C4Gen: LLM-Driven Recovery of C4 Architecture Diagrams from Software Repositories

Sven van Dam[1], Guus Ramackers[1] and Joost Visser[1]

[1]Leiden University, The Netherlands

Abstract

Architectural documentation is valuable for system understanding and stakeholder communication, but in practice it is often outdated or absent. Large Language Models (LLMs) create opportunities to accelerate architecture recovery from repositories, but their stochastic and black-box nature introduces risks of hallucination and reduced user control. In this paper, we present C4Gen, a traceable, human-in-the-loop framework for recovering C4 container- and component-level architectural knowledge from software repositories. C4Gen combines LLM-driven analysis techniques with diagram rendering and intermediate human validation steps, such that practitioners remain in control throughout the recovery process. We evaluate the framework on four real-world systems and additionally evaluate its practical applicability in an IT consulting context. Our results indicate that C4Gen consistently generates diagrams that are syntactically correct and that generally align with expectations. At the same time, observed hallucinations and false negatives indicate that C4Gen should be positioned as a human-in-the-loop support tool rather than as a fully autonomous architecture recovery agent. In comparison with general-purpose AI coding assistants, C4Gen improves over a zero-shot baseline (GitHub Copilot) and slightly trails a fully autonomous agentic topline (Claude Code). Compared to the topline, C4Gen brings value by improving the traceability of how architectural information is recovered. Feedback sessions with IT consulting practitioners indicate that the framework has practical relevance as a support tool for accelerating architectural recovery.

Keywords

software architecture recovery, software engineering, LLMs, human-in-the-loop

1. Introduction

Software documentation, including architecture descriptions, can improve stakeholder communication and system understanding in the context of software systems [1]. Architectural knowledge is essential when practitioners (e.g., IT consultants) need to reason about a software system's structure and communicate about its decomposition and underlying relationships. However, in practice, architectural documentation is often outdated because of iterative development, and practitioners do not experience the creation and maintenance of documentation as enjoyable [2]. As a result, the recovery of architectural insights from software repositories can be a challenging activity in software engineering, especially when practitioners are unfamiliar with a software system and must obtain architectural insights in a short time frame.

The increasing availability of and improvements in Large Language Models (LLMs) have created opportunities to accelerate architecture recovery, because LLMs can analyse source code and identify higher-level patterns in software systems [3, 4]. However, LLMs are stochastic systems that employ information pathways that are different from those in humans [5], which may result in output that does not align with practitioners. Because LLMs inevitably hallucinate [6], LLM-generated architectural insights may be inaccurate or difficult to justify. In agentic pipelines, such inaccuracies may amplify when the output of one LLM invocation is used as input for subsequent LLM invocations [7]. In architecture recovery, these limitations are problematic because they can lead to architectural insights

BENEVOL2026: Belgium-Netherlands Software Evolution Workshop 26–27 November, 2026, Madrid, Spain
s.y.van.dam@umail.leidenuniv.nl (S. van Dam); g.j.ramackers@liacs.leidenuniv.nl (G. Ramackers); j.m.w.visser@liacs.leidenuniv.nl (J. Visser)
https://jstvssr.github.io/ (J. Visser)
0009-0003-4133-1279 (S. van Dam); 0000-0001-7033-7184 (G. Ramackers); 0000-0003-0158-3095 (J. Visser)

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

that are inaccurate or increasingly distorted as they propagate through agentic pipelines, making the reliability of those insights harder for practitioners to assess.

To address these limitations, we present C4Gen, which is a human-in-the-loop framework for recovering C4 model [8] container- and component-level architectural knowledge from software repositories. The full framework, including prompts, is openly available via GitHub [9]. As architecture descriptions are not solely determined by technical details, but are also shaped by stakeholder-specific concerns [10], AI-driven architecture recovery should enable practitioner intervention throughout the recovery process. This allows practitioners to correct inaccuracies and to incorporate stakeholder-specific perspectives into the resulting architecture artefacts. C4Gen combines LLM-driven analysis with human validation to enable practitioner involvement. We evaluate the syntactic correctness of C4Gen's output for four real-world systems and evaluate the semantic quality using semantic scoring metrics. Additionally, we evaluate the practical applicability of the framework by conducting feedback sessions with practitioners in the IT consulting domain.

2. Related Work

Two general types of approaches exist for automated software architecture documentation: traditional bottom-up approaches and more modern top-down approaches. In bottom-up approaches, low-level implementation artefacts are analysed first. Subsequently, low-level findings are aggregated and used to create higher-level insights. Bottom-up approaches [11, 12, 13] often rely on deterministic lexical parsing techniques that parse source code into intermediate representations, which can subsequently be used to create documentation. Such techniques are useful for generating low-level documentation, but they generally do not capture the higher-level architectural structure of a software system, and they are often constrained to specific programming languages.

More recent work has explored how to leverage LLMs to support documentation tasks. For example, DocAgent implements a multi-agent LLM-driven approach to generate documentation based on source code [14]. However, the tool only targets low-level docstrings and does not retrieve higher-level architectural structure. A 2023 bottom-up approach by Rukmono et al. leverages LLMs to generate summaries of lower-level atomic software elements, which subsequently are used to generate higher-level module and component summaries [3].

In contrast to bottom-up approaches, Rukmono et al. also introduced a top-down approach to software architecture recovery in which the recovery process starts with higher-level architectural structure rather than with low-level source code [4]. Such a top-down approach begins by hypothesising or identifying the higher-level architectural structure of a software system and providing an architecture outline, after which lower-level components are mapped to elements of that outline. Whereas bottom-up approaches iteratively build artefacts upwards, top-down approaches first establish architectural boundaries and subsequently map implementations to these boundaries.

