https://doi.org/10.31891/2219-9365-2026-86-20
UDC 004.4, 004.8

BOIKO Viacheslav
Khmelnitskyi National University
https://orcid.org/0009-0004-9443-0286
e-mail: bviacheslav.12@gmail.com
MARTYNIUK Valerii
Khmelnitskyi National University
https://orcid.org/0000-0001-5758-4244
e-mail: mac30973097@gmail.com

METHOD OF AUTOMATED CONTINUOUS GENERATION OF UML DIAGRAMS IN LARGE-SCALE SOFTWARE REPOSITORIES

Large-scale software projects often struggle to keep design documentation, such as UML diagrams, up-to-date with rapid code changes. This article proposes a method to integrate automated UML diagram generation into the continuous integration/continuous deployment (CI/CD) pipeline of a large .NET code repository. Upon each commit to the main branch, the CI pipeline triggers a secondary process that uses an AI component to detect code structure changes and update or create corresponding UML diagrams. Existing diagram definitions are retrieved and provided to the AI alongside code diffs, enabling it to generate revised UML. The updated diagrams are then published back to the documentation page, replacing outdated content. This ensures that architectural diagrams continuously evolve with the codebase, eliminating documentation drift and reducing manual effort. The main pipeline comprises: building the primary solution, executing the UML generation module, updating documentation, and then proceeding with further build steps. The approach leverages text-based "diagrams-as-code" and AI to maintain living documentation in real time. The paper analyzes recent research in continuous documentation and AI-driven code visualization, outlines the implementation in detail, and discusses the benefits and challenges.

Keywords: automated UML models generation, AI-driven UML modeling, software quality assurance, object-oriented programming, CI/CD pipeline, large-scale repositories.

БОЙКО В'ячеслав, МАРТИНЮК Валерій
Хмельницький національний університет

МЕТОД АВТОМАТИЗОВАНОГО ГЕНЕРУВАННЯ UML ДІАГРАМ У ВЕЛИКОМАСШТАБНИХ РЕПОЗИТОРІЯХ КОДУ

Великомасштабні програмні проєкти часто стикаються з проблемою актуальності проєктної документації, зокрема UML-діаграм, у контексті швидких змін у вихідному коді. У цій статті запропоновано метод інтеграції автоматизованого генерування UML-діаграм у конвеєр безперервної інтеграції та розгортання (CI/CD) великого .NET-репозиторію. Після кожного коміту до головної гілки CI-конвеєр запускає вторинний процес, який використовує компонент штучного інтелекту для виявлення змін у структурі коду та створення або оновлення відповідних UML-діаграм. Існуючі визначення діаграм отримуються та передаються разом із змінами в коді, що дає змогу AI-модулю генерувати оновлені UML-моделі. Оновлені діаграми публікуються на сторінці документації, замінюючи застарілий зміст.

Це забезпечує безперервну еволюцію архітектурних діаграм разом із кодом, усуваючи розрив між документацією та реалізацією й зменшуючи ручні витрати. Основний конвеєр включає: збирання основного рішення, виконання модуля генерування UML, оновлення документації та подальші етапи збірки. Підхід використовує текстовий формат «діаграми-як-код» у поєднанні зі штучним інтелектом для підтримки живої документації в режимі реального часу. У статті проаналізовано сучасні дослідження з безперервної документації та AI-керованої візуалізації коду, докладно описано реалізацію та розглянуто переваги й виклики.

Ключові слова: автоматизоване генерування UML моделей, AI-кероване UML-моделювання, забезпечення якості програмного забезпечення, об'єктно-орієнтоване програмування, конвеєр CI/CD, великомасштабні репозиторії.

