

Leiden University

ICT in Business and the Public Sector

C4Gen: Human-in-the-Loop Architecture Recovery for Software Repositories

Name:	Sven Yannick van Dam
Student ID:	s2634376
Date:	19/06/2026
1st supervisor:	Prof. dr. ir. J.M.W. Visser
2nd supervisor:	Dr. G.J. Ramackers
Company supervisor:	D. Geurtjens

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Einsteinweg 55
2333 CC Leiden
The Netherlands

Abstract

Background: Software Architecture Documentation (SAD) can be used as a tool for understanding complex software systems, and as a support tool for communication about such systems. Architecture recovery and the production of the corresponding SAD are time-consuming and largely manual processes that could be partially automated using AI-driven technologies. In an IT consulting context, where trust and traceability are of high importance, AI-generated SAD introduces risks such as inaccurate information, hallucinations, and regulatory non-compliance.

Aim: In this research, we aim to develop a traceable, explainable, and programming language-agnostic human-in-the-loop framework that uses LLM-driven analysis techniques to generate C4 model insights about software repositories, which can be used in an IT consulting context.

Method: We investigate what architectural information can be extracted from repositories using LLMs, how this information can be structured into an intermediate representation, and how the retrieval and generation process can be automated within a human-in-the-loop framework. We evaluate the framework through syntactic, semantic, and pragmatic quality metrics, by combining quantitative evaluation of generated diagrams with qualitative feedback from industry experts.

Design: Our proposed framework, C4Gen, is a sequential pipeline of five AI agents that combines a top-down approach for identifying C4-level containers and components with a bottom-up approach based on source code summarization. A final deterministic step aggregates all generated insights into an SAD artifact. We integrate human validation after each agentic step to improve accuracy and traceability.

Results: We observe low syntactic error rates and acceptable semantic quality scores in C4Gen's output, indicating that generated diagrams are generally perceived as accurate and useful by stakeholders. Compared to a baseline (`GitHub Copilot`), C4Gen improves syntactic and semantic quality. C4Gen's output quality is comparable to that of a state-of-the-art fully autonomous agentic solution (`Claude Code`), while mutually providing improved traceability and user control. Evaluation by industry experts indicates that C4Gen improves productivity and supports architectural analysis, although manual validation remains necessary.

Conclusion: Our research indicates that AI-driven analysis techniques can be used to create a reliable and traceable human-in-the-loop workflow for automatic SAD generation in an IT consulting context. C4Gen provides a methodology for responsibly integrating LLMs into documentation processes, enabling more efficient extraction of architectural insights while maintaining trust and traceability. However, limitations remain in handling relationships, external dependencies, and variability across stakeholders and systems. Future work should focus on improving consistency and expanding architectural coverage by supporting broader use cases such as cross-repository analysis.

Keywords— C4 Model, Architecture Modeling, AI, Software Architecture Recovery, Software Architecture Documentation, LLMs, Software Engineering

Acknowledgments

This thesis would not have been possible without the outstanding supervision of Joost Visser, Guus Ramackers and Douwe Geurtjens. They have given me the freedom to explore my own research topic, while remaining highly committed to and actively engaged in the research process.

I would like to thank Joost for his frequent feedback and for helping me out when I encountered difficulties in defining quality metrics, Guus for his enthusiastic interest in my design and for continually encouraging me to improve it further, and Douwe for his time and effort in familiarizing me with the host organization, and for his valuable feedback on my research.

Secondly, I would like to thank KPMG NL as a host organization for enabling me to conduct my own research within a corporate setting. This research would not have achieved its current quality without the opportunities and connections provided by KPMG. I would like to express my particular gratitude to Dennis Stam for connecting me with the right people and for his strong interest in my research topic.

Additionally, I would like to thank everyone who contributed to the evaluation process of my research. Their time, insights, and feedback were essential in strengthening the quality of this work.

Furthermore, I would like to thank Folkert, Miguel, Bob and Tijn for making the past six months an enjoyable experience. The many coffee breaks and intern meetings were always a welcome opportunity to step away from the desk and take a break from the screen.

Finally, I would like to thank Sybe, Jort, Niels, Imran and Mark for their frequent collaboration throughout our Master's programme. They supported me in successfully completing all courses, and they made the two-year programme an engaging and enjoyable experience.

Contents

1 Introduction **5**
- 1.1 Problem statement . . . 5
- 1.2 Research questions . . . 6
- 1.3 Research relevance . . . 6
- 1.4 Research context . . . 6
- 1.5 Structure of the thesis . . . 7

2 Background and related work **8**
- 2.1 Software Architecture Documentation . . . 8
  - 2.1.1 Architecture descriptions . . . 8
  - 2.1.2 C4 model . . . 9
- 2.2 Documentation in IT consultancy . . . 11
  - 2.2.1 Different types of SAD in IT consulting . . . 12
  - 2.2.2 Modeling tools in IT consulting . . . 12
- 2.3 Regulatory drivers for SAD . . . 13
- 2.4 Large Language Models . . . 13
  - 2.4.1 Natural Language Processing . . . 14
  - 2.4.2 Self-supervised pre-training . . . 14
  - 2.4.3 Generative Pre-trained Transformer . . . 14
- 2.5 Prompt engineering and AI agents . . . 15
- 2.6 AI transparency . . . 17
- 2.7 Automated software architecture documentation . . . 17
  - 2.7.1 Bottom-up versus top-down approach . . . 18
  - 2.7.2 Automated SAD using LLMs . . . 18
  - 2.7.3 Existing tools . . . 21

3 Methodology **24**
- 3.1 Solution objectives . . . 24
  - 3.1.1 Requirements . . . 24
  - 3.1.2 Input project selection . . . 25
- 3.2 Design and development . . . 26
- 3.3 Evaluation methodology . . . 26
- 3.4 Deliverables . . . 32

4 Design **33**
- 4.1 Conceptual design . . . 33
  - 4.1.1 Individual steps in the design . . . 34
  - 4.1.2 Design decisions . . . 37
- 4.2 Proof-of-Concept: C4Gen . . . 40
- 4.3 C4Gen: demonstration . . . 42

5 Evaluation **45**
- 5.1 Baseline results (GitHub Copilot) . . . 45
- 5.2 Top-line results (Claude Code) . . . 46
- 5.3 Syntactic quality (C4Gen) . . . 48
- 5.4 Semantic quality (C4Gen) . . . 49

5.5 Output consistency (C4Gen) . . . 51
5.6 Pragmatic quality (C4Gen) . . . 52
5.7 Token usage (C4Gen) . . . 53

6 Discussion **54**
6.1 Quality of C4Gen output . . . 54
6.2 Solution limitations . . . 57
6.3 Applicability of C4Gen in IT consulting . . . 59
6.4 Threats to validity . . . 60
6.5 Reflection: mechanisms for AI-driven architecture recovery . . . 63
6.6 Reflection: software architecture in the age of agentic AI . . . 63

7 Conclusions **65**
7.1 Answers to the research questions . . . 65
7.2 Future work . . . 66

A Prompts **73**
A.1 Container information file paths identifier prompt . . . 73
A.2 Container identifier prompt . . . 74
A.3 Container relationship information file paths identifier prompt . . . 74
A.4 Container relationship identifier prompt . . . 75
A.5 Component identifier prompt . . . 76
A.6 Component summarization prompt . . . 77
A.7 Container summarization prompt . . . 77
A.8 Component relationship identification prompt . . . 78
A.9 Claude Code top-line comparison prompt . . . 79

B C4 Diagrams **81**
B.1 C4Gen . . . 81
B.2 Abacus . . . 87
B.3 AI4MDE . . . 91
B.4 Consistency study (C4Gen) . . . 99

C Semantic evaluation classifications **102**

D Survey responses **104**

Acronyms

AD Architecture Descriptions. 8, 58, 59, 63

AI Artificial Intelligence. 1, 5, 6, 8, 14, 16–18, 20, 22–25, 30, 33, 37, 39, 54–56, 58–60, 63–67

API Application Programming Interface. 22, 25, 37, 40, 48, 53, 58, 60

CI/CD Continuous Integration/Continuous Delivery. 60

CoT Chain-of-Thought. 15, 16, 19, 20, 40, 41

CRA Cyber Resilience Act. 6, 13

DSRM Design Science Research Methodology. 24

EU European Union. 6, 13, 17

GPT Generative Pre-trained Transformer. 14

ISO International Organization for Standardization. 8, 59, 63

IT Information Technology. 1, 5, 12, 13, 18, 21, 31, 32, 39, 52, 54, 58–60, 65–67

KPMG Klynveld Peat Marwick Goerdeler. 6, 40, 60

KPMG NL Klynveld Peat Marwick Goerdeler Netherlands. 6, 25

LLM Large Language Model. 1, 5, 8, 13–23, 25, 29, 30, 32, 35–41, 43, 54–56, 60–63, 65–67

MDE Model-Driven Engineering. 25

ML Machine Learning. 13, 14, 17, 19

NLP Natural Language Processing. 14, 18

POC Proof-of-Concept. 7, 20, 25, 26, 29–33, 37, 39–41, 45, 61, 62, 81–87, 89, 91, 93, 95, 97, 99–101, 105–108

RA Reference Architecture. 18

RAG Retrieval Augmented Generation. 21

RCI Recursively Criticize and Improve. 16, 30, 54, 55, 63

SAD Software Architecture Documentation. 1, 5, 6, 8, 11–13, 17, 18, 20, 21, 24, 25, 29, 33, 34, 36–38, 40, 42, 44, 53, 60, 62, 66

SAK Software Architecture Knowledge. 19

SALC Software Architecture Life Cycle. 18

SAR Software Architecture Recovery. 18, 20

SBOM Software Bill of Materials. 6, 13

SDLC Software Development Lifecycle. 21

SSL self-supervised learning. 14

SSPT Self-Supervised Pre-Training. 14

SWUM Software Word Usage Model. 18

TAM Technology Acceptance Model. 31

UI/UX User Interface/User Experience. 53, 60, 66

UML Unified Modeling Language. 9–12, 20, 22, 23, 25, 33, 34, 57

XML Extensible Markup Language. 40, 41

Chapter 1

Introduction

This chapter introduces our research by setting out the core concepts and structure. First, we introduce the problem statement in Section 1.1. We formally define the research questions in Section 1.2. Subsequently, we specify the research relevance in Section 1.3. In Section 1.4 we provide the research context. Finally, we outline the structure of the thesis in Section 1.5.

1.1 Problem statement

Software documentation, including Software Architecture Documentation (SAD), can improve stakeholder communication and system understanding in the context of software systems [11]. IT consultants often use architectural insights to support strategic decision-making and to communicate project risks and improvement areas to the client. In such a consulting context, it is important that architectural documentation is accurate and traceable, as trust is essential in the development of relationships between professional service practitioners and their clients [6]. Extracting architectural insights and producing accurate documentation is a time-consuming process that practitioners often do manually, and is not experienced as enjoyable [27]. Consultants are typically navigating through repositories manually, conducting stakeholder interviews, and reading existing documentation, all to gain a *helicopter overview* of a software system and its underlying relationships, such that they can translate these findings into architectural documentation. In an IT consulting context, especially in technical due diligence engagements, consultants work in short time-frames and only have limited time to explore the source code of a software system. Frameworks that speed up and assist architectural knowledge extraction can improve the effectiveness of such engagements.