Besides bottom-up and top-down approaches, general-purpose AI coding assistants such as GitHub Copilot and Claude Code can also be instructed to generate architectural diagrams. These assistants can analyse source code and respond to textual prompts, which may help practitioners obtain a quick initial impression of a system's architecture. However, they are not specifically designed for architecture recovery. As a result, they do not provide explicit decomposition strategies and intermediate validation steps. Therefore, traceability and user control are not guaranteed.

In our research, we use the C4 model to represent recovered architectural knowledge. Simon Brown introduced the C4 model as an *"easy to learn, developer-friendly approach to software architecture diagramming"* [8]. The C4 model is complementary to UML [15] and can be used to decompose software systems into a hierarchy of four abstraction layers. Of these layers, the container and component views are particularly relevant to visualise a software repository's architecture, because they describe its technical structure and can be derived from source code. Figures 1 and 2 show an example of a C4 container and component diagram. We have chosen to use the C4 model because of its simplicity and its ability to visualise systems from different viewpoints.

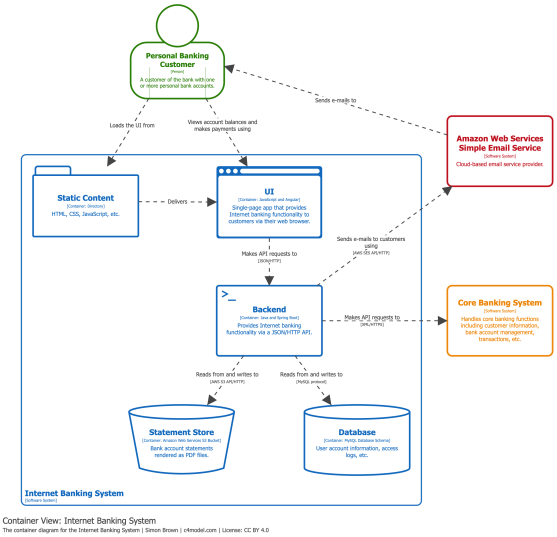


Figure 1: Example of a C4 container diagram. Image by Simon Brown. [8]

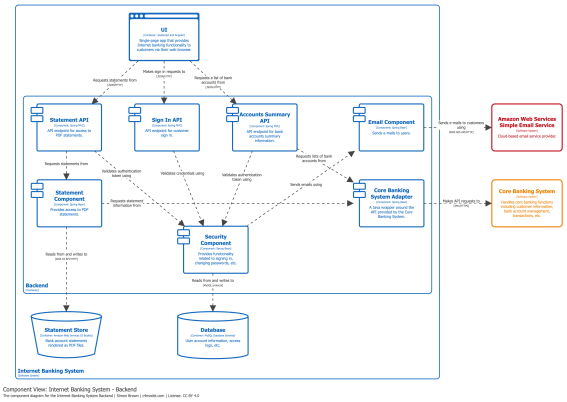


Figure 2: Example of a C4 component diagram. Image by Simon Brown. [8]

Existing approaches [11, 12, 13, 14] provide limited support for recovering architectural structure from repositories in a traceable, human-in-the-loop manner. Many existing tools either generate low-level documentation and rely on deterministic and language-specific lexical parsing, or they employ LLMs without integrating validation and correction possibilities.

There exists a variety of methods to evaluate LLM-generated output, including precision/recall/F1, BLEU/ROUGE [16, 17], and LLM-as-a-Judge. Precision, recall, and F1 metrics can be used to assess the performance of classification models, but they require a set of ground-truth labels to compare to. Such ground-truth labels are difficult or even impossible to collect in architecture modelling, because architectural viewpoints are stakeholder-specific, can have a different technical focus, and may vary in architectural depth. Thus, such metrics are not suitable for our research. BLEU and ROUGE likewise rely on lexical overlap with a reference text, and thus are also not appropriate for evaluating the semantic quality of LLM-generated architectural models. LLM-as-a-Judge could in principle be used, but this would introduce an additional stochastic evaluator whose judgements may not align with system stakeholders. Tseng et al. [18] introduced a novel methodology for evaluating the completeness of LLM-generated output using four different task-specific qualitative classification types: *match*, *valid*, *irrelevant*, and *hallucination*. This methodology is particularly relevant for use cases where multiple correct answers may exist, and where correctness depends on qualitative interpretation rather than on lexical overlap or on a fixed comparison with a ground truth.

3. The C4Gen Framework

Design Overview We developed a framework for recovering the C4 container- and component-level architectural decomposition from software repositories. The framework aims to support higher-level architectural recovery, rather than low-level docstring generation. In the framework, we combine LLM-driven analysis techniques with deterministic generation of architectural documentation. Figure 3 shows a high-level overview of our workflow, combining a top-down approach of identifying containers, container-to-container relationships, and components with a bottom-up approach of generating summaries and proposing component-to-component relationships. The top-down approach is inspired by the 2024 approach by Rukmono et al. [4], and the bottom-up approach is inspired by the 2023 approach by Rukmono et al. [3]. Table 4 in the appendix specifies the task, approach, inputs, and outputs of the different steps in the framework.

The workflow takes a single software repository as input, after which the top-down phase is triggered. This top-down phase encompasses two LLM-driven agents sequentially. The first agent tries to identify