Стаття надійшла до редакції / Received 19.02.2026
Прийнята до друку / Accepted 15.04.2026
Опубліковано / Published 31.05.2026

ANALYSIS OF THE LATEST RESEARCH

Ensuring software documentation remains current in fast-paced development is a long-standing challenge. Industry reports and research indicate that in many organizations, the creation of supporting documents cannot keep up with the rapid rhythm of code delivery under DevOps. This problem in updating documentation becomes a bottleneck that limits the value delivered to customers, even as deployment cycles accelerate. Recognizing this, recent years have seen a push towards continuous documentation practices, sometimes termed "DocOps" or DevDocOps, which integrate documentation tasks into the development pipeline to produce up-to-date docs alongside the code. For example, Rong et al. introduced DevDocOps, an approach for automated continuous documentation in DevOps. Their implementation uses templates and toolchains within the delivery pipeline to generate technical documents in tandem with code changes [1]. In a large enterprise case study, this reduced the lag between product release and documentation release from 1 or 2 months to under 2 days. This finding underscores that bridging the gap between rapid development

and slower documentation processes is possible with automation, thereby extending DevOps principles to documentation continuity.

Other papers have similarly highlighted the importance of keeping documentation in sync with code for knowledge preservation. Theunissen et al. discuss the risk of "knowledge vaporization" when project decisions and design rationale are not documented during agile development. They propose approaches like "Just Enough Upfront" documentation and treating executable artifacts as "Executable Knowledge" to capture design information continuously. Automated text analysis is another approach in their study, reflecting a growing interest in applying AI/NLP to aid documentation in agile processes [2]. Poniszewska-Marańda et al. also note that agile teams often minimize documentation. However, they stress that some documentation (architecture descriptions, etc.) is still essential and propose lightweight techniques to integrate documentation in agile workflows [3].

A key enabling technology for continuous documentation is the concept of treating documentation as code. This entails using text-based formats under version control and generating human-readable docs from these sources. In the context of design diagrams, this is exemplified by tools like PlantUML and Mermaid, which allow UML diagrams to be defined in a concise text syntax. Such tools have gained popularity since 2020 for automating diagram creation directly from code or markdown documents. For instance, PlantUML is an open-source tool that can generate class, sequence, and other UML diagrams from a plain text description embedded in source files. Because the "diagram source" is plain text, it can be stored in Git alongside code, diffed, reviewed, and updated as part of normal development. Notably, PlantUML diagrams can be automatically regenerated as part of CI/CD pipelines, ensuring they always reflect the latest state of the codebase.

Industry guides highlight that integrating such text-based UML generation in CI/CD helps eliminate documentation drift: every code change that affects design can trigger an update to diagrams, producing "living documentation" [4]. As one technical blog put it, "Automated diagram generation within CI/CD pipelines keeps documentation updated with every code change" [5]. This approach, combined with embedding diagrams in project wikis or platforms like Confluence via automated scripts, means teams no longer have to manually redraw architecture diagrams after each refactoring. Instead, the latest system structure is always available in visual form, improving communication and reducing misunderstandings.

Beyond traditional rule-based tools, AI and large language models (LLMs) have recently been explored for automating software documentation tasks, including UML diagram generation. The emergence of advanced LLMs (such as GPT-5) opened new possibilities to interpret code or natural language descriptions and produce design artifacts. Research in 2024 – 2025 has started to assess the feasibility of using LLMs to generate UML diagrams automatically. Krishnan et al. conducted a comparative analysis of several LLMs for generating UML use case diagrams from textual software requirements [5]. Similarly, other researchers have demonstrated that domain-specific training or prompt engineering enables LLMs to produce correct UML notations. For instance, an experiment by Alessio et al. demonstrated that GPT-based models can translate natural language descriptions into code snippets and structural representations, including UML elements [6]. These studies collectively indicate that AI is a promising tool for automating design documentation, as it can understand code or requirement changes and directly produce updated UML diagrams.

In practice, we also see early adoption of such AI capabilities in tools. For example, Visual Paradigm's 2023 update introduced an AI-assisted UML generator that guides users through diagram creation and even auto-generates draft class diagrams and suggestions [7]. Miro and other collaboration platforms have added AI plugins to create diagrams from text prompts as of 2023 – 2024. Additionally, case studies by tech enthusiasts illustrate using ChatGPT to quickly create PlantUML diagrams, saving developers the effort of learning diagram syntax and manually drawing. For instance, Kostrikova demonstrated a workflow where a system design description is fed to ChatGPT, which returns a PlantUML sequence diagram that can be rendered to visualize the design [8]. This confirms that current LLMs can perform UML modeling – translating design intent into diagram code. However, academic evaluations caution that without careful validation, AI-generated diagrams might contain mistakes or omissions, so a human review or automated checks are recommended [5][6].

