Study," *arXiv preprint*, arXiv:2108.02133, 2021. Available: https://arxiv.org/abs/2108.02133.

[4] O. Räihä, K. Koskimies, and E. Mäkinen, "Generating software architecture spectrum with multi-objective genetic algorithms," in *Proc. 2011 Third World Congress on Nature and Biologically Inspired Computing (NaBIC)*, 2011, pp. 29–36. doi: 10.1109/NaBIC.2011.6089413.

[5] S. Berhe, M. Maynard, and F. Khomh, "Software Release Patterns: When is it a good time to update a software component?" *Procedia Computer Science*, vol. 170, pp. 618–625, 2020. Available: https://doi.org/10.1016/j.procs.2020.03.142.

[6] S. Berhe, M. Maynard, and F. Khomh, "Maintenance Cost of Software Ecosystem Updates," *Procedia Computer Science*, vol. 220, pp. 608–615, 2023. Available: https://doi.org/10.1016/j.procs.2023.03.077.

[7] S. Berhe, V. Kan, O. Khan, N. Pader, A. Z. Farooqui, M. Maynard, and F. Khomh, "Attribute-Driven Maintenance Cost Estimation in Graph-Based Software Ecosystem Models," *Accepted to Journal of Ubiquitous Systems & Pervasive Networks*, vol. TBD, no. TBD, pp. TBD, 2024.

[8] M. Erder and P. Pureur, *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World*, Morgan Kaufmann, 2015.

[9] N. A. Ernst and M. P. Robillard, "A study of documentation for software architecture," *Empirical Software Engineering*, vol. 28, no. 5, p. 122, 2023.

[10] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley Professional, 2018.

[11] S. Luján-Mora, P. Vassiliadis, and J. Trujillo, "Data Mapping Diagrams for Data Warehouse Design with UML," in *Proc. 23rd Int. Conf. Conceptual Modeling (ER 2004)*, Shanghai, China, Nov. 2004, pp. 191–204. Available: https://doi.org/10.1007/978-3-540-30464-7_16.

[12] N. Bali and A. Sawant, "SoftArchViz: An Automated Approach to Visualizing Software Architectures," in *Proc. Int. Conf. Software Engineering*, 2007, pp. 60–66.

[13] C. Narawita and K. Vidanage, "UML Generator - An Automated System for Model Driven Development," in *Proc. 2016 16th Int. Conf. Advances in ICT for Emerging Regions (ICTer)*, Colombo, Sri Lanka, 2016, pp. 250–256. doi: 10.1109/ICTER.2016.7829928.

[14] Y. Gamage, "Automated Software Architecture Diagram Generator using Natural Language Processing," 2023. Available: https://doi.org/10.13140/RG.2.2.31866.26563.

[15] M. Hamza, S. Aly, and H. Hosny, "An Approach for Generating Architectures for Pervasive Systems from Selected Features," 2012.

[16] Y. Ghanam and S. Carpendale, "A Survey Paper on Software Architecture Visualization," 2008.

[17] J. Lärfors, "Making Software Architecture a Continuous Practice," 2020.

[18] Renovate, "Automated Dependency Updates for GitHub and Other Repositories," *Renovate Documentation*. Available: https://docs.renovatebot.com/, [Accessed: Jul. 5, 2024].