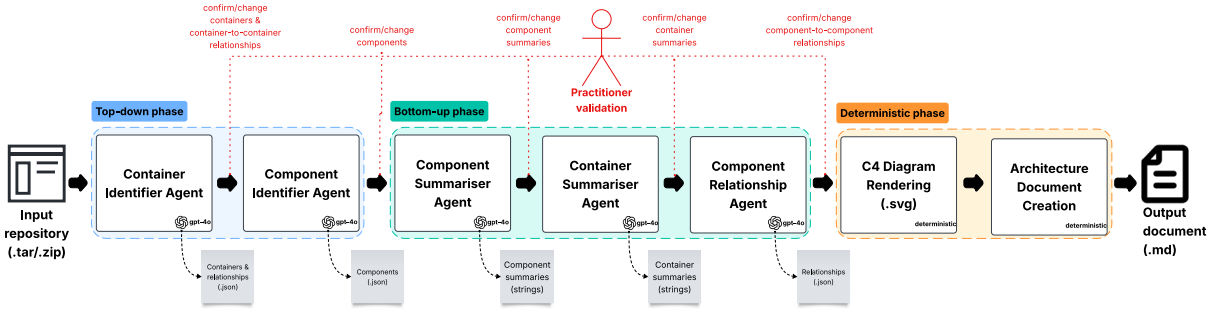


Figure 3: High-level overview of C4Gen's workflow. The framework combines top-down and bottom-up LLM-driven analysis techniques with deterministic generation of an output software architecture documentation artefact.

containers and container-to-container relationships based on the input repository's directory tree and the contents of high-level configuration files (e.g., Docker Compose file, Kubernetes manifests, service definition files). Such configuration files can hold evidence of what C4 containers (independently deployable/runnable units) the target system is composed of. The container identifier agent can use such evidence to recover the architecture of the system. For each identified container, the second agent tries to identify child components based on the container's properties and directory subtree. When this agent has finished, the workflow triggers the bottom-up phase.

The bottom-up phase encompasses three LLM-driven agents. For each component, the first agent generates a textual summary based on the component's source code. This summary describes the component's purpose, its key functionalities, and its relationships with other components. The second agent aggregates the child-component summaries for each parent container and generates a summary for each container (*summary-of-summaries* approach). Container summaries also describe the container's purpose, its key functionalities, and its relationships with other containers. For each container, the third agent aggregates the container summary and all child-component summaries and uses this information to try to identify intra-container component-to-component relationships.

The agents store retrieved information in intermediary container, component, and relationship objects. After the bottom-up phase, the workflow finalises with two more deterministic steps: the first creates PlantUML-C4 strings based on the intermediary objects, and the second aggregates diagrams and summaries in an output documentation artefact.

Traceable Human-in-the-Loop Recovery In this workflow, we integrate multiple mechanisms to improve traceability and to ensure that practitioners remain in control. Firstly, after all agent calls, we integrate a human validation step. We present the agent-identified elements (containers, components, relationships, summaries) and give the option to edit or delete them or to manually create new ones. In this way, practitioners can inspect and override generated output after each step before the workflow proceeds, preventing an amplification of inaccuracies. Additionally, the three identification agents generate a chain-of-thought motivation before coming up with a final judgement. Chain-of-thought prompting techniques aim to break down complex reasoning tasks into smaller sequential reasoning steps [19], which makes the overall reasoning process more observable and explainable. The generated chain-of-thought motivations are shown to the user, which increases observability. Finally, our workflow provides file-level traceability throughout the identification steps. The framework tracks the file paths of source code files that were used to come up with judgments, and presents these file paths to the user and additionally includes them in the presented motivations. In this way, observability and traceability are maintained, as humans can derive which data agents used to come up with judgments.

Prototype We implemented the framework in a Proof-of-Concept called C4Gen, which is composed of a React-based frontend, a Python/FastAPI backend, and a local PlantUML server for SVG rendering of C4 diagrams. The frontend provides the user interface for uploading repositories and validating

Table 1
Characteristics of the four different input systems that we use for evaluation.

Input system	Description	Programming languages/frameworks	Evaluated lines of code	Open-source?
C4Gen	Dogfooding case	Python, React	14,013	Yes
AI4MDE	System for AI-assisted model-driven engineering	Python, Django, Flask, React	33,724	Yes
Abacus	System for election counting procedures	Rust, React	143,616	Yes
Qubus	Platform for questionnaire development	C#, ASP.NET	38,805	No

agent-generated output. The backend implements prompt-building logic and response-parsing logic. For all agents, we select `gpt-4o` as an LLM because of its availability and demonstrated performance in architectural tasks [20, 21].

4. Evaluation

This paper reports the main elements of the evaluation design and its results. A complete description of the evaluation methodology is provided in the accompanying master's thesis [22]. We evaluate C4Gen on four different input systems: C4Gen itself (dogfooding), AI4MDE [23], Abacus [24], and Qubus [25]. These input systems were selected because of convenience. Table 1 provides the characteristics of the four different software systems. The systems are programmed using a mix of different programming languages and have different sizes (lines of code). The source code of one of the four systems is not disclosed publicly. We assess the syntactic correctness and semantic quality of the generated diagrams and compare them against two general-purpose AI coding assistants (GitHub Copilot and Claude Code). Additionally, we evaluate the practical applicability of the framework in an IT consulting context. Figures 4–6 show an example of a component diagram generated by GitHub Copilot, C4Gen, and Claude Code.

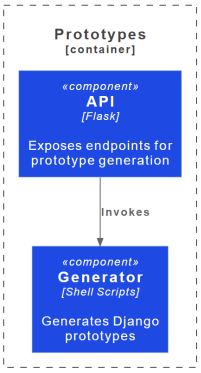


Figure 4: GitHub Copilot-generated component diagram for AI4MDE.

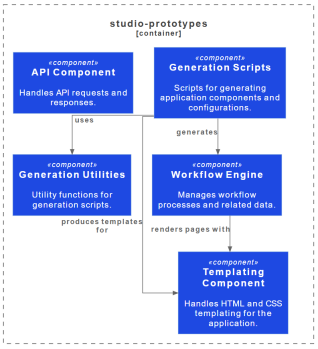


Figure 5: C4Gen-generated component diagram for AI4MDE.