In summary, recent research and practice converge on a few key points: documentation generation should be continuous and automated to keep pace with agile DevOps workflows; text-based diagrams-as-code and CI/CD integration are effective means to achieve up-to-date UML diagrams in large projects [9][10]; AI-powered tools and LLMs are emerging as powerful aids to automate the understanding of code changes and generation of UML artifacts, though best results often come from domain-specific tuning. These insights set the stage for the method proposed in this article, which combines these elements to maintain current UML models in a large-scale .NET repository automatically.

FORMULATION OF THE GOALS OF THE ARTICLE

The goal of this article is to present a novel method for automated, continuous UML diagram generation as part of the CI/CD process in a large software repository. The method aims to ensure that model diagrams are updated on each relevant code change without human intervention. We seek to design the integration of an AI-driven UML generation step into the build pipeline of a .NET project, triggered by code commits, detect and reflect code structure

changes in the UML documentation, and automatically publish these updated UML diagrams to a Confluence documentation page. In doing so, the article aims to demonstrate how this approach keeps documentation consistent with the codebase in real-time and to evaluate its benefits and potential challenges. The ultimate goal is improving project maintainability and knowledge sharing by eliminating stale design docs.

PRESENTING THE MAIN MATERIAL

The context is a large-scale .NET software project with an existing CI pipeline (e.g., Azure DevOps or GitHub Actions) that builds and deploys the application. We insert into this pipeline a new job responsible for generating UML diagrams whenever the code changes on the main branch. The process consists of 4 steps. These steps are described below.

A developer's commit merged to the main (master) branch, triggering the continuous integration pipeline. This typically includes steps like compiling the code, running tests, and perhaps packaging artifacts. In our extended pipeline, after a successful build of the main .NET solution, control passes to the UML generation stage.

A separate project or module referred to as the UML Generator is built and executed. This component contains the logic to produce updated UML diagrams. It can be a small .NET console application that runs as part of the CI job. The UML Generator is configured with access to both the latest code and the documentation repository (Confluence, for example). Upon running, it performs two primary functions: change detection and model PlantUML code extraction from the existing document in Confluence via API calls, and calling an AI service to generate the diagram.

The UML Generator identifies what has changed in the codebase that might affect architectural diagrams. This could be done by parsing the git diff or comparing the latest build artifacts (assemblies) with a previous baseline. For example, if a new class was added or an interface changed, these are noted. In a large project, generating a full class diagram of the entire system would be overwhelming, so the tool can scope the diagrams to specific modules or components that have changed. Alternatively, the approach can regenerate a set of high-level diagrams on each run – ensuring they are always current. The choice between incremental updates vs. full regeneration can depend on performance considerations. For our method, we assume the UML Generator knows what diagrams to produce and what sections of the code each covers.

Once relevant code changes are identified, the UML Generator prepares input for the AI model. This includes extracting the current design information. If an existing UML diagram is being updated, the tool will retrieve the latest diagram definition as a PlantUML text from Confluence using Confluence's REST API. This serves as context so that the AI can modify it rather than generating it from scratch. Alongside this, the tool gathers code context. This could be a simplified representation of the code – for instance, a list of classes and their relationships in the changed module, or even the raw code of the classes. A pragmatic approach is to use a lightweight static analysis: the tool could utilize Roslyn, the .NET compiler API, to reflect on the structure and then summarize these changes in text form. For example, the prompt to the AI might say: "Update the UML class diagram to include the following changes: Class X (inherits Y, implements interface Z, calls class W) … The current UML diagram definition is: @startuml ... @enduml. Modify it accordingly." This prompt engineering ensures the AI has both the context (existing diagram) and the delta (code changes).