In existing literature and tools, we did not identify evidence of an approach that *reliably* uses AI to extract architectural insights from repositories. Fully automatic AI-driven workflows that generate architecture documentation pose great risks in a consulting context, because falsely generated insights and unverified assumptions might be introduced in documentation products that are sent out to clients, as hallucinations remain a persistent risk when using a Large Language Model (LLM) [74]. Due to this, such fully automatic approaches are challenging, especially in a consultancy context, where correctness and traceability are a must.

In this research, we will address the lack of an explainable and traceable AI-driven framework that can be used in a consulting context. We will develop a framework that leverages LLM-driven analysis techniques to reliably create a structured C4 model [8] representation of an input repository, which can be used by consultants to create SAD. Consultants can use such a framework to validate generated insights against a project's repository and alter the insights if needed. The final, validated model can subsequently be used to create extensive SAD. The workflow will leverage LLMs to automatically gain container-level and component-level insights of an input repository. We focus on container- and component-level views because they are derivable from source code and most relevant in short IT consulting engagements. The framework aims to bridge the gap between low-level software implementations and high-level architectural representations in a manner in which consultants remain in control.

1.2 Research questions

This research aims to propose a framework that consultants can use to reliably use AI-driven analysis techniques for SAD. Such a framework can be developed by answering the following research question:

Research question

RQ. "How can a framework be developed that uses AI-driven analysis techniques to generate a structured and traceable C4 model representation of architectural structure, that supports consultants in producing reliable software architecture documentation?"

To be able to find a concrete answer to the research question, four different sub-questions should be answered:

- **SQ1.** "What raw data can be extracted from a software system by Large Language Models to establish a C4-level understanding of the system?"
- **SQ2.** "How can extracted data be structured into an intermediate and traceable representation that links architectural concepts to specific source code locations?"
- **SQ3.** "How can the automatic process of creating a structured and intermediate representation of architectural information be adapted into a reliable human-in-the-loop framework?"
- **SQ4.** "To what extent does the proposed human-in-the-loop framework improve a consultant's workflow of creating architectural documentation?"

1.3 Research relevance

Introducing a framework as described above is important for practical reasons. Consultants spend a significant amount of time manually navigating through source code to identify a system's components and functionalities. This manual process is time-consuming, inefficient, and difficult to reproduce for multiple projects. By introducing a framework that partially automates this process, consultants can spend more time on higher-level architectural reasoning than on low-level code navigation. The increasing availability of AI technologies presents opportunities to accelerate architecture recovery. However, this potential acceleration comes with explainability and traceability risks. Thus, a new framework must securely take into account these considerations.

Additionally, the introduction of an explainable and traceable framework is important for regulatory reasons. In 2027, the European Union (EU) enforces the Cyber Resilience Act [19]. The regulations that are introduced in this act provide cybersecurity standards for digital products within the EU, and will make it obligatory for software vendors to maintain a Software Bill of Materials (SBOM), which is *"a list of ingredients that make up software components"* [13]. Maintaining up-to-date SAD can support compliance by contextualizing dependencies, which can be useful during impact analyses.

1.4 Research context

We conduct this research as part of a research internship at the Tech Delivery team at KPMG NL. This team specializes in the delivery processes of software products and provides technical due diligence services for clients. Additionally, the team develops and maintains custom software for KPMG clients, and is interested in the (partial) automation of their architecture recovery process. We conduct this research over a six-month period in which a proposed framework is developed iteratively in collaboration with stakeholders within the Tech Delivery team. The research contributes to the academic field of software architecture and software engineering by proposing and evaluating a human-in-the-loop automated architecture recovery framework. The research builds further on existing literature on architecture documentation and architecture recovery by introducing a framework that can recover high-level architectural patterns from software repositories. Furthermore, this research contributes to the academic field of how to responsibly use AI in the software development life cycle. By following design research principles,

this research provides a reusable framework that can be applied to different systems within different organizations, rather than to only one case study.

1.5 Structure of the thesis

The thesis encompasses seven chapters:

- In **Chapter 1** (this chapter), we describe the problem statement and provide the research questions. Additionally, we state the relevance of the research and provide the research context.
- In **Chapter 2**, we describe relevant background information and related work for the thesis.
- **Chapter 3** explains the methodology and evaluation metrics we use in the research.
- In **Chapter 4**, we discuss the design of the proposed framework and demonstrate our corresponding Proof-of-Concept (POC): C4Gen.
- In **Chapter 5**, we show the quantitative and qualitative results of the evaluation of the proposed framework on real-world software systems, and additionally provide the same evaluation results for a baseline and top-line comparison.
- Subsequently, **Chapter 6** interprets these results and discusses the implications of these interpretations. Additionally, we enumerate the solution limitations and provide threats to validity of the research.
- Finally, in **Chapter 7** we answer the research questions, and provide possible directions for future research.

Chapter 2

Background and related work

This chapter provides background information on concepts that must be understood to be able to understand the research. Additionally, we describe state-of-the-art techniques and tools in the field of automated architecture recovery and SAD creation. First, we provide background information on SAD in Section 2.1, specifically on the concept of Architecture Descriptions (AD) and the C4 model framework. Subsequently, we provide background information on the current practices of software documentation in the context of IT consultancy in Section 2.2. Next, we explain regulatory drivers for automating architecture recovery and SAD creation in Section 2.3. In Section 2.4, we provide a concise explanation of LLMs and their underlying mechanisms. In Section 2.5, we explain prompt engineering and AI agents. Next, in Section 2.6, we give a definition of what it means for an AI system to be transparent. Finally, in Section 2.7, we enumerate state-of-the-art techniques and tools that are used for automatic architecture recovery and SAD creation.

2.1 Software Architecture Documentation

Software Architecture Documentation (SAD) artifacts aim to express and communicate software Architecture Descriptions (AD). ADs are practices, techniques, or representations that illustrate the architecture of a software system. There are multiple standards and frameworks that software architecture practitioners use.

The International Organization for Standardization (ISO) has specified the ISO/IEC/IEEE 42010 [32] standard that describes the international standards for ADs of software systems. This standard is widely used by system developers and enterprise architects. The standard defines concepts, terms, and practices for ADs which can be used to create a clear and standardized view of systems.

Another framework that is widely used in the software architecture domain is the C4 model [7, 8]. The C4 model is a graphical notation technique that is used to model system architectures in a graphical manner.

2.1.1 Architecture descriptions

The ISO/IEC/IEEE 42010 standard is an international standard for ADs in systems and software engineering. According to the ISO, an AD is a *"description that is used to express architectures for systems of interests"* [32]. They can be used to express the state of a system, as well as its evolution. The standard specifies that complex systems can have multiple stakeholders who each have individual interests in the architecture of a system. Such concerns can be expressed using *architecture viewpoints*, similar to how viewpoints are defined in Kruchten's $4+1$ views model [39]. The conceptual idea of the standard is that ADs should capture stakeholders' concerns and present a system's architecture through the lens of the stakeholder to which the AD is presented. For example, a developer's viewpoint of a software system may be more focused on structure and interfaces, whereas a security engineer's viewpoint may emphasize access controls and vulnerabilities. The ADs that are presented to these different stakeholders should match their roles and concerns.

2.1.2 C4 model

The C4 model is a graphical notation technique that Simon Brown introduced formally in 2018 [7]. The model aims to be an *"easy to learn, developer-friendly approach to software architecture diagramming"* [8]. The framework can be used to decompose a software system into a set of four hierarchical abstraction layers that describe the system. Each abstraction layer is accompanied by a diagram that is used to visualize the layer for stakeholders. Brown introduced the framework as a simplification of the Unified Modeling Language (UML) [46], which mutually takes into account Kruchten's 4+1 views philosophy [39], as the four different layers and their corresponding diagrams are intended for different stakeholders. The hierarchical relationship between the four layers is illustrated in Figure 1. The C4 model decomposes a system into the following four hierarchical abstraction layers:

- **Software system.** *"A software system is the highest level of abstraction and describes something that delivers value to its users, whether they are human or not."* [8]
- **Container.** *"A container represents an application or a data store. A container is something that needs to be running for the overall software system to work."* [8] Examples of containers are databases, web applications, mobile apps, or file systems.
- **Component.** *"A component is a grouping of related functionality encapsulated behind a well-defined interface."* [8] In object-oriented programming languages, components are typically a set of classes or interfaces.
- **Code.** Code is the actual source code of the system. A code *element* in the C4 model is typically a reference to a certain code snippet or implementation within a component.

A C4 model can describe multiple systems. Each system is built using one or multiple containers. Each container is built up using components, which are implemented using actual source code.

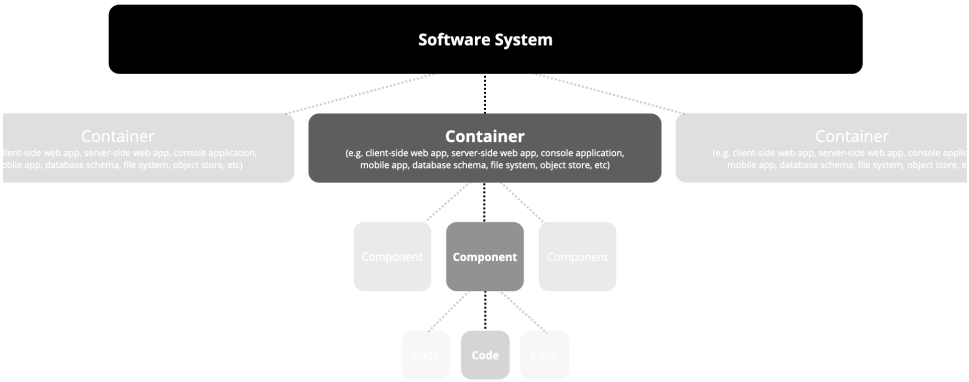


Figure 1: Hierarchical relationship between the four abstraction layers in the C4 model. Image by Simon Brown [8].

Each of the four abstraction layers can be visualized using a corresponding diagram notation:

- **System context diagram (software system layer).** This diagram is typically the starting point in documentation, and describes one software system and its interactions with other software systems. Additionally, the diagram may contain people (users, actors, roles, or personas) that interact with the system. System context diagrams are intended to be shown to a wide variety of stakeholders, including both technical and non-technical people who can be either inside or outside of a development team. An example of a system context diagram is shown in Figure 2.
- **Container diagram (container layer).** When systems have been visualized, the next step is to visualize the containers within a system. In a container diagram, the containers and their underlying interactions within a system are shown. Additionally, interactions with external systems and people are shown. Containers can have different icons to illustrate what kind of container they are; for example, a database container is visualized using a different icon than a backend container. The intended audience of a container diagram is technical people, both inside and outside the development team. An example of a container diagram can be seen in Figure 3.

- **Component diagram (component layer).** A container can be further decomposed by specifying its components in a component diagram. They show the responsibilities of individual components inside a container, and can also show the implementation details of these components and what technology is used for these implementations. Relationships between individual components are shown using internal connections, and external connections indicate the relationships between components and external containers or software systems. Component diagrams are intended to be shown to technical people inside the development team, for example, software architects and developers. Component diagrams should only be added to documentation if they provide actual value to stakeholders. An example of a component diagram can be seen in Figure 4.
- **Code diagram (code layer).** The implementation details of an individual component can be visualized using a code diagram. In the C4 model, there is no standardized method to visualize code. Instead, different notation techniques can be used, such as UML class diagrams or entity relationship diagrams. Code diagrams are intended to be shown to technical stakeholders, primarily software developers in the development team. Code diagrams are often left out of documentation, as they must be changed frequently and usually do not provide more value than analyzing source code manually. An example of a code diagram can be seen in Figure 5.