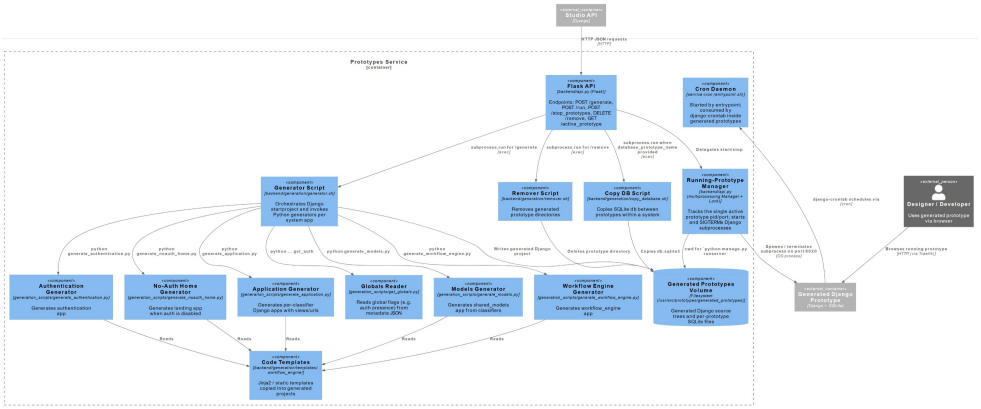


Figure 6: Claude Code-generated component diagram for AI4MDE.

Table 2
Descriptions for different classification types and corresponding scores. We reward for *match* and *valid*, and penalise for *irrelevant*, *hallucination* and *missing*.

Type	Description	Score
Match	"The element/relationship is present in the repository, and I would expect it in a container/component diagram."	+2
Valid	"The element/relationship is present in the repository, but I am indecisive about whether to expect it in a container/component diagram."	+1
Irrelevant	"The element/relationship is present in the repository, but I would definitely not expect it in a container/component diagram."	−1
Hallucination	"The element/relationship is not present in the repository, and I would definitely not expect it in a container/component diagram."	−2
Missing	"The element/relationship is missing in the container/component diagram, but it is present in the repository, and I would definitely expect it in a container/component diagram."	−2

Syntactic Correctness of Output We first assess whether C4Gen is capable of producing C4 diagrams that are syntactically correct. To do so, we evaluate whether the framework generates PlantUML-C4 strings that can be rendered successfully without manual corrections. C4Gen consistently produces renderable container and component diagrams for all four input systems, which indicates that the design choice of separating stochastic semantic reasoning from deterministic diagram rendering prevents syntax-level errors in the output.

Semantic Quality of Output To evaluate whether C4Gen creates semantically accurate architectural diagrams without any human intervention, we evaluate the semantic quality of generated container and component diagrams for four input systems using stakeholders' judgments. For each generated diagram, input system stakeholders classify containers, container-to-container relationships, components, and component-to-component relationships as *match*, *valid*, *irrelevant*, or *hallucination*. These classification types are inspired by the research conducted by Tseng et al. [18]. Additionally, we ask input system stakeholders to identify false negatives. Table 2 contains the descriptions that correspond to the different classification types, together with their corresponding scores. Based on the classifications and scores, we can calculate a raw semantic score for different element types using the following formula:

$$S_{raw} = 2n_m + n_v - n_i - 2n_h - 2n_{mi}$$

In this formula, n_m is the number of matching elements, n_v the number of valid elements, n_i the number of irrelevant elements, n_h the number of hallucinated elements, and n_{mi} the number of missing elements. Subsequently, we can normalise the raw score to the interval $[0, 1]$ using linear min-max normalisation:

$$S_{normalised} = \frac{S_{raw} - MinScore}{MaxScore - MinScore}$$

where:

$$MaxScore = 2 \times (n_m + n_v + n_i + n_{mi}) \qquad MinScore = -2 \times (n_m + n_v + n_i + n_h + n_{mi})$$

A normalised semantic score of 1 indicates a perfect match with stakeholder expectations and a score of 0 indicates a total mismatch. Additionally, we can calculate hallucination rates and missing rates, which indicate the proportion of generated elements that stakeholders classify as hallucinated, and the proportion of false negatives:

$$hallu\% = \frac{n_h}{n_m + n_v + n_i + n_h} \qquad missing\% = \frac{n_{mi}}{n_m + n_v + n_i + n_{mi}}$$

Table 3
Semantic quality of C4Gen's output compared with general-purpose AI coding assistants. Baseline refers to GitHub Copilot using `gpt-4o` and zero-shot prompting, C4Gen refers to the output produced by C4Gen, and Topline refers to Claude Code using Opus 4.7 with Thinking mode and RCI prompting. Reported values denote the arithmetic mean of normalised semantic scores, hallucination rates, and missing (false negative) rates across all four evaluated element types. Baseline and Topline values are unavailable for Qubus because its source code is confidential and could therefore not be provided to third parties.

Input system	**Normalised score**			**hallu%**			**missing%**		
	Baseline	C4Gen	Topline	Baseline	C4Gen	Topline	Baseline	C4Gen	Topline
C4Gen	0.38	0.85	0.93	15.0	0.0	0.0	50.9	9.2	0.0
AI4MDE	0.51	0.83	0.82	0.0	5.9	6.7	39.9	12.0	0.0
Abacus	0.62	0.92	0.96	0.0	0.0	0.0	23.5	0.3	0.0
Qubus	N/A	0.77	N/A	N/A	7.2	N/A	N/A	6.5	N/A

We conduct a total of eleven semantic evaluations across four different systems (one for the dogfooding case, four for AI4MDE, four for Abacus, and two for Qubus). For every input system, we take the mean of stakeholder-specific scores for every element type. Of these element-specific means, we can take the arithmetic mean to get the mean semantic score of an input system. The same can be done for the hallucination and missing rates. In Table 3, the mean semantic score, mean hallucination rate, and mean missing rate are given for each input system. As shown in the table, the system mean semantic scores range from 0.77 to 0.92. These results indicate that the generated diagrams generally align with what stakeholders expect to see in C4 container and component diagrams of the input systems. At the same time, we observe hallucinations in two of the four input systems' generated diagrams and false negatives for all systems, indicating that C4Gen does not generate fully defensible diagrams on its own, and human validation remains necessary. Tables 5–7 in the appendix show the element-specific, stakeholder-specific evaluation results.