We leverage an OpenAI model (such as GPT-5) via its API to generate the updated UML diagram code. The model, with the prompt described, should output PlantUML syntax representing the new or changed diagram. For instance, if a new class is added, it will add a box for that class and draw associations or inheritance arrows as described. If a class was modified or removed, the model will update or remove those elements in the PlantUML. Because the model has limitations, the UML Generator post-processes the output. It can verify syntax by running it through a PlantUML parser/renderer in headless mode. If any syntax errors are found, the tool can either attempt minor fixes or log an error. (In a future enhancement, one could even have the AI self-correct by showing it the error, but our current method assumes a relatively straightforward generation that typically passes syntax).

After obtaining the updated UML diagram code, the UML Generator uses the Confluence REST API to update the documentation page. The target is a Confluence page that either contains the diagram in an embedded form or contains an attachment for the diagram. There are a couple of strategies: one is to use a Confluence PlantUML plugin or macro, where the page content includes the PlantUML source between special tags, and Confluence renders it. In this case, our tool would perform a page update operation, replacing the old PlantUML block with the new one. Another strategy is to generate an image from the PlantUML and upload that image. However, maintaining the text source in Confluence is preferable, so that future updates have a base to diff against, and humans can edit if needed. Using the Confluence Cloud API, the UML Generator authenticates with a service account or API token, fetches the current page content in storage format or markdown, and locates the diagram section. It then constructs the new page content with the updated diagram. The Confluence API requires increasing the page version when updating; our tool handles this by retrieving the current version number and setting version = old+1 in the update payload. The update is posted via an HTTP PUT request to the Confluence REST endpoint for page content [9]. On success, Confluence stores the new version of the page with the updated diagram. This automation essentially replaces what used to be a manual edit in Confluence.

Once the documentation is updated, the CI/CD process can continue to subsequent steps such as running additional tests, packaging, and deployment. If the UML generation stage fails, it will not block the pipeline. Moreover, the further steps and artifacts deployment happen in parallel to the documentation generation process. The whole process is shown in Fig. 1.