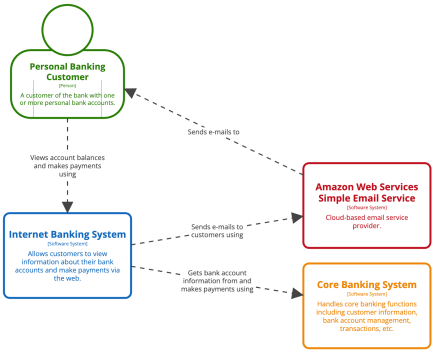


Figure 2: Example of a C4 system context diagram. Image by Simon Brown. [8]

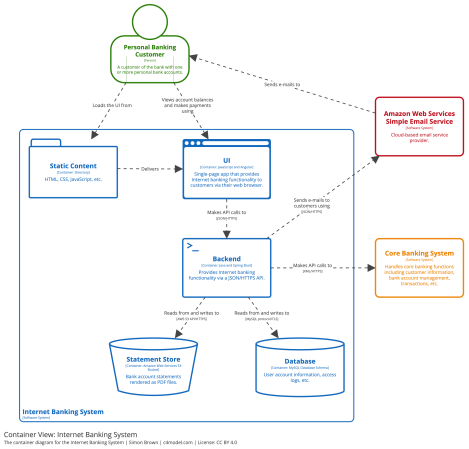


Figure 3: Example of a C4 container diagram. Image by Simon Brown. [8]

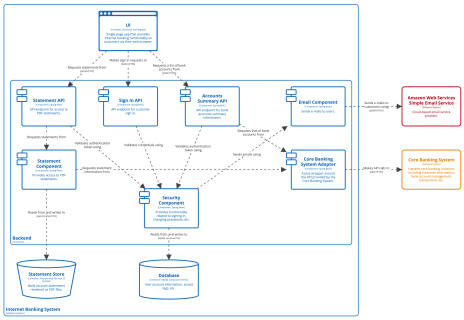


Figure 4: Example of a C4 component diagram. Image by Simon Brown. [8]

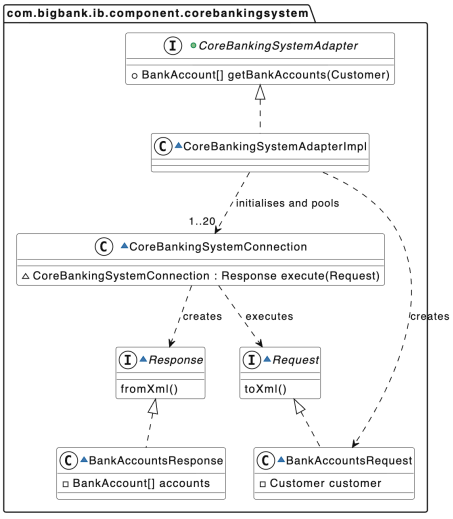


Figure 5: Example of a C4 code diagram. Image by Simon Brown. [8]

System context, container, component, and code diagrams all describe the static structure of the cor-

responding hierarchical layers in a software system. To be able to show the dynamic perspectives of a software system, three additional types of diagrams are introduced in the C4 model:

- **System landscape diagram.** This is a broader version of system context diagrams. Whereas one system context diagram aims to describe the context of an individual software system and its interactions with stakeholders and other software systems, a system landscape diagram aims to show the entire landscape of systems in a single diagram. System landscape diagrams are typically used for both technical and non-technical people, and can be used to describe enterprise architectures.
- **Dynamic diagram.** These diagrams show the collaboration at runtime between the static elements in the structural diagrams. Dynamic diagrams are similar to UML sequence and communication diagrams. They should only be added to documentation if the documentation is required to show complex sets of interactions between systems, containers, and components.
- **Deployment diagram.** These diagrams show how the elements that are illustrated in the structural diagrams are deployed on an organization's infrastructure, similar to UML deployment diagrams. C4 deployment diagrams introduce the concepts of deployment nodes and infrastructure nodes, which are respectively used to illustrate where elements are located in the infrastructure and how these deployments communicate with each other over the network (e.g., network bridges, load balancers, firewalls). It is recommended to add deployment diagrams to deployment documentation that is intended to be shown to technical people from both inside and outside the development team, ranging from software architects and developers to infrastructure architects and operational staff.

2.2 Documentation in IT consultancy

In the 2000s, the academic field of SAD has shifted from the creation of technical frameworks and methodologies towards a more stakeholder-centric paradigm, which is more concerned about the practical implications and challenges of SAD in organizations [71]. In 2009, Jansen et al. [33] identified a number of challenges of SAD in practice:

- **Understandability.** The audience of SAD might have a different perspective on the content of the documentation than the perspective and intentions of the author. Such a misalignment seems to occur more frequently when systems are larger and when there are stakeholders with different backgrounds [33].
- **Finding relevant information.** When SAD is extensive and knowledge is spread over multiple documents, stakeholders spend a significant amount of time searching for relevant information. To reduce this workload, SAD should be structured clearly and might make use of outlines and glossaries.
- **Traceability.** It is often unclear how SAD documents are related to software requirements. Figures tend to be better at communicating the relationships between documents than textual or tabular techniques. Such figures could, however, decrease understandability [33].
- **Making reliable impact analyses.** Jansen et al. [33] describe how SAD usually lacks explicit communication on how architectural design decisions might impact an overall system. For developers and technical architects, such knowledge is essential to be able to make proper design decisions.
- **Evaluating SAD maturity.** It is often difficult to assess whether SAD is correct and complete [33].
- **Trust.** In practice, SAD is often outdated because of iterative development. The creation and updating of SAD is usually not experienced as enjoyable by stakeholders [27]. Additionally, writing SAD often does not provide direct value to an organization's client, but is rather an internal supportive tool that is costly to maintain. If SAD is not up-to-date because it has not evolved together with a software system, stakeholders lose their trust in the documentation [33].

In professional services, trust is essential for building relationships with clients [6]. To maintain trust, artifacts that are provided to clients must be accurate and reliable. Some public scandals have occurred

in big consulting firms, which were caused by the inclusion of hallucinated information in documents that were sent to clients, resulting in large fines [14] and a damaged brand image. To prevent similar risks in the future, it is important that SAD is accurate and traceable if it is used to make important decisions that have an impact on clients.

2.2.1 Different types of SAD in IT consulting

Consulting firms typically have two different types of SAD: documentation that is sent out to external stakeholders as part of due diligence engagements or IT audits, and documentation that is used by internal stakeholders for project development.

SAD used in due diligence/audits. In technical due diligence engagements/IT audits, an organization's software systems, processes, or IT assets are examined to determine whether the organization adheres to specified standards [26]. Such standards could be in the context of security, performance, compliance, or effectiveness. Communication in IT audits can be supported by the inclusion of SAD, e.g., by modeling a system's as-is and to-be situation, or by highlighting specific software components that are vulnerable to security issues. IT systems on which an audit is conducted can be legacy systems for which documentation is lacking. The goal of SAD in IT audits is to illustrate structure and patterns in a software system in an abstract, non-technical manner, such that it can be used to inform decision-making. This type of documentation is sent out to client stakeholders who are not part of the consulting firm, and thus, such documentation should be accurate and reliable.

Internal SAD for development. Besides audits, IT consulting firms might also develop software systems for clients. These systems can grow in complexity, which is when the creation and maintenance of SAD might become relevant as a supportive tool in the development process. This internal SAD is more technical in nature, and often is not sent out to clients but rather used to guide design decisions that are made by developers or architects. Even though this SAD might not be directly sent out to external stakeholders, it is still important that the content is accurate and reliable because it might influence decision-making that indirectly still has an impact on clients.

As shown above, the type of SAD that is used in a project does not determine whether the documentation must be accurate and reliable. Inaccurate SAD that is directly sent out to clients may have direct, severe consequences, and these consequences may still be present in the case of internal SAD. Thus, stakeholders within an IT consultancy firm should strive for accurate and reliable SAD at all times.

2.2.2 Modeling tools in IT consulting

IT consultants can use modeling tools for various use cases. Several tools and frameworks exist, each of which all have their own advantages and disadvantages.

UML is a set of different diagramming notations that can be used to describe software systems. They can be used to model technical details, but can grow complex and be difficult to comprehend by non-technical stakeholders. Such diagrams can be overwhelming when shown to clients without a technical background.

To mitigate this problem, IT consultants sometimes manually model software systems using tools such as Microsoft PowerPoint. Such tools offer great flexibility, but lack standardization and automation.

IT consultants can use the C4 model as a modeling technique that combines the advantages of UML with the advantages of more flexible tools such as PowerPoint. As described in Section 2.1.2, the C4 model can be used for different abstraction layers. These abstraction layers have different purposes and audiences, and thus can be applied in different use cases in an IT consulting context:

- **System context diagrams** describe the interactions of different personas with a system, as well as the system's interactions with external systems. In an IT consulting context, these types of diagrams are especially interesting when trying to map a system's functional position within a context. This is generally done in the enterprise architecture domain, where consultants gather data from multi-modal information streams (e.g., interviews, business process documentation), rather than only analyzing source code. Thus, it is difficult to create such diagrams from technical source code only, as human factors must be taken into account.

- **Container diagrams** are of a more technical nature and describe the high-level, technical static structure of an individual software system. In an IT consulting context, these diagrams can be used as a supporting tool in communication about the architectural structure of a software system. Generally, these diagrams can be created using source code only, as they describe the technical structure of a repository.
- **Component diagrams** have a similar use case in an IT consulting context as container diagrams, but they are even more technical and low-level in nature. They are often used as a supporting tool in communication between developers, e.g., when software products are developed internally by an IT consulting organization.

2.3 Regulatory drivers for SAD

Besides the practical use cases of documentation in the software development life cycle, SAD is becoming increasingly relevant because of regulatory drivers. Modeling techniques are frequently used in compliance checking processes because they can be used to create artifacts that visualize systems or processes in an abstract way [10]. However, tools that can be used to automate and standardize compliance checking processes are lacking [10]. One of the primary regulatory drivers for maintaining up-to-date SAD is the EU Cyber Resilience Act (CRA).

The CRA [19] is a European cybersecurity policy that the European Commission entered into force in 2024. As of 2027, the obligations that are introduced as a result of the policy will be enforced across the EU. The goal of the CRA is to ensure that all digital products, both hardware and software, that are sold and used in European markets are safe from cyber attacks. The CRA introduces standards and requirements for manufacturers that cover the planning, design, development, and maintenance phases of products with digital elements [20]. Examples of such obligations are, among others:

- Carrying out risk assessments for products before going to market [22]
- Carrying out conformity assessment procedures [21]
- Writing technical documentation that indicates how products comply with cybersecurity requirements [22]
- Actively reporting vulnerabilities that have been exploited and cybersecurity incidents that have occurred in the organization [23]

By actively complying with these obligations, organizations can mitigate cybersecurity risks. The CRA should be seen as a policy that requires organizations to implement *minimal* required cybersecurity measures. Organizations are encouraged to invest more in security and to set cybersecurity measures that are more strict than those defined in the CRA.