Comparison with Generic AI Alternatives To position the syntactic and semantic quality of C4Gen's output against general-purpose AI coding assistants, we compare it against two alternatives: GitHub Copilot and Claude Code. Both tools can analyse source code repositories and can be instructed to retrieve architectural knowledge, but they are not specifically designed for these tasks as they are general-purpose assistants.

As a baseline, we instruct GitHub Copilot to create a C4 container diagram and corresponding component diagrams using a straightforward zero-shot prompting technique. We select `gpt-4o` as an LLM. This baseline gives an indication of an AI coding assistant's bare performance, without extending it with any additional logic or specifically designed agents.

As a topline, we also instruct Claude Code to create a C4 container diagram and corresponding component diagrams, but in this case, we use a refined Recursively Criticise and Improve (RCI) prompting technique [26]. We select Opus 4.7 as an LLM and enable Thinking mode in Claude Code. The combination of Claude Code's fully autonomous agentic planning and reasoning with an RCI prompting technique sets a powerful topline to compare C4Gen against.

For both the baseline and topline comparison, we evaluate the syntactic and semantic quality of the generated diagrams for three input systems (we cannot run the tools on Qubus due to confidentiality constraints). We conduct the evaluation ourselves and use the same evaluation metrics as before.

In contrast to C4Gen, GitHub Copilot did not always generate fully renderable container and component PlantUML-C4 strings. GitHub Copilot had difficulties in correctly generating package importing logic and sometimes used invalid data types. Claude Code did consistently generate renderable PlantUML-C4 strings for each input system, suggesting that a deterministic phase is not a hard requirement for generating valid syntax. Table 3 shows the mean semantic scores, mean hallucination rates, and mean missing rates for the different input systems, both for the baseline and the topline

comparison. We observe a mean semantic score range from 0.38 to 0.62 for the baseline and a range from 0.82 to 0.96 for the topline. For both the baseline and the topline, we observe hallucinations in one input system. The baseline has a higher missing rate, and the topline has no false negatives at all. These results indicate that, under the used evaluation methodology, C4Gen's output has a higher semantic quality than the baseline, but a slightly lower quality than the topline.

Practical Applicability We evaluated the applicability of C4Gen in a practical context by conducting feedback sessions with five IT consulting practitioners from the Dutch branch of an international consultancy firm (KPMG NL), who have experience in the field of software development and tech due diligence. We collected qualitative responses to open-ended questions about perceived usefulness, ease of use, and intention to use, inspired by the Technology Acceptance Model [27]. Four of five respondents reported a positive overall impression of C4Gen, while one respondent was neutral. Additionally, four respondents indicated that the generated diagrams (partially) covered the repository that they uploaded. All five respondents could foresee using C4Gen in their operations, suggesting practical relevance.

The responses indicate that practitioners would use C4Gen as a support tool for accelerating architectural analysis. Three respondents stated that the tool could improve their productivity, and four stated that it could improve their effectiveness. All respondents considered the tool easy to use.

However, two respondents raised concerns about output consistency, and several improvement opportunities were identified, including support for identifying cross-container-component relationships, improved export options, versioning, and better UI/UX. Overall, the feedback confirms that C4Gen should be positioned and used as a human-in-the-loop support tool rather than as a fully autonomous architecture recovery agent.

5. Discussion

Interpretation of Results The evaluation results indicate that C4Gen can recover C4 container- and component-level architectural knowledge from software repositories. Across the evaluated input systems, the generated diagrams generally aligned with stakeholder expectations, which suggests that the proposed workflow can identify higher-level architectural structure. In comparison with general-purpose AI coding assistants, C4Gen outperforms the GitHub Copilot baseline in terms of syntactic and semantic quality, which suggests that decomposing architecture recovery into smaller agentic steps increases performance over zero-shotting an LLM. However, C4Gen scores slightly lower than the Claude Code topline, suggesting that there is a trade-off between maximising semantic quality and preserving user control and traceability throughout the recovery process. Taken together, these findings indicate that C4Gen is effective as a human-in-the-loop support tool for architecture recovery, but not as a fully autonomous recovery agent. This supports our philosophy that in explainable AI-driven architecture recovery, intermediate judgments should be visible and editable by practitioners.

Strengths and Weaknesses of the Approach C4Gen appears to perform more strongly on recovering structural elements than on recovering the relationships between structural elements. Input project stakeholders classify most identified containers and components as a *match* or a *valid*, whereas relationships are classified more frequently as *irrelevant*, *hallucinated*, or *missing*. We assume this to be caused by the fact that structural elements are often reflected in directory structures and usually are defined explicitly in source code, whilst relationships may depend on more implicit knowledge that cannot always be recovered directly from source code.

There are several broader limitations of our approach. Firstly, C4Gen provides no support for recovering cross-container-component relationships and dependencies on external systems. Additionally, our framework only takes single repositories as input. C4Gen's output can be inconsistent, especially for lower-level decompositions where there might be multiple valid architectural interpretations and nuances. Finally, our human-in-the-loop approach prevents the amplification of inaccuracies in our pipeline, but this comes at the cost of speed, as users are forced to validate intermediate judgments.

Viewpoint Dependence During evaluation, stakeholders did not always agree on which elements and relationships are expected in a C4 diagram. We propose that this should not be regarded solely as a C4Gen-specific insight, but rather as a reflection of the fact that architectural descriptions are shaped by stakeholder concerns and their expected abstraction of a system's architecture. Prior work has long recognised that different architectural views are needed to address different informational needs and levels of abstraction [28, 10, 29]. From this perspective, variation in classification might not only be caused by disagreement, but also by a difference in expectations.