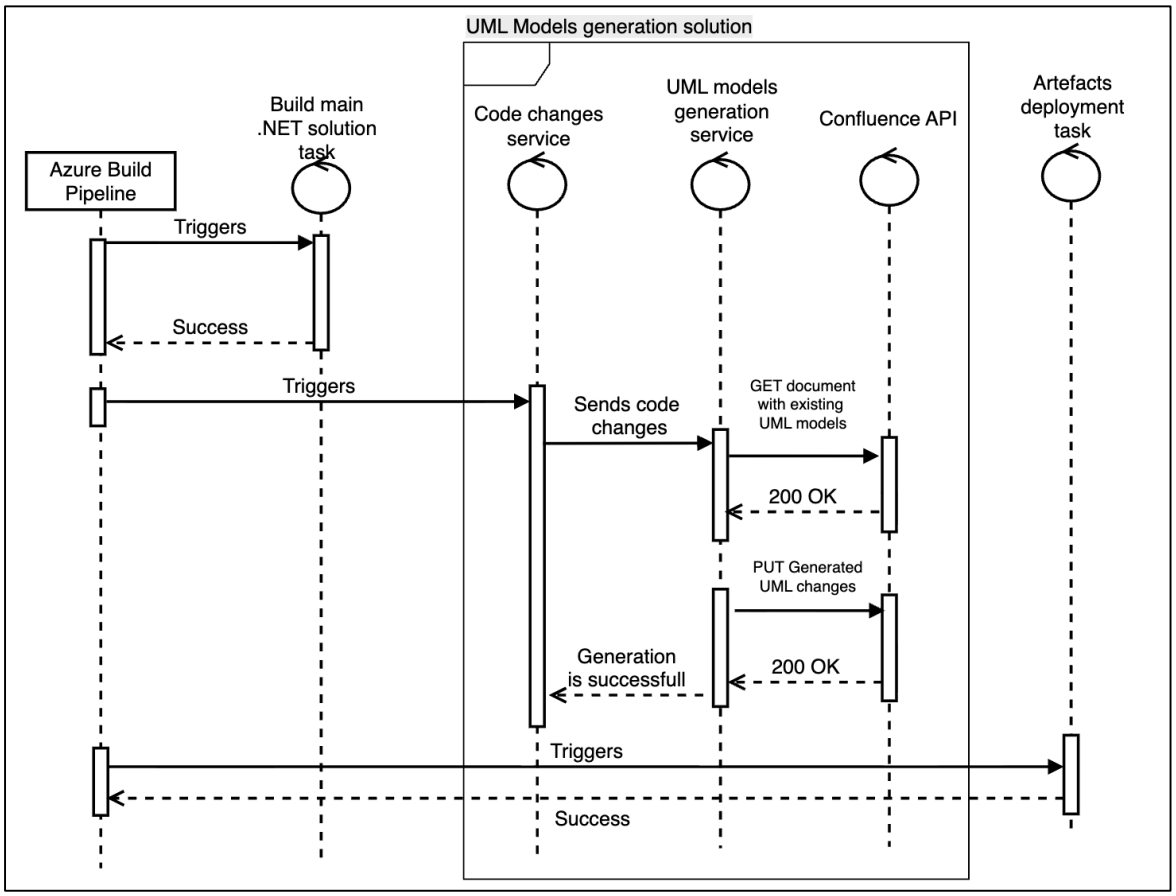

Fig. 1. Automated UML diagrams generation

While our implementation focuses on class and component diagrams for a .NET system, the concept can extend to other diagram types and tech stacks. For instance, sequence diagrams could be updated if there is a change in a workflow, though automated detection of sequence changes is harder (it may require instrumentation or conventions in code to map to sequence steps). Some research prototypes generate sequence diagrams from execution traces or user stories [6], which could be integrated with AI as well. Another use case is updating C4 model diagrams (Context, Container, Component, Code diagrams), which many modern development teams use for architecture documentation. The C4 model is essentially a set of hierarchical diagrams of the system; our pipeline approach could ensure C4 diagrams are kept in sync.

CONCLUSIONS FROM THIS STUDY AND PROSPECTS FOR FURTHER RESEARCH IN THIS DIRECTION

Keeping documentation in lockstep with code is no longer an unattainable ideal but a realistic goal given modern tools and techniques. This article presented a method to automate UML diagram updates as part of the CI/CD pipeline for large-scale projects. By integrating an AI-driven UML generation step into the build process, we ensure that every significant code change triggers a corresponding documentation change. The approach was discussed in the context of a .NET project using Azure DevOps and Confluence, but the principles are general: treat diagrams as code, use automation to update them continuously, and harness AI to interpret complex changes.

The proposed method effectively eliminates the problem of outdated architecture diagrams. Developers and stakeholders can trust that the diagrams always reflect the latest implementation. This has multiple positive impacts: new developers onboarding onto a large system can rely on up-to-date visuals to understand software structure, reducing ramp-up time. Teams can conduct design reviews or threat modeling on current diagrams, not ones that are months old. Knowledge retention is improved – design decisions are captured when code is written, avoiding the loss of rationale that happens when documentation lags. Additionally, automating the work saves developer effort. Rather than spending hours manually drawing UML in a tool after implementing a feature, the developer's job is simply to write code; the pipeline handles the rest. Over time, this can yield substantial productivity gains and ensure more

consistent documentation quality. This approach aligns with the concept of continuous documentation and has proven its value in related studies.

While the method is powerful, we encountered some challenges that suggest areas for future work. One challenge is ensuring the accuracy of AI-generated diagrams. LLMs are imperfect; there were cases in testing where the AI omitted a class relationship or misnamed an element. To mitigate this, one could incorporate a verification step, for instance, by comparing the AI's output against a simple static analysis output. A potential research direction is to combine symbolic analysis with AI: use a parser to get a ground truth list of classes and relations, and use the LLM to arrange or annotate the diagram, flagging any discrepancies. Another challenge is scaling to very frequent commits. If dozens of commits are merged per day, do we regenerate the diagram each time? This could lead to excessive load or meaningless churn if changes are minor. A smarter trigger mechanism could be implemented. This might involve analyzing commit messages or diff content to decide whether to run the UML update. Integrating such logic is a prospect for further refinement.