To comply with the CRA, software vendors are required to create, maintain, and retain a Software Bill of Materials (SBOM) for their software products. SBOMs can be seen as an *ingredient list* for software products, that state what a product is made of. The CRA states that software vendors must *"identify and document vulnerabilities and components contained in products with digital elements, including by drawing up a software bill of materials in a commonly used and machine-readable format covering at the very least the top-level dependencies of the products"* [19]. Such an SBOM must be retained for at least ten years after the product has been introduced in the market [19]. Additionally, SBOMs must be kept up to date throughout the life cycle of the product [19]. SBOMs do not have to be disclosed publicly, but an organization must be able to provide them to regulatory authorities [19].

Whereas an SBOM is an ingredient list of software components, SAD captures the architectural structure and interaction between software components. SAD can provide context on how software components are used, which can help in understanding the impact of security vulnerabilities. By maintaining up-to-date SAD together with an SBOM, organizations may comply more easily with the CRA because they may conduct risk assessments and vulnerability analyses more effectively.

2.4 Large Language Models

A Large Language Model (LLM) is a type of Machine Learning (ML) model that is used to generate textual output data based on textual input data that was fed to the model. LLMs are widely used and

can be integrated into a diverse set of business operations. Examples of common LLM applications are chatting with an AI virtual assistant, writing assistance, code generation, and information retrieval.

This section will explain the relevant concepts that are needed to understand LLMs. First, we explain Natural Language Processing (NLP) in Section 2.4.1, which is the essence of what LLMs do. In Section 2.4.2, we describe the concept of Self-Supervised Pre-Training (SSPT), which is the method that is used to train LLMs. Finally, we discuss the Generative Pre-trained Transformer (GPT) architecture in Section 2.4.3, which is the architecture that is used in many state-of-the-art LLMs [42, 48].

2.4.1 Natural Language Processing

Natural Language Processing (NLP) is the processing of natural language by computers. NLP has a long history, dating as far back as the 1950s when Alan Turing introduced the Turing test [66]. The goal of NLP is to use computers to extract meaning from text or speech by enabling natural, human-like communication with computers. There are two approaches to NLP: the historical *symbolic* approach and the modern *statistical* approach.

In symbolic NLP, models are taught rules by explicitly defining grammar rules and vocabulary. Symbolic models are transparent because generated outputs can be clearly explained by backtracking the rules that have been applied. However, rules have to be written by hand and can grow in complexity exponentially, which makes symbolic models infeasible for real-world applications.

Statistical or *stochastic* approaches to NLP use statistical methods to calculate associations between words, making them more generally applicable. The growth of ML and neural networks greatly increased the capabilities of stochastic NLP models, making statistical methods the most widely used approaches to language processing using computers.

2.4.2 Self-supervised pre-training

ML models need to be trained to become useful. During the training process, the model is fit to its purpose by providing training data. During training, the model predicts output data based on input data, and this generated output data is compared against a ground truth. The parameters (decision points) in the model are subsequently adjusted based on these comparisons.

Modern LLMs are trained using Self-Supervised Pre-Training (SSPT). SSPT is a combination of self-supervised learning (SSL) and pre-training. In SSL, ML models are trained on tasks by finding patterns in unlabeled datasets and extrapolating such patterns into output. Pre-training is the first step of training ML models. During pre-training, models are trained on vast amounts of diverse data, aiming to make a model adaptable to a diverse range of tasks. After pre-training, a model can be fine-tuned for specific tasks by updating the model's parameters using task-specific data.

In LLMs, the concept of SSPT is applied by training models on a large and diverse textual corpus. State-of-the-art models are usually trained by using large amounts of publicly available internet content as training data [47]. This training data is fed to the LLM, which can predict textual output after training when it is provided with textual input. By using a large and diverse dataset as training data, such pre-trained models are widely applicable to a wide variety of tasks, including source code analysis and pattern recognition.

2.4.3 Generative Pre-trained Transformer

A Generative Pre-trained Transformer (GPT) is a type of LLM architecture that uses SSPT as a training mechanism. During training, the GPT is provided with vast amounts of diverse unlabeled data. This unlabeled data is used to train so-called *transformer* blocks, which are the core building blocks of GPTs and should be seen as blocks that can capture long-range dependencies between words in textual input data. The inclusion of such transformer blocks enables LLMs to capture long-range dependencies over a large input text, which prevents the model from failing to retrieve important information if the input size is large. Many state-of-the-art NLP models are based on transformer architectures like a GPT architecture. Notable models that are built using a GPT architecture are OpenAI's GPT-series, Meta's Llama-series, and the China-based DeepSeek models.

2.5 Prompt engineering and AI agents

LLMs can be useful tools that can create content quickly without the intervention of humans. However, to do so, an LLM must be instructed using explicit and effective prompts. Prompt engineering is the field of study that aims to write effective prompts, or *"the process of designing and refining input queries (...) to elicit desired responses from LLMs"* [44]. Prompt engineering is relevant to our research, because we will use LLMs to retrieve insights from input repositories. In order to do so effectively, we must write effective prompts. The process of creating effective prompts can be undertaken by following these five steps [44]:

1. Defining the goal of the prompt
2. Understanding the capabilities of the used LLM
3. Choosing the right format for the prompt
4. Providing relevant context to the prompt
5. Testing the prompt and making refinements

There are multiple, widely used techniques in the field of prompt engineering:

Zero-shot prompting

In zero-shot prompts, an LLM is directly instructed with a task without providing examples or additional context. The following is an example of a simple zero-shot prompt.

Prompt:

"Classify in which programming language the following code is written.
Code: <code>"

In simple tasks, LLMs that have incorporated the task-specific information into their parameters, zero-shot prompting may outperform more advanced prompting techniques [56].

One-shot prompting

A one-shot prompt aims to enhance a zero-shot prompt by providing a single example or demonstration on how to come up with a valid answer to a question or task. The following is an example of a simple one-shot prompt.

Prompt:

"Give the country in which the city is located.
Example:
City: Leiden -> Country: the Netherlands
Do this for the following:
City: New York"

One-shot prompts aim to simulate the concept of human learning, in which tasks can be taught by providing an example of how to perform a task.

Few-shot prompting

A few-shot prompt incorporates the same techniques used in one-shot prompts, but provides multiple examples. Few-shot prompts have been shown to be effective in modern LLMs [9].

Chain-of-Thought (CoT) prompting

In CoT prompting, an LLM is instructed to come up with an answer step-by-step. This technique aims to simulate human reasoning methods, in which problems are solved in sequential steps. The following is an example of a prompt that incorporates a CoT instruction. The CoT instruction is marked in red.

Prompt:

> "IT consultancy firm ABC has two teams: Team A with 12 available software developers, and Team B with 14 available software developers. The firm has put 4 developers from Team A and 8 developers from Team B on a project. How many developers are left in each team? Explain step-by-step."

CoT prompts have been shown to outperform other prompting techniques in complex tasks, such as arithmetic and reasoning tasks [72]. CoT can be combined with few-shot learning to come up with more accurate results [72].

Emotional prompting

Emotional prompting aims to simulate the concept of emotional manipulation into prompts. Polite or impolite language can be added to prompts, with the goal of steering the LLM into particular directions. The following is an example of emotional prompting. Emotional language is marked in red.

Prompt:

> "Please provide me with an explanation of the C4 software architecture model. It is important for me to know how the model works."

Emotional prompting has been shown to have an impact on the introduction of misinformation in LLM responses. The inclusion of impolite language in prompts can significantly amplify misinformation in responses when compared to the inclusion of neutral or polite language [70].

Critique prompting

In critique prompting, an LLM is instructed to criticize and improve its own output. Kim et al. [38] formally introduced the technique in 2023 by introducing the Recursively Criticize and Improve (RCI)-mechanism. In this mechanism, sequential prompts are fed to the LLM such that the output is improved. First, the LLM is specified with a task. Secondly, a *critique* prompt is fed that instructs the LLM to find problems with the previously generated output. Finally, an *improve* prompt is fed that instructs the LLM to improve its output based on the identified problems. Critique and improve prompts can be used multiple times (either for a limited number of times, or until a stop condition is met), making the technique a recursive technique. The following is an example of a sequence of RCI prompts.

Task prompt:

> "Implement a simple calculator app in Python."

Critique prompt:

> "Evaluate your answer and identify problems in your answer. Provide them in a list."

Improve prompt:

> "Based on the problems that you have identified, improve the answer that you have given."

The RCI-mechanism has been shown to outperform zero-shot and CoT in benchmark tasks [38].

AI agents

If conventional prompting engineering techniques do not suffice for more advanced use cases, AI agents may be used. There exists a wide diversity of definitions for AI agents or agentic AI. According to Kapoor et al. [35], AI agents are *"AI systems that can be instructed in natural language and act autonomously on the user's behalf"* and *"that require less user supervision"*. Additionally, such systems are systems *"whose control flow is driven by an LLM"* and they work in a complex environment. More broadly speaking, AI agents are systems that operate somewhat autonomously and adapt the steps they take if needed to

attain a goal. AI agents can be seen as *"ensembles of LLMs that interact with each other and have access to tools and databases and that can solve tasks in a multi-step way"* [45].

In 2025, Abou Ali et al. [1] conducted a literature study on current practices in the field of agentic AI, and they identified two distinct types of approaches for implementing AI agents: a *symbolic/classical* approach, and a *neural/generative* approach. In the symbolic approach, agents are guided using rule-based, deterministic techniques. Neural approaches are stochastic and prompt-driven, and almost always incorporate some deep learning model in their implementation.

An advantage of AI agents over conventional prompting techniques is the elimination of human decision-making, which speeds up the decision-making process. This does, however, come at the cost of possible amplification of LLM hallucinations and a decrease in traceability and ownership. Agentic systems are widely used in bigger organizations, but they are often difficult to implement and require active monitoring [59]. More importantly, the incorporation of agentic approaches in an organization's operations can introduce regulatory risks [12].

2.6 AI transparency

Because of the stochasticity and complexity of modern ML models and LLMs, such models are often regarded as *black boxes*, because they make decisions that are not fully transparent to human beings. Besides their stochastic nature, LLMs employ information pathways that are different than those in humans, which often leads to false, inaccurate responses by LLMs [52]. Active research is being conducted on how to make AI systems transparent and reliable. In 2019, the *High-Level Expert Group on AI* provided guidelines for the concept of *trustworthy* AI [24]. One of these guidelines mentions that AI systems should be *transparent*, which can be decomposed into three dimensions: (1) traceability, (2) explainability, and (3) communication.

According to the High-Level Expert Group on AI, traceability of an AI system is *"the capability to keep track of the system's data, development and deployment processes, typically by means of documented recorded identification"* [24]. AI traceability is an active field of study in which supportive tools are actively being developed [17, 30, 60].

Explainability is defined as *"the ability to explain both the technical processes of an AI system and the related human decisions"* [24]. Technical explainability is the concept that decisions that are made by systems that incorporate AI must be able to be understood by human beings. Explainability ensures that for choices that have been made and that have a significant impact on an individual's life, it can always be explained to the individual *how* these choices are made.

Communication refers to the concept that *"AI systems should not represent themselves as humans to users"* [24]. EU citizens should have the right to know if they are interacting with an AI system, and they should have the possibility to opt out of using such systems. If they do, organizations must provide alternative human-based services.