Threats to Validity Several threats to validity should be considered when interpreting our findings. Firstly, the evaluation was conducted on a limited number of input systems, stakeholders, and practitioners. Semantic evaluation was done on only four systems, for a total of eleven evaluations, and pragmatic feedback sessions were conducted with only five practitioners. This constrains the extent to which the results can be generalised.

The semantic evaluation of C4Gen's output relies on stakeholder judgments, which are subjective and may be influenced by differences in architectural viewpoints. Also, the baseline and topline evaluations were done by the authors, and thus can only be used as indicative reference points due to possible evaluator bias. The semantic scoring formula that was used for semantic evaluation introduces a construct validity threat, as the weighting scheme incorporates a fixed set of weights. Different weighting choices could lead to different normalised semantic scores that potentially affect the relative comparison between C4Gen and the baseline and topline alternatives.

The evaluation was only done for one specific implementation configuration. We did not conduct a systematic study into the effects of LLM selection, hyperparameter configurations, or prompting variations. This is particularly relevant because C4Gen was evaluated using the older `gpt-4o` model, whereas the topline used the newer `Opus 4.7` model. As a result, the comparison does not isolate the effect of C4Gen's workflow from the effect of the underlying LLM, and C4Gen's slightly lower performance may partly reflect the use of an older model rather than a limitation of the framework itself. As a result of these threats to validity, our findings can only be interpreted as evidence for the feasibility of our proposed approach, rather than as a definitive assessment of its performance.

Implications for Software Evolution Our findings indicate that LLM-driven approaches may be used to enable the automatic extraction and structuring of architectural insights. Such approaches reshape how architectural knowledge is constructed and interpreted. The outputs of agentic architecture recovery solutions should not be interpreted as a ground-truth representation of a system, but as an automatically generated view of a system's architecture that is shaped by available input data and agentic reasoning. This shift has implications for the role of practitioners. Traditionally, architectural work encompasses both the construction and validation of architectural representations. With agentic tools, part of the construction process is automated, but the validation aspect remains with the architect (and becomes more important). Consequently, the role of a practitioner may shift towards a role that is more focused on the evaluation and refinement of AI-generated artefacts.

6. Conclusion and Future Work

Conclusion In this paper, we presented C4Gen, a traceable human-in-the-loop framework for recovering C4 container- and component-level architectural knowledge from software repositories. C4Gen combines LLM-driven analysis techniques with deterministic diagram generation and intermediate human validation steps. Our evaluation indicates that the framework consistently generates syntactically correct diagrams that generally align with stakeholder expectations and that the framework is practically applicable in an IT consulting context. C4Gen improves over a generic zero-shot baseline (GitHub Copilot), but scores slightly lower than a fully autonomous agentic topline (Claude Code). Our findings indicate that LLMs can reliably support architecture recovery when embedded in a workflow that preserves traceability and human oversight.

Future Work Future work could investigate how to make C4Gen more broadly applicable. Firstly, the framework could be extended with support for cross-container-component relationships and external dependencies. Secondly, the tool could be extended to support multi-repository analysis, and research could be conducted on how the framework could create corresponding C4 system context diagrams. Additionally, the robustness of C4Gen could be investigated more thoroughly by evaluating the tool for more input systems, more LLM configurations, and different prompting strategies. For example, future research could investigate C4Gen's performance if the framework is combined with the Opus 4.7 LLM and an RCI prompting technique, because such a configuration showed promising results in the topline study. Finally, the framework could be extended with smaller functionalities that practitioners reported to be missing, such as more export options and the inclusion of versioning.

7. Declaration on Generative AI

During the preparation of this work, the authors used Microsoft Copilot and Grammarly for grammar and spelling checks. After using these tools, the authors reviewed and edited the content as needed. The authors take full responsibility for the publication's content.

References

[1] V. Chomal, J. Saini, Significance of software documentation in software development process, International Journal of Engineering Innovation and Research 3 (2014) 410–416.

[2] A. Jansen, P. Avgeriou, J. S. van der Ven, Enriching software architecture documentation 82 (2009) 1232–1248. doi:10.1016/j.jss.2009.04.052.

[3] S. A. Rukmono, L. Ochoa, M. R. V. Chaudron, Achieving high-level software component summarization via hierarchical chain-of-thought prompting and static code analysis, in: 2023 IEEE International Conference on Data and Software Engineering, ICoDSE 2023, ICoDSE 2023, IEEE, Piscataway, NJ, USA, 2023, pp. 7–12.

[4] S. A. Rukmono, L. Ochoa, M. R. V. Chaudron, Deductive software architecture recovery via chain-of-thought prompting, in: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER'24, Association for Computing Machinery, New York, NY, USA, 2024, p. 92–96. doi:10.1145/3639476.3639776.

[5] M. Perc, Counterfeit judgments in large language models, Proceedings of the National Academy of Science 122 (2025) e2528527122. doi:10.1073/pnas.2528527122.

[6] Z. Xu, S. Jain, M. Kankanhalli, Hallucination is inevitable: An innate limitation of large language models, 2025. URL: https://arxiv.org/abs/2401.11817. arXiv:2401.11817.

[7] Nature Machine Intelligence, Multi-agent ai systems need transparency, Nature Machine Intelligence 8 (2026). doi:10.1038/s42256-026-01183-2, editorial.

[8] S. Brown, C4 Model, 2018. URL: https://c4model.com/, accessed: Jun 3, 2026.

[9] S. Y. van Dam, C4gen: A tool for automatically generating c4 model container and component diagrams from software repositories., 2026. URL: https://github.com/syvandam/c4gen, github repository. Accessed: Jun 10, 2026.

[10] ISO/IEC/IEEE, ISO/IEC/IEEE 42010:2022 – software, systems and enterprise — architecture description, 2022. URL: https://www.iso.org/standard/74393.html, accessed: Jun 8, 2026.