The prospects for further research include evaluating the approach in different environments and quantifying its benefits. For instance, user studies or industry trials could measure how much time is saved in design documentation and whether code comprehension improves for the team. Another area is exploring fine-tuning LLMs on code-to-diagram tasks: by training on a dataset of code and corresponding UML, one could create a specialized model that might output even more precise and complex diagrams. Also, investigating the use of retrieval-augmented generation could improve the consistency and correctness of AI outputs.

In conclusion, the method of automated continuous UML generation in CI/CD offers a practical solution to a common documentation problem in large software projects. It leverages the synergy of DevOps automation and cutting-edge AI to ensure documentation is no longer a stale afterthought but a living part of the development process. This enhances knowledge sharing, reduces errors (since design mismatches can be caught early when docs and code differ), and ultimately contributes to higher quality and maintainability of software. Future enhancements and research will further streamline this integration, possibly generalizing it into a standard practice in the software industry. The trajectory suggests that we are moving into an era where up-to-date documentation is automatically generated as an integral part of software delivery, fulfilling one of the long-standing goals of software engineering.

References

1. Rong G., Jin Z., Zhang H., Zhang Y., Ye W., Shao D. DevDocOps: Enabling continuous documentation in alignment with DevOps / Rong G., Jin Z., Zhang H., Zhang Y., Ye W., Shao D. // Software: Practice and Experience, 50(3). – New York, USA, 2020. – P. 210–226. DOI: 10.1002/spe.2770.

2. Theunissen T., Hoppenbrouwers S., Overbeek S. Approaches for Documentation in Continuous Software Development / Theunissen T., Hoppenbrouwers S., Overbeek S. // Complex Systems Informatics and Modeling Quarterly, (32). – Riga, Latvia, 2022. – P. 1–27. DOI: 10.7250/csimq.2022-32.01.

3. Poniszewska-Marańda A., Zieliński A., Marańda W. Towards project documentation in agile software development methods / Poniszewska-Marańda A., Zieliński A., Marańda W. // Lecture Notes on Data Engineering and Communications Technologies, 30. – Cham, Switzerland, 2020. – P. 1–18. DOI: 10.1007/978-3-030-19069-9_1

4. Kurotych A. O., Bulatetska L. V. Optimizing the process of ER diagram creation with PlantUML / Kurotych A. O., Bulatetska L. V. // CEUR Workshop Proceedings. – Lutsk, Ukraine, 2024. – P. 47–57.

5. Krishnan N.G.S., Ambadi S., Thushara M.G. Comparative Analysis of Large Language Models for Automated Use Case Diagram Generation / Krishnan N.G.S., Ambadi S., Thushara M.G. // In Proc. of 3rd International Conference on Futuristic Technology (INCOFT 2025), vol. 2. – India, 2025. – P. 465–471. DOI: 10.5220/0013594700004664.

6. Alessio F., Sallam A., Chetan A. Model Generation from Requirements with LLMs: an Exploratory Study / Alessio F., Sallam A., Chetan A. // Preprint. – 2024. – P. 1–20.

7. Ardimento P., Bernardi M.L., Cimitile M. Teaching UML using a RAG-based LLM / Ardimento P., Bernardi M.L., Cimitile M. // In Proc. of 2024 International Joint Conference on Neural Networks (IJCNN 2024), IEEE. – Yokohama, Japan, 2024. – P. 1–8. DOI: 10.1109/IJCNN60899.2024.10651492

8. Kostrikova Y. (2024). Automating UML Diagrams with ChatGPT and PlantUML. Personal blog, January 28, 2024. https://kostrikova.dev/blog/uml-chatgpt-plantuml

9. Doc-E.ai. (2025). AI and Continuous Integration (CI) for Seamless Documentation Updates. Doc-E Blog. https://www.doc-e.ai/post/ai-and-continuous-integration-ci-for-seamless-documentation-updates

10. Bates A., Vavricka R., Carleton S., Shao R., Pan C. Unified Modeling Language Code Generation from Diagram Images Using Multimodal Large Language Models / Bates A., Vavricka R., Carleton S., Shao R., Pan C. // Machine Learning with Applications, 20. – Amsterdam, Netherlands, 2025. – P. 100660. DOI: 10.1016/j.mlwa.2025.100660.