AI transparency is particularly important in the field of agentic AI. In multi-agent software solutions, false outputs and hallucinations can be amplified because an ensemble of LLM calls is used, rather than using one individual prompt [45]. Such false outputs can be more difficult to trace than in conventional AI methods because an agent's output can be used as input for other agents. Thus, it is of extra importance to ensure that users stay in the loop across all stages in agentic pipelines, and that decisions that individual agents make are logged and can be retrieved, and possibly are validated by humans.

AI transparency is of importance in our research because we will leverage LLMs to extract data from input repositories and parse such data into actual C4-level insights. If false data is retrieved, or C4-level insights are created that are not in line with a ground-truth, it is important that a human can deduce *where* and *why* LLMs introduced such errors in the process. By ensuring that the process is transparent (e.g., by logging LLM decisions and mapping decisions to specific source code), humans can derive the details of errors.

2.7 Automated software architecture documentation

Automated SAD is the domain of tools or frameworks that aim to document the architecture of software systems automatically, rather than writing documentation manually. There are generally two different

approaches to automated SAD: an inductive *bottom-up* approach and a deductive *top-down* approach. In Section 2.7.1 we explain the difference between these two approaches. In Section 2.7.2, we describe different studies that have been conducted in the field of automatically generating SAD using AI-driven techniques. Finally, we provide a list of existing SAD tools in Section 2.7.3, and describe their shortcomings and formally state the research gap.

2.7.1 Bottom-up versus top-down approach

In the bottom-up approach of automatic SAD creation, individual code snippets are analyzed automatically using (deterministic) techniques, and subsequently text is created for these individual snippets, which can further be used to create higher-level SAD. Bottom-up techniques have been around for decades [31, 50, 69], and often involve lexical parsers that parse code snippets into abstract syntax trees, which in turn can be used to generate textual documentation strings. Such tools are inflexible because they often need to be re-implemented for different programming languages. Moreover, they solely generate documentation for individual methods and fail to capture the more abstract, high-level architectural structure of a software system.

Recent advancements in the field of AI have facilitated research on deductive top-down approaches for automated SAD. Rukmono et al. [58] introduced a top-down framework for Software Architecture Recovery (SAR), which aims to validate a system's implementation against a Reference Architecture (RA). Globally, the following steps are followed in a top-down approach: (1) define or hypothesize a RA (2) define/identify the architectural components in a system, (3) break down the responsibilities of components and their interactions into a set of indicators, (4) evaluate the system's source code against the indicators, and finally (5) aggregate the results of the previous step. Whereas bottom-up approaches start at the code level of a system and iteratively work upwards to create SAD, top-down approaches work downwards by first setting the boundaries of what the documentation should look like, and then classifying components within a system against these boundaries. The top-down approach that Rukmono et al. [58] proposed uses LLMs to analyze source code, making it a stochastic approach that is more generally applicable to different programming languages. The framework was only used for SAR, and not for SAD creation. Because of the stochasticity of LLMs, output that is generated by top-down approaches may not always be accurate.

2.7.2 Automated SAD using LLMs

Generative AI, especially LLMs, has opened up possibilities in the field of software architecture. In the Software Architecture Life Cycle (SALC), LLMs are primarily used in the *requirement-to-code* and *architecture-to-code* phases [18]. Research in the domain of integrating LLMs in SAD is more niche, but interesting research has been conducted:

Source code documentation using NLP

Before LLMs were prevalent, research had already been conducted that aimed to automate source code documentation using more straightforward NLP techniques. In 2020, Arthur [4] proposed a framework that adopts a so-called Software Word Usage Model (SWUM). SWUM is a linguistic model that tries to retrieve the grammar within source code identifiers, e.g., within variable names, method names, and class names. Within such identifiers, SWUM retrieves verb, noun, and action relationships. These retrieved relationships can subsequently be used to build natural-language-like structures that can be used to generate software documentation. Even though this framework automates the software documentation process, the generated documentation is low-level and inflexible. The implementation only works for low-level C code, and the generated documentation is more of a duplicate of the source code that is written down in more human-like sentences. The implementation fails to capture high-level, abstract, architectural structure in complex projects, making the framework not applicable to enterprise-grade software solutions in an IT consulting context.

LLM comparison in source code summarization

In 2024, Sun et al. [62] conducted a study on the performance of source code summarization by four different LLMs (`CodeLlama-Instruct`, `StarChat-`β, `GPT-3.5`, and `GPT-4`). This study employs a bottom-up approach for software documentation, as documentation was only generated for individual code snippets, rather than generating high-level, abstract SAD. The effectiveness of source code summarization using

five different prompting techniques is explored (zero-shot, few-shot, CoT, critique, and expert). Finally, the influence of two hyperparameters is investigated. Experiments were conducted on datasets containing a vast amount of projects, including projects written in object-oriented programming languages as well as in functional and logic programming languages. It was found that *"among all automated evaluation methods, the GPT-4-based evaluation method overall has the strongest correlation with human evaluation"* [62]. Regarding prompting strategies, it was found that *"more advanced prompting techniques expected to perform better may not necessarily outperform simple zero-shot prompting"* [62]. Finally, it was found that *"across all five types of programming languages, LLMs consistently perform the worst in summarizing code written in logic programming languages"* [62].

LLM comparison in Software Architecture Knowledge (SAK)

SAK is the knowledge that stakeholders have about the architecture of a software project. In 2026, Soliman et al. [61] conducted research that compared the performance of seven different LLMs in answering SAK-related queries. An input dataset was used with 229 SAK-related questions and their corresponding ground-truth answers. These questions were submitted to the LLMs, whose responses were evaluated against the ground truth. The questions were written in a zero-shot style, without providing any extra instructions or examples that would not be given if the questions were asked to a human being. It was found that LLMs have a low precision but a higher recall for architecture-related questions. `DeepSeek` achieves better results than other larger models, and `TinyLLaMA` and `Qwen` had the worst precision and recall scores overall.

Mapping source code to software architecture using LLMs

In 2024, Johansson et al. [34] evaluated the performance of different classification techniques that aim to automatically map source code to architectural modules. In their research, descriptions of architectural modules were generated using a generative model, which were subsequently pre-processed and vectorized using a vectorization model. These vectors, together with labels of the architectural modules, were then used to train a classification ML model that aims to classify which architectural module a code snippet belongs to. During testing, code snippets were pre-processed and vectorized, which were then used to test the classifier model. Predicted labels were then compared to actual labels, giving accuracy and F1-scores. This process was repeated for different vectorization models and different classifier models. The researchers found that using LLMs for vectorization of text outperforms other methodologies. No significant difference in performance was found for different classifier models.

Architecture design recovery using LLMs

Amalfitano et al. [2] conducted a study that evaluates the performance of four different LLMs in three different tasks: (1) the identification of code-level class diagrams from source code input, (2) the identification of more high-level architectural structure and design patterns from source code input, and (3) the identification of architectural styles in software projects. The LLMs were evaluated for hallucinations (false positives), mistakes (incorrect identifications), and missings (false negatives). The LLMs that were evaluated were OpenAI's `GPT-4o`, Google's `Gemini 1.5 Pro`, Anthropic's `Claude 3.5 Sonnet`, and Mistral's `Mistral Large`. A self-reflection prompting technique was used to encourage the models to improve their initial responses. It was found that all models can support identifying classes and architectural styles, but they struggle with relationship recovery and the identification of design patterns. `GPT-4o` and `Mistral Large` showed the most consistent performance in identifying design patterns, and `Mistral Large` also had the best performance in identifying architectural styles.

Component summarization using hierarchical CoT prompting

In a 2023 article, Rukmono et al. [57] proposed a bottom-up approach of automated software documentation that aims to summarize software systems on a component-level by hierarchically composing findings of lower-level elements (functions) to comprehend higher-level elements (classes and components) in a software system. The research aimed to bridge the gap between the field of automated software documentation and higher-level views of a system. The proposed framework implements a *cyclic* approach, in which a CoT prompting strategy is used to iteratively summarize software elements in an ascending order of element size (bottom-up approach). First, summaries are generated for the *atomic* level of abstraction (operations). Subsequently, these summaries are aggregated for each operation within a module, and

this aggregated list is used to generate a summary of an entire module. Simplified prompt templates that were used for these two steps are given in Figure 6.

```
function summarize(operation):
    prompt ← "This is operation
        {operation.name} of module
        {operation.parent.name}:
        ‘‘‘{operation.sourceCode}‘‘‘
        Provide a summary for this operation."
    → llm.ask(prompt)
```

(a) Prompt template for summarizing *atomic* abstraction layers (operations).

```
function summarize(module):
    op_summaries: list of string
    for each operation in module:
        op_summary ← summarize(operation)
        op_summaries.add("{operation.name}:
        {op_summary}")
    prompt ← "The module {module.name} of
        component {module.parent.name} has the
        following operations:
        {op_summaries}
        Provide a summary for this module."
    → llm.ask(prompt)
```

(b) Prompt template for summarizing higher-level modules (classes).

Figure 6: Prompt templates that were used to hierarchically summarize software modules. [57]

The feasibility of the approach was evaluated on `JHotDraw`, a Java framework for two-dimensional graphics. The proposed approach in this study has several limitations. Even though component summaries were generated successfully, the content of these summaries was not evaluated. Additionally, the framework is programming language-specific because the input project's hierarchy must be known to match operations to modules. In the implementation, `javapers` was used to extract an input project's hierarchy. Thus, this implementation only works for programming languages for which hierarchy extraction packages are available.

Top-down SAR using CoT prompting

As described in Section 2.7.1, Rukmono et al. [58] proposed a framework that adopts a top-down, deductive approach for automated SAR. The authors implemented a Proof-of-Concept (POC) that follows the five steps that were described in Section 2.7.1, and uses OpenAI's `GPT-4` model in the implementation. The POC was evaluated on a subset of classes of an Android email application that was programmed in Java. A CoT prompting strategy was employed in the implementation's prompts. The implementation that was used achieved an F1-score of 69% in architecture module classification. The implementation only classifies the architectural modules to which code snippets belong, but does not generate SAD artifacts. Moreover, the implementation was not tested on enterprise-grade solutions.

Generative AI and architectural diagrams

Manjula and Dube [43] did an exploratory study in 2024 on current approaches and challenges in how generative AI could be leveraged to automatically generate architectural diagrams from textual input and code repositories. They also explored how AI techniques can be used as a supportive tool in understanding and analyzing architectural diagrams. The authors identified two categories of approaches that use generative AI in the field of architectural diagrams: a *text-to-diagram* approach and a *code-to-diagram* approach. The authors found that a lack of common standards in the domain of architectural diagramming prevents pre-trained AI models from performing consistently in the generation of architectural diagrams. It was found that generative models have problems in generating UML models because of UML's complexity and wide variety of different notations. Even though this study aimed to do an exploratory study that identifies current trends in generating architectural diagrams using AI, details were omitted, and no studies were cited to justify claims. Thus, this study should be seen as a high-level overview of the current technique landscape that does not go into depth on the advantages and disadvantages of such techniques.

Agentic automated code documentation

`DocAgent` is a multi-agent approach for generating software documentation using LLMs [75]. An agentic approach is envisioned with the aim of reducing incompleteness, inaccuracy, and lack of helpfulness in simpler LLM-based methods. The framework that is proposed in this research is composed of two

modules: a deterministic navigator module that parses a repository into an abstract syntax tree, which is subsequently used to create a dependency graph that is topologically sorted, and a documentation generation module that is composed of five agents. These agents collaborate to create software documentation based on the output of the navigator module. The framework is evaluated for *Completeness*, *Helpfulness*, and *Truthfulness*. Evaluation is done using deterministic checks, as well as by using an LLM-as-a-judge method. It was found that the documentation that is generated using the proposed framework is more complete, helpful, and accurate than documentation that is generated using more conventional LLM-based methods. `DocAgent` was only tested on Python repositories and focuses only on the static structure of repositories, ignoring the dynamic nature of real-world software systems. Moreover, the tool only generates low-level inline docstrings that do not capture the higher-level architectural structure of a software system. The usage of agents makes the framework computationally and financially more costly than more conventional methods.

Software explanation using RAG

In a 2024 Master's thesis by Pex [53], a tool called `FLASE` was introduced. This tool is a RAG-based chat application that can be used to answer software architecture-related questions for an input system. The effectiveness of `FLASE` was evaluated using three setups: (1) a setup that includes vector embeddings of the input system's source code in `FLASE`'s database, (2) a setup that includes a knowledge graph of the input system's components, classes, methods, and relationships as context, and (3) a combination of both setups.

It was found that *"for component-level questions, setups using the knowledge graph provided more comprehensive and accurate answers"* [53]. The first setup *"struggled with component-level questions due to the summarization of multiple classes, leading to a loss of detail"* [53]. In contrast, it was found that for lower-level questions the first setup *"excelled in correctness and completeness due to the access to the full relevant source code"* [53]. Furthermore, during `FLASE`'s evaluation, Pex observed that the quality of generated software explanations is subjective: *"a 'good explanation' is not universal and depends on the personality of the person"* [53].

This thesis shows that LLMs, combined with RAG extensions, can be used to extract architectural insights of input systems. However, the setups that incorporate knowledge graphs in context require the use of a deterministic knowledge graph extraction framework (e.g., `javapers`). This requirement prevents the tool from being fully *language-agnostic*, as deterministic knowledge graph extraction frameworks are *language-specific* and do not exist for all programming languages. Moreover, `FLASE` is a chat-based tool and can thus only be used to answer specific questions one at a time. The tool only generates textual software explanations, and no architectural diagrams.

Agentic C4 model creation from requirements

Another multi-agent pipeline in the field of architecture modeling was proposed by Szczepanik and Chudziak [64] in 2025. This pipeline can be used in the early-stage system design phase of the SDLC to envision a software architecture when the system has not been implemented yet. The pipeline takes *system briefs* (system description, requirements, constraints) as an input, and uses multiple agents to create C4 diagrams. These agents *"simulate a dialogue between role-specific experts who analyze requirements and generate the Context, Container, and Component views of the C4 model"* [64]. The semantic quality of the generated C4 model is evaluated using an LLM-as-a-Judge approach. The research shows that multi-agent solutions can be used to successfully create C4 diagrams, but the proposed solution can only be applied to system briefs (envisioned systems), rather than on existing systems (artifacts, source code), which makes the solution less applicable in an IT consulting context, where the architectures of already existing systems are analyzed.

2.7.3 Existing tools

Many (commercial) tools exist that aim to automate the SAD process. These tools generate different types of documentation, varying from developer and project documentation to marketing and end-user documentation. Below, we provide concise summaries of several tools and additionally mention their limitations.

PlantUML Generator

`PlantUML Generator` [67] is an open-source tool that can be used to automatically generate UML class diagrams from Java source code. It uses a deterministic algorithm to retrieve Java packages from source code and analyzes the package's classes and underlying relationships. These are then used to create PlantUML strings, which can subsequently be used to render UML class diagram images. The tool cannot be used to retrieve high-level container or component insights from input software projects, and only works for Java packages.

GitUML

`GitUML` [73] is a commercial tool that can be used to reverse engineer software projects into UML diagrams. Currently, it supports the generation of class, sequence, activity, and state diagrams. It uses a deterministic algorithm to build the diagrams, which can subsequently be rendered in HTML, Markdown, or PDF format. Additionally, the tool has AI functionality to explain source code and to explain GitHub repositories. AI is not used for hierarchical analysis and diagram generation. The tool accepts multiple programming languages as input, and the source code is all analyzed using deterministic methods.

Swimm

`Swimm` [63] is a commercial AI agent that is developed by Swimm Tech. It is a reverse engineering tool that can be used to extract dependency mappings and control flows from source code. It uses deterministic language-specific analysis tools for different programming languages that extract knowledge from source code, which is then analyzed using a deterministic language-agnostic analysis tool to create a knowledge base. This knowledge base is subsequently fed to an LLM, which uses this knowledge base to generate documentation files. Even though Swimm provides the option to generate inline comments and architecture diagrams (class, state, and sequence diagrams), the generation process is not transparent or modifiable, and the generated documentation contains incorrect information in practice [25].

Doxygen

`Doxygen` [69] is an open-source framework that generates software documentation using deterministic methods. It supports multiple programming languages, including C, Java, and Python, and can generate documentation both in HTML and in LaTeX. The tool can visualize the relationships between elements by automatically generating dependency graphs and collaboration diagrams. This generation is done deterministically without the use of LLMs. This deterministic approach causes the tool to generate technical documentation that looks similar to source code, rather than high-level, abstract documentation that can be understood by stakeholders who have a less technical background.

Javadoc

`Javadoc` [49] is a tool maintained by Oracle that is used for automatically generating the documentation for Java APIs. Javadoc is part of the standard `Java Development Kit`. The tool can automatically generate documentation that is formatted in HTML. The tool generates documentation deterministically and requires developers to use appropriate `Javadoc` syntax within their source code. `Javadoc` can be extended using *doclets* (plugins to change default documentation output). Such doclets can, for example, be used to generate UML diagrams. Limitations of `Javadoc` are its limited applicability (Java-only) and its requirement of syntax in source code.

DocAgent

As described in Section 2.7.2, `DocAgent` is a multi-agent approach for generating software documentation with LLMs. It is an open-source tool that has been developed by Meta Research. The tool first performs a deterministic, hierarchical traversal over source code and creates a topological dependency ordering, which subsequently is used to draft docstrings using an agentic approach. Such docstrings are only useful for lower-level elements in a software system, and fail to capture higher-level architectural structure. `DocAgent` is computationally expensive and only focuses on the static structure of repositories. Additionally, `DocAgent` has only been tested on Python repositories.

GitHub Copilot

`GitHub Copilot` [28] is an AI-assistant that has been developed by GitHub and OpenAI. The tool supports developers in the development program by providing features such as code auto-completion, code generation, and documentation generation. The tool is a chatbot that can be instructed using natural language. The tool can run commands in its environment, e.g., for creating and deleting files in a software repository, and additionally suggest changes. The tool is language-agnostic, meaning that it works for any programming language or even for natural language. `GitHub Copilot` can be connected to different LLMs and is fully integrated with Microsoft's tools, such as GitHub, VS Code, and Visual Studio. Even though the tool can be instructed to retrieve architectural insights and generate architectural diagrams, the tool was not specifically designed for these tasks, and should rather be seen as an all-purpose AI chatbot that has writing access to files.

Claude Code

`Claude Code` [3] is a highly capable state-of-the-art agentic AI system that has been developed by Anthropic. The tool is similar to `GitHub Copilot`, but is specialized in coding tasks and can be used to automate software development processes. Just like `GitHub Copilot`, the tool should be seen as an all-purpose AI chatbot that has writing access to files, but is more sophisticated and can act fully autonomously.

Tool comparison

An overview of the properties of the tools mentioned above can be seen in Table 2.1. *Deterministic* means that a tool does not use any stochastic method, e.g., LLMs, and thus generated documentation is fully traceable. *Documentation type* explains what type of software documentation is generated, e.g., low-level docstrings or high-level architectural diagrams. *Open-source* describes whether a tool is open-source, and which license applies to the tool. *Language-agnostic* describes whether the tool can be used on a wide variety of programming languages or standards.

Tool	Deterministic	Documentation type	Open-source	Language-agnostic
PlantUML Generator	Yes	Low-level	Yes (MIT)	No (Java)
GitUML	Partially	Low-level	No	Yes
Swimm	Partially	Low-level & architectural diagrams	No	Yes
Doxygen	Yes	Low-level	Yes (GNU)	Yes
Javadoc	Yes	Low-level & architectural diagrams	Yes (GPL)	No (Java)
DocAgent	Partially	Low-level	Yes (MIT)	Unknown[1]
GitHub Copilot	No	Any[2]	No	Yes
Claude Code	No	Any[3]	No	Yes

Table 2.1: Properties of tools that aim to automate software (architecture) documentation.

Shortcomings of existing tools and research gap

As shown above, there already exist a variety of tools that (semi-)automatically generate software documentation. Many existing tools primarily generate low-level documentation strings that do not capture architectural insights, e.g., by generating documentation for individual methods or classes. Some of the tools support the generation of low-level diagrams (e.g., UML class and activity diagrams), but they fail to retrieve high-level architectural insights from source code and do not create corresponding high-level architectural diagrams. Additionally, most tools are deterministic and only generate documentation in specific, inflexible formats. Such tools do not leverage the possibilities that LLMs bring in the domain of automatic source code analysis and automatic source code summarization. The tools that do integrate LLMs in their generation processes do not include methods that can be used such that a human remains in control. This research aims to bridge these gaps by proposing a human-in-the-loop framework that leverages LLMs to automatically generate C4 diagrams that capture the high-level architectural structure of a system, all in a programming language-agnostic manner.

[1] The performance of DocAgent was only evaluated on Python repositories.

[2] Even though GitHub Copilot can *theoretically* generate any kind of documentation, the generated documentation might not always be accurate, as the tool essentially is a stochastic chatbot.

[3] The same holds for Claude Code.

Chapter 3

Methodology

In this research, we follow the steps as described in the Design Science Research Methodology (DSRM) by Peffers et al. [51]. This methodology comprises six steps, as shown in Figure 7. In Chapters 1 and 2, we have defined the problem statement and motivated the need for a solution. This chapter discusses the remaining five steps of the DSRM framework. First, we provide the solution objectives in Section 3.1. Secondly, we describe the design and development approach in Section 3.2. Subsequently, in Section 3.3 we explain how we will demonstrate and evaluate our solution. Finally, we explain how we will communicate our solution by stating our research deliverables in Section 3.4.

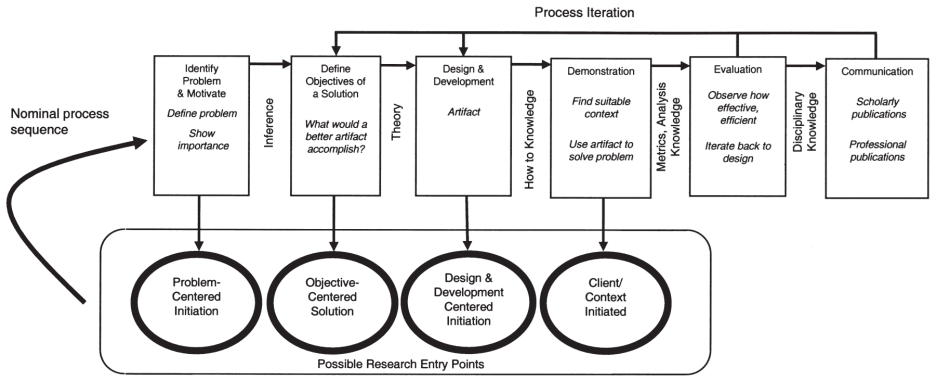


Figure 7: The DSRM framework by Peffers et al. [51].

3.1 Solution objectives

As described in our research question, this research aims to develop a framework that uses AI-driven source code analysis to create a structured and traceable C4 model representation of architectural structure, which can be used by consultants to produce reliable SAD. We will develop the framework as part of research within a firm that provides professional IT consulting services. We have conducted conversations with stakeholders within the firm to gain functional and non-functional requirements of what such a framework should encompass. We provide these requirements below. Subsequently, we will mention the projects that we will use for evaluation of the framework.

3.1.1 Requirements

The proposed framework must cover the following functional requirements:

- **FR1.** The framework must be able to take software project repositories as input.
- **FR2.** The framework must generate C4-level insights for the input project. The *container* and *component* abstraction layers are of most importance. Insights/documentation for individual code snippets is optional.

- **FR3.** Generated insights/SAD must be accurate and may not contain false or inaccurate information.
- **FR4.** If generated insights/SAD contain errors, it must be possible to correct these.
- **FR5.** The framework must be language-agnostic, meaning it works for an arbitrary programming language.

Additionally, we provide important non-functional requirements:

- **NFR1.** Project source code from confidential projects may never leave the internal network.
- **NFR2.** The pipeline must be responsive. Users should not wait for long before being able to proceed to the next steps.
- **NFR3.** The generation process must be transparent and reliable.

3.1.2 Input project selection

This research aims to create a framework that is both language-agnostic and project-agnostic. In an ideal world, the framework works for any software project that is written in any programming language. In practice, we must implement a POC and can only evaluate it on a handful of projects. We will evaluate the POC on the following projects:

Abacus

Abacus [37] is a software project that digitalizes processes in the Dutch governmental elections and is used to verify counting procedures. The software is developed by the Dutch Kiesraad, and it has been used in a number of municipalities as part of the 2026 Dutch municipal elections. The software is scheduled to be used in general elections across the entire country. We chose the project because of familiarity within the host organization, and because of its already existing C4 architecture documentation. We can use this existing documentation to evaluate SAD that will be generated by our framework. Abacus has a three-tier architecture, composed of a TypeScript and React frontend, a Rust backend, and a SQLite database. We provide a container overview diagram of Abacus in Figure 29 (Appendix B).

AI4MDE

AI4MDE [40] is an open-source research initiative at the Leiden Institute of Advanced Computer Science that integrates AI functionalities in Model-Driven Engineering (MDE). It is a software solution that end users can use to generate UML diagrams using LLM-driven pipelines. These diagrams can subsequently be used to deterministically generate software prototypes. The authors of this paper have been intensely involved in the development of AI4MDE, causing them to be familiar with the technical architecture of the software solution. AI4MDE is built using a Django API, a React-based frontend, and a Flask-based microservice that is used to generate Django prototypes. We provide a container overview diagram of AI4MDE in Figure 37 (Appendix B).

Qubus

Qubus is a software solution that is developed within KPMG NL. The solution is a platform that can be used to develop questionnaires, collect responses, and analyze insights. The source code of the solution is proprietary, which is why we will not mention its technology stack and composition.

Dogfooding our POC

Besides evaluating our POC on existing software projects, we will also evaluate our POC on its own source code (so-called *dogfooding*: using a product on itself). During development and evaluation, the source code of the POC is not open-source yet, which ensures that LLMs have not been trained on the source code and thus will not have any biases about the project integrated into their parameters. Moreover, the authors of this research are familiar with the technical architecture and details of the POC, because it was developed by themselves. This makes it easy to evaluate the diagrams that are generated for the source code of the POC. We give a full, technical description of our POC in Section 4.2.

3.2 Design and development

We will propose an artifact by iteratively developing a design that covers the functional and non-functional requirements that are specified in Section 3.1.1. This design will be the result of developing a POC that is tested against the Abacus, AI4MDE, and dogfooding cases. The design will be more abstract than the implemented POC, leaving out implementation details and only capturing the conceptual idea of the workflow. During the iterative phases of development, we align the design and corresponding implementation with stakeholders by keeping them partially involved in the development process.

3.3 Evaluation methodology

When the development phase has ended, and the proposed artifact has been finalized, we demonstrate and evaluate the artifact on all projects that are mentioned in Section 3.1.2. We evaluate our POC using metrics for Lindland's three categories of quality [41]. We evaluate the syntactic and semantic quality of the output by our POC by analyzing generated diagrams of different abstraction levels individually, such that we can investigate how our POC performs in different abstraction layers (high level versus low level). We evaluate pragmatic quality of the entire workflow by conducting qualitative stakeholder interviews. Additionally, we compare the syntactic and semantic quality of our POC against a baseline, and we evaluate the consistency of the generated diagrams.

Syntactic quality

This category concerns the quality of the textual structure of output (e.g., container names, relationship labels) without taking into consideration the semantic meaning of the output. In the case of a C4 model output, this type of quality refers to whether correct notation is used. We evaluate the syntactic quality of generated diagrams by subjectively assessing whether the elements in the generated diagrams adhere to pre-defined properties. We define the following syntactic quality metric:

$$ErrorRate = \frac{n_{\text{error}}}{n_{\text{total}}} \times 100\%$$

Where n_{total} is the total number of elements to be evaluated in a generated diagram. For container diagrams, these consist of (1) container names, (2) container descriptions, (3) container technologies, and (4) relationship labels. For component diagrams, these consist of (1) component names, (2) component descriptions, and (3) relationship labels.

n_{error} is the number of elements in a generated diagram that are syntactically incorrect. We will evaluate container diagrams and component diagrams individually, by determining for all elements in a diagram whether they are *well-formed*, *concise*, and *descriptive*. If an element misses one of these properties, it is marked as syntactically incorrect. The properties are defined as follows:

- **Well-formed:** The element is constructed using correct grammatical rules, e.g. *"responsible for data retrieval and aggregation"* instead of *"responsible data aggregation retrieval"*.
- **Concise:** The element is brief and comprehensive, e.g. *"backend"* instead of *"the backend service container"*.
- **Descriptive:** The element describes properties, and does this in an objective manner, e.g. *"implements Dijkstra's algorithm"* instead of *"a component for something"*.

Moreover, if a generated diagram cannot be rendered because of syntax errors (e.g., because of incorrect PlantUML syntax), the number of syntax errors is added to n_{error}.

Semantic quality

The semantic quality of a model refers to the quality of the deeper *meaning* of its elements. In the case of a C4 container diagram and C4 component diagram output, this would mean that containers and components are correctly defined and segregated, and that their relationships are actual dependencies that can be found in the source code. Semantic quality of an element can be assessed by comparing it against a ground truth, or by comparing it against the opinion of an expert who has knowledge about a system's technical architecture.

Classical deep learning metrics such as *recall*, *precision*, and *F1-scores* are difficult to apply for evaluating the semantic quality of generated C4 diagrams because there is no strict ground truth in architectural modeling. The same architecture can be modeled using various diagrams that differ in focus, audience, technical nuances, or subjective opinions by the modeler. These differences can be explained by the notion of different *architectural viewpoints*, as elaborated upon in Section 2.1.1. Because of these reasons, we opt for an evaluation method that takes into account the subjective opinion of different project stakeholders.

We send out surveys to project stakeholders to evaluate the semantic quality of generated diagrams. In these surveys, stakeholders are provided with the generated diagrams and corresponding reference diagrams. The stakeholders must evaluate the semantic meaning of generated elements by classifying them according to descriptions given in Tables 3.1 and 3.2. These descriptions are based on a scoring system that was introduced by Tseng [65], but has been altered for our research. Additionally, we add the **Missing** type to also cover false negatives. We provide reference diagrams as reference material (Appendix B) that the stakeholders can use to come up with a judgment, but the final judgment is based on the subjective opinion of stakeholders, rather than explicitly comparing generated diagrams against the reference diagrams. By giving this mandate to project stakeholders, technical nuances and subjective architecture representation decisions can be taken into account in the semantic evaluation, rather than blindly comparing generated elements against a strict ground truth.

We assign stakeholders with the following tasks:

1. Classify the containers in the generated container diagram according to the descriptions in Table 3.1, and additionally supply a list of missing containers.
2. Classify the container–container relationships in the generated container diagram according to the descriptions in Table 3.2, and additionally supply a list of missing container–container relationships.
3. For each generated component diagram, classify the components in the diagram according to the descriptions in Table 3.1, and additionally supply a list of missing components.
4. For each generated component diagram, classify the component–component relationships in the diagram according to the descriptions in Table 3.2, and additionally supply a list of missing component–component relationships.
5. For each generated diagram (container and component), give a small textual motivation of why the generated diagram is acceptable/not acceptable.

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

Table 3.1: Scoring rubric for containers in generated container diagrams and components in generated component diagrams.

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

Table 3.2: Scoring rubric for relationships in generated container diagrams and generated component diagrams.

For each survey response, we can count the number of elements for an element type individually (containers, components, container–container relationships, and component–component relationships):

- n_m = number of matching elements (of one element type)
- n_v = number of valid extra elements (of one element type)
- n_i = number of irrelevant elements (of one element type)
- n_h = number of hallucinated elements (of one element type)
- n_{mi} = number of missing elements (of one element type)

When the elements have been counted, we can calculate a raw score for an element type by applying the following formula:

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

In this formula, we reward for matching and valid generated elements, and penalize for irrelevant and hallucinated elements, and we additionally penalize for missing elements. This formula is also based on the scoring system used by Tseng [65], but has been extended by including false negatives (missing elements).

After we calculate the raw score for an element type, we normalize over the best and worst possible scores for the generated diagram. We apply linear min-max normalization to transform the scores into a bounded range of $[0, 1]$. By doing this, we normalize the scores because different systems can have different diagram sizes, and thus, raw scores can be higher or smaller overall for different diagrams. Moreover, container diagrams are usually smaller than component diagrams. We can apply the following formula to raw scores to normalize them:

$$S_{\text{normalized}} = \frac{S_{\text{raw}} - MinScore}{MaxScore - MinScore}$$

where:

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

The $MaxScore$ and the $MinScore$ are based on what the best and worst version of the current generated diagram could be, instead of what the best or worst ground-truth diagram would be (a ground-truth diagram does not exist). Because of this, the $MaxScore$ and $MinScore$ can differ for different diagrams of the same input project. We give a reward of 2 for the elements in the $MaxScore$ because these elements would be classified as matches in the 'perfect' diagram. We include missings in the $MaxScore$, because the false negatives that were missed in the generated diagram were expected in the 'perfect' version of that diagram (the false negatives would be classified as matches). We exclude hallucinated elements in the $MaxScore$, because these would be omitted in the "perfect" version of that diagram and thus should

not be weighted as possible matches. We include all types in the $MinScore$, because this is the sum of potential hallucinated and missing elements in the generated diagram. In this "worst" possible diagram, all elements that are present in the generated diagram would be classified as a hallucination or missing (penalize a value of 2).

Subsequently, we can use this normalized score to determine whether the generated elements for a specified element type are close to what elements would be expected:

$S_{\text{normalized}} = 1$	Perfect match: all expected elements are present and none are missing,
$S_{\text{normalized}} = 0.5$	Partial match: some expected elements present, some missing or ambiguous,
$S_{\text{normalized}} = 0$	No match: none of the elements are expected.

Additionally, we calculate the hallucination rate for different element types. This rate gives an indication of the accuracy of our generated diagrams. For individual element types, we use the following formula:

$$hallu\% = \frac{n_h}{n_m + n_v + n_i + n_h}$$

We approximate the **acceptability** of a generated diagram by comparing its semantic quality scores and hallucination rates against heuristic thresholds. These heuristic thresholds are indicative guidelines for acceptability, and have been set to $S_{\text{normalized}} \geq 0.75$ and $hallu\% \leq 10\%$ for all element types. We have set these heuristic thresholds in collaboration with respondents, and they should not be seen as a strict boundary. We have set the thresholds by asking semantic evaluation respondents whether they believe a generated diagram is acceptable. We provide these responses (*acceptable / not acceptable / partially acceptable / debatable whether acceptable*) in the semantic evaluation results. By analyzing the responses and their corresponding calculated $S_{\text{normalized}}$ and $hallu\%$, we can reverse-engineer the thresholds to the given values. We can conduct this reverse-engineering by aligning the calculated metrics with corresponding stakeholder judgments and then identifying the boundary region where the label flips. As will be seen in the evaluation results, this boundary region is not very strict, and given a limited number of evaluation respondents, no sophisticated statistical method can be used to reverse-engineer the boundary. Due to this, we choose conservative thresholds, based on manual analysis, of 0.75 and 10%, for which most stakeholders describe diagrams as acceptable.

Comparison with baseline

As described in Section 2.7.3, a variety of tools exist that generate SAD or architecture diagrams for input projects. Of these tools, only `GitHub Copilot` and `Claude Code` can generate C4 diagrams based on input source code. To benchmark the effectiveness of our POC, we compare the syntactic and semantic quality of its output against that of the output of `GitHub Copilot`. We evaluate the output of `GitHub Copilot` ourselves. We use the reference diagrams in Appendix B as a reference guideline during evaluation. We instruct `GitHub Copilot` with the following prompt for input projects:

Prompt:

> "Based on the source code of this software project, generate a C4 container diagram and corresponding C4 component diagrams (a component diagram for every container). Include containers, components, container-to-container relationships, and component-to-component relationships in these diagrams. Use C4-PlantUML (PlantUML standard library) for generating diagrams. Do **not** ask any further questions, start generating immediately based on the source code of the project."

We have chosen to use a simple zero-shot prompt as a baseline, such that we can compare the performance of our solution against the bare performance of `GitHub Copilot`, without including additional logic and agents that we implement in our solution. We use this method for C4Gen, AI4MDE, and Abacus as input projects. We cannot evaluate `GitHub Copilot` on the Qubus project because of confidentiality constraints. We select `gpt-4o` as the LLM used by `GitHub Copilot`, because this is the same LLM that we will use in our POC. We use `GitHub Copilot` agent mode and give the agent read access to the input project's source code. We only provide the given prompt; no other additional instructions. If `GitHub Copilot` successfully generates C4 diagrams (PlantUML strings), we evaluate the syntactic and semantic

quality of these diagrams using the same metrics as described above. We can use these evaluations as a baseline to compare our POC's evaluations against. The baseline scores show how a general-purpose LLM-driven tool can generate C4 diagrams without any specialized workflow or human validation, which establishes a lower-bound reference for our solution.

Comparison with top-line

Besides comparing the proposed solution against a baseline, we introduce a top-line comparison such that we can position the performance of our solution relative to a state-of-the-art, highly capable agentic AI system. Whereas we used `GitHub Copilot` to provide a lower-bound reference, we use `Claude Code` to provide an upper-bound.

We select `Claude Code` because of its capabilities in autonomous reasoning and iterative refinements. Similar to the baseline comparison, we generate C4 diagrams and evaluate their syntactic and semantic qualities by using the reference diagrams in Appendix B as a guideline and applying the previously defined quality metrics. Unlike the baseline setup, which uses a simple zero-shot prompt, the top-line prompt explicitly instructs `Claude Code` to use the RCI-mechanism that we explained in Section 2.5. We provide the full prompt in Appendix A.9. In this prompt, we first state the objective for the agent (generating C4 diagrams), after which we enumerate the requirements and rules for the generation process. Subsequently, we state the stop condition for the RCI-mechanism. We use `Claude Code` with the `Opus 4.7` model (`Thinking` mode on), which enables extensive reasoning capabilities. To be able to fairly compare the syntactic and semantic quality of the models that are generated in this top-line study against the models generated by our POC, we only evaluate the syntax and semantics of containers, container–container relationships, components, and component–component relationships.

Output consistency

The design that we envision in Chapter 4 encompasses a sequential agentic pipeline, which introduces the stochastic ensemble effect that we described in Section 2.6. Because of this stochasticity, we must evaluate the consistency of the output of our envisioned design and corresponding POC. We evaluate this consistency using the following methodology. The goal of this evaluation is not to perform a full statistical study of model variance, but rather to demonstrate whether the POC behaves consistently in a practical use case.

We select the POC dogfooding case as described in Section 3.1 as an input project to evaluate consistency for. We choose this input project because of architectural familiarity and independence of external stakeholders. We generate container diagrams and corresponding component diagrams for the source code of the POC for a total of $k = 3$ times. Each run produces one container diagram and two corresponding component diagrams. We evaluate the consistency of these diagrams individually using the rubric in Table 3.3. Across runs, we use the same LLM temperature and POC hyperparameters.

Rating	Description
Full consistency	"In all runs, the generated diagrams contain *functionally identical* architectural information. The architecture of the system is decomposed using *identical* containers/components/relationships. There are only *trivial linguistic variations* (e.g. different phrasings or synonyms)."
High consistency	"In all runs, the generated diagrams contain *functionally identical* architectural information, with only *minor trivial variations* (e.g., optional detailed elements added/omitted, or trivial information is aggregated/separated). The architecture of the system is decomposed using *interchangeable* containers/components/relationships."
Moderate consistency	"The generated diagrams are not *functionally identical*, but the *architectural viewpoint* is preserved across all runs. Functional differences are present in *non-critical* parts of the system's architecture. These variations do not affect the overall *understanding* of the system's architecture."
Low consistency	"There are differences in the *architectural viewpoints* of the generated diagrams across runs, also in *critical* parts of the system's architecture. Separate runs do not agree on a common architecture."
High inconsistency	"The generated diagrams do not capture the same *architectural viewpoint* at all across runs. *Critical* containers/components/relationships are added or removed, creating completely different architectural interpretations."

Table 3.3: Rating rubric for consistency of generated diagrams.

Pragmatic quality

We define the pragmatic quality of our solution as *"the subjective quality of the solution when using it in an IT consulting context, as experienced by IT consultants"*. We assess this subjective quality qualitatively by conducting feedback sessions with stakeholders who have an IT consulting profession. These stakeholders will give qualitative feedback on the output that is generated by our POC, as well as feedback on the user experience and applicability of the solution.

To be able to receive this qualitative feedback, we organize sessions with IT consultants and let them experiment with the POC. Afterwards, we send a survey with questions with the aim of collecting qualitative feedback. We provide a list of the professional roles of our participants in Table 5.16. We ask questions that are inspired by the Technology Acceptance Model (TAM) [16], which is a model that can be used to evaluate the pragmatic quality of a software solution. According to this model, the *Behavioral Intention* (BI) of whether people will be using a technology is influenced by their *Attitude* (A) towards the technology. This attitude is influenced by the *Perceived Usefulness* (PU) and *Perceived Ease of Use* (PEU) of the technology. We opt to use open-ended questions, rather than a Likert scale, which is used in conventional TAM questionnaires, with the aim of gathering detailed textual feedback from a few key stakeholders, rather than retrieving numerical insights from many stakeholders.

We present the following questions in the survey, which cover the different dimensions of the TAM:

- **Q1.** "What is your overall impression of the POC? (A)"
- **Q2.** "Do you believe that the generated diagrams cover the input system's architecture?" (PU)
- **Q3.** "Could you foresee yourself using the POC in any of your operations within your organization?" (PU)
- **Q4.** "Do you think the POC would improve the productivity in your operations?" (PU)
- **Q5.** "Do you think you can be more effective by using the POC?" (PU)
- **Q6.** "What functionalities do you think that the POC is lacking?" (PU)
- **Q7.** "Do you think the POC is easy to use?" (PEU)

- **Q8.** "Do you have any suggestions for improvements that would make the solution easier to use?" (PEU)
- **Q9.** "Do you intend on using this solution in your operations?" (BI)

Evaluation responsibilities

To reduce ambiguity about who evaluates what, we distinguish between author-led and stakeholder-led evaluations. The authors conduct baseline and top-line evaluations using the defined syntactic and semantic quality metrics to provide reference bounds to compare to. The authors also evaluate C4Gen's syntactic quality. Both external stakeholders and the authors evaluate C4Gen's semantic quality. IT consultants evaluate C4Gen's pragmatic quality.

Only the authors evaluate syntactic quality because this type of quality only concerns syntactic errors. C4Gen's semantic quality also includes external evaluation such that architectural correctness can be assessed under different viewpoints. IT consultants conduct C4Gen's pragmatic evaluation, such that the applicability of the tool in an IT consulting context can be assessed. Ideally, the baseline and top-line evaluations would also be done by external stakeholders, but they are conducted by the authors to unburden stakeholders and because the results are only used as reference points.

3.4 Deliverables

We communicate our solution using three deliverables:

- **Conceptual design.** We create a conceptual design that can be used to address the problem statement. This design is a high-level overview of the solution that can be used to create a POC. This conceptual design refrains from including implementation details, such as which programming languages were used, which LLM was chosen, and which dependencies are needed. We discuss the proposed conceptual design in Section 4.1. The purpose of this high-level overview is to communicate the solution in an accessible manner, without going into implementation details. Moreover, the conceptual design can be used to implement similar solutions that use different technologies.
- **Proof-of-concept.** The conceptual design is used to implement a POC. This POC is the deliverable that we evaluate. The POC is the actual source code of an implementation of the conceptual design, and can be used to validate the proposed solution on real projects. We address the proposed POC in Section 4.2, where we also state which technologies were used for implementation.
- **List of limitations.** After evaluating the solution on real software projects, we provide a list of limitations of the conceptual design and its corresponding POC. We provide and discuss this list in Section 6.2.

Chapter 4

Design

This chapter discusses our proposed solution to the problem statement. First, we provide the high-level conceptual design of the solution in Section 4.1, and additionally motivate some crucial design decisions that have been made. Subsequently, in Section 4.2, we dive into implementation details of C4Gen, which is a POC that implements the proposed conceptual design. Finally, we provide a demonstration example in Section 4.3, which illustrates how C4Gen works in practice.

4.1 Conceptual design

Our conceptual design can be divided into three global phases. First, a top-down phase of identifying containers, container relationships, and components, and then a bottom-up phase of generating component summaries and container summaries, and identifying component relationships based on component summaries. The third phase encompasses the deterministic rendering of C4 diagrams and SAD creation. The first phase is inspired by the top-down approach by Rukmono et al. [58] in 2024. The second phase is inspired by the 2023 bottom-up summarization technique by Rukmono et al. [57]. Figure 8 shows a simplified diagram that shows the different phases. Figure 9 shows a more detailed UML activity diagram of the full conceptual design that we propose as a solution to the problem statement. It is a sequential pipeline that integrates the AI agents with deterministic algorithms. As an input, it takes a software repository, and after following the sequential steps, it generates an SAD document that contains C4-level container and component diagrams.