[11] D. van Heesch, Doxygen, 2011. URL: https://www.doxygen.nl/index.html, accessed: Jun 3, 2026.

[12] Oracle Corporation, Javadoc, 2017. URL: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html, accessed: Jun 3, 2026.

[13] L. Vainigli, Plantuml generator, 2022. URL: https://github.com/lorenzovngl/plantuml-generator, gitHub repository. Accessed: Jun 3, 2026.

[14] D. Yang, A. Simoulin, X. Qian, X. Liu, Y. Cao, Z. Teng, G. Yang, DocAgent: A multi-agent system for automated code documentation generation, in: P. Mishra, S. Muresan, T. Yu (Eds.), Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics, volume 3, ACL,

Association for Computational Linguistics, Vienna, Austria, 2025, pp. 460–471. doi:10.18653/v1/2025.acl-demo.44.

[15] Object Management Group, Unified Modeling Language, Technical Report UML 2.5.1, Object Management Group, 2017. URL: https://www.omg.org/spec/UML/, formal specification, version 2.5.1.

[16] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, in: P. Isabelle, E. Charniak, D. Lin (Eds.), Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, ACL '02, Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 2002, p. 311–318. doi:10.3115/1073083.1073135.

[17] C.-Y. Lin, ROUGE: A package for automatic evaluation of summaries, in: Text Summarization Branches Out, Association for Computational Linguistics, Barcelona, Spain, 2004, pp. 74–81.

[18] R. Tseng, J. M. W. Visser, C. J. Stettina, Exploring Large Language Models in IT Audit: Improving Efficiency in Control Testing, Master's thesis, Leiden University, Leiden, The Netherlands, 2025. URL: https://theses.liacs.nl/3623.

[19] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, D. Zhou, Chain-of-thought prompting elicits reasoning in large language models, in: S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, A. Oh (Eds.), Proceedings of the 36th International Conference on Neural Information Processing Systems, volume 35, Curran Associates Inc., Red Hook, NY, USA, 2022, pp. 24824–24837.

[20] D. Amalfitano, M. De Luca, T. Santilli, P. Pelliccione, A. R. Fasolino, Automated software architecture design recovery from source code using llms, in: V. Andrikopoulos, C. Pautasso, N. Ali, J. Soldani, X. Xu (Eds.), Software Architecture: 19th European Conference, ECSA 2025, Limassol, Cyprus, September 15–19, 2025, Proceedings, volume 15929 of *Lecture Notes in Computer Science*, Springer Nature Switzerland, 2026, pp. 73–89. doi:10.1007/978-3-032-02138-0_5.

[21] M. Soliman, E. Ashraf, K. M. K. Abdelsalam, J. Keim, A. P. S. Venkatesh, Llms for software architecture knowledge: A comparative analysis among seven llms, in: V. Andrikopoulos, C. Pautasso, N. Ali, J. Soldani, X. Xu (Eds.), Software Architecture: 19th European Conference, ECSA 2025, Limassol, Cyprus, September 15–19, 2025, Proceedings, volume 15929, Springer Nature Switzerland, 2026, p. 99–115. doi:10.1007/978-3-032-02138-0_7.

[22] S. Y. van Dam, G. Ramackers, J. Visser, D. Geurtjens, C4gen: Human-in-the-loop architecture recovery for software repositories (master's thesis), 2026. URL: https://doi.org/10.5281/zenodo.23920602.

[23] AI4MDE, Ai4mde core studio, 2025. URL: https://github.com/ai4mde/studio, github repository. Accessed: Jun 5, 2026.

[24] Kiesraad, Abacus: software voor verkiezingsuitslagen en zetelverdeling, 2026. URL: https://github.com/kiesraad/abacus, github repository. Accessed: Jun 5, 2026.

[25] Qubus, Qubus 7 | powerful insights, 2020. URL: https://www.qubussoftware.com/, accessed: Jun 5, 2026.

[26] G. Kim, P. Baldi, S. McAleer, Language models can solve computer tasks, in: A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, S. Levine (Eds.), Proceedings of the 37th International Conference on Neural Information Processing Systems, volume 36 of *NIPS '23*, Neural Information Processing Systems Foundation, Curran Associates Inc., Red Hook, NY, USA, 2023, pp. 39648–39677. doi:10.5555/3666122.3667845.

[27] F. D. Davis, User acceptance of information systems : the technology acceptance model (TAM), Working paper 529, University of Michigan, School of Business Administration, Division of Research, 1987. URL: https://hdl.handle.net/2027.42/35547.

[28] F. M. J. Pex, M. R. V. Chaudron, L. Ochoa, S. A. Rukmono, Exploring Software Explanation using Retrieval Augmented Generation, Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2024. URL: https://research.tue.nl/en/studentTheses/exploring-software-explanation-using-retrieval-augmented-generati/.

[29] P. Kruchten, The 4+1 view model of architecture, IEEE Software 12 (1995) 45–50. doi:10.1109/52.469759.

A. Task, approach, inputs, and outputs for different steps in the framework.

Step	Task	Approach	Inputs	Outputs
Container Identifier Agent	Identifies file paths of high-level configuration files, and based on the contents of those files, identifies C4 containers and container-to-container relationships.	Uses high-level information as context to recover the container-level structure of the system.	The system's directory tree.	A JSON list of containers, and a JSON list of container-to-container relationships. A container has a name, type, description, and root path.
Component Identifier Agent	Identifies C4 components.	Uses information about the parent container and the parent container's directory tree as context to recover the component-level structure of the parent container.	The parent's container metadata (name, description, type) and the parent container's directory tree.	A JSON list of components within the parent container. A component has a name, type, used programming languages/frameworks, description, and corresponding file paths.
Component Summariser Agent	Generates a textual summary of a component.	Uses metadata and source code to generate a summary.	The parent container's metadata (name, type, description) and directory structure, the component's metadata (name, description), and the contents (source code) of the component's files.	A textual summary (string) of the component, including its purpose, key functionalities, and relationships with other logic/components.
Container Summariser Agent	Generates a textual summary of a container.	Uses the summaries of child components to generate a summary (*summary-of-summaries* approach).	The container's metadata (name, type, description), the container's directory tree, and the child components' summaries.	A textual summary (string) of the container, including its purpose, key functionalities, and relationships with other containers and components.
Component Relationship Agent	Identifies component-to-component relationships.	Uses component summaries to identify component-to-component relationships.	The parent container's metadata (name, type, description) and summary, and the summaries of components that reside in the parent container.	A JSON list of component-to-component relationships within the parent container.
C4 Diagram Rendering	Generates C4 diagrams based on the recovered information.	Deterministic parsing to PlantUML strings, which are rendered to .SVG diagrams.	Containers, components container-to-container relationships, and component-to-component relationships.	One C4 container diagram and corresponding C4 component diagrams (SVG).
Architecture Document Creation	Aggregates diagrams and summaries in a final output artefact.	Deterministic parsing.	C4 container diagram, container summaries, C4 component diagrams, and component summaries.	A final Markdown file.

Table 4: Task, approach, inputs, and outputs for the different steps in the framework.

B. Stakeholder-specific evaluation results

	Evaluator	$S_{normalised}$ (cont.)	$S_{normalised}$ (cont.–cont. rel.)	$S_{normalised}$ (comp.)	$S_{normalised}$ (comp.–comp. rel.)	$S_{normalised}$ (arithmetic mean)
(C4Gen)	**evaluator mean**	**1.00**	**1.00**	**0.82**	**0.59**	**0.85**
	developer	1.00	1.00	0.82	0.59	0.85
(AI4MDE)	**evaluator mean**	**0.94**	**0.79**	**0.84**	**0.73**	**0.83**
	developer	1.00	0.57	0.83	0.82	0.80
	developer	1.00	0.57	0.86	0.65	0.77
	architect	1.00	1.00	0.84	0.69	0.88
	project manager	0.75	1.00	0.84	0.75	0.83
(Abacus)	**evaluator mean**	**0.98**	**0.97**	**0.86**	**0.88**	**0.92**
	observer	1.00	1.00	0.83	0.84	0.92
	reviewer	1.00	1.00	0.99	0.93	0.98
	reviewer	0.92	0.88	0.88	1.00	0.92
	reviewer	1.00	1.00	0.75	0.75	0.88
(Qubus)	**evaluator mean**	**0.75**	**0.65**	**0.83**	**0.86**	**0.77**
	developer	0.50	0.67	0.66	0.71	0.64
	developer	1.00	0.63	1.00	1.00	0.91

Table 5: Normalised semantic quality scores for different element types in C4Gen-generated diagrams, as evaluated by individual stakeholders. (*cont. = container; comp. = component; rel. = relationship*)

	Evaluator	*hallu%* (cont.)	*hallu%* (cont.–cont. rel.)	*hallu%* (comp.)	*hallu%* (comp.–comp. rel.)	*hallu%* (arithmetic mean)
(C4Gen)	**evaluator mean**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
	developer	0.0	0.0	0.0	0.0	0.0
(AI4MDE)	**evaluator mean**	**0.0**	**16.7**	**2.4**	**4.6**	**5.9**
	developer	0.0	33.3	0.0	7.4	10.2
	developer	0.0	33.3	0.0	7.4	10.2
	architect	0.0	0.0	0.0	3.7	0.9
	project manager	0.0	0.0	9.7	0.0	2.4
(Abacus)	**evaluator mean**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
	observer	0.0	0.0	0.0	0.0	0.0
	reviewer	0.0	0.0	0.0	0.0	0.0
	reviewer	0.0	0.0	0.0	0.0	0.0
	reviewer	0.0	0.0	0.0	0.0	0.0
(Qubus)	**evaluator mean**	**0.0**	**25.0**	**0.0**	**3.6**	**7.2**
	developer	0.0	50.0	0.0	7.1	14.3
	developer	0.0	0.0	0.0	0.0	0.0

Table 6: Hallucination rates for different element types in C4Gen-generated diagrams, as evaluated by individual stakeholders. (*cont. = container; comp. = component; rel. = relationship*)

	Evaluator	*missing%* (cont.)	*missing%* (cont.–cont. rel.)	*missing%* (comp.)	*missing%* (comp.–comp. rel.)	*missing%* (arithmetic mean)
(C4Gen)	**evaluator mean**	**0.0**	**0.0**	**8.8**	**28.1**	**9.2**
	developer	0.0	0.0	8.8	28.1	9.2
(AI4MDE)	**evaluator mean**	**5.0**	**16.7**	**8.4**	**18.0**	**12.0**
	developer	0.0	33.3	6.1	10.7	12.5
	developer	0.0	33.3	6.1	10.7	12.5
	architect	0.0	0.0	8.8	25.7	8.6
	project manager	20.0	0.0	12.5	25.0	14.4
(Abacus)	**evaluator mean**	**0.0**	**0.0**	**1.2**	**0.0**	**0.3**
	observer	0.0	0.0	0.0	0.0	0.0
	reviewer	0.0	0.0	0.0	0.0	0.0
	reviewer	0.0	0.0	4.8	0.0	1.2
	reviewer	0.0	0.0	0.0	0.0	0.0
(Qubus)	**evaluator mean**	**0.0**	**0.0**	**14.3**	**11.8**	**6.5**
	developer	0.0	0.0	28.6	23.5	13.0
	developer	0.0	0.0	0.0	0.0	0.0

Table 7: Missing (false negative) rates for different element types in C4Gen-generated diagrams, as evaluated by individual stakeholders. (*cont. = container; comp. = component; rel. = relationship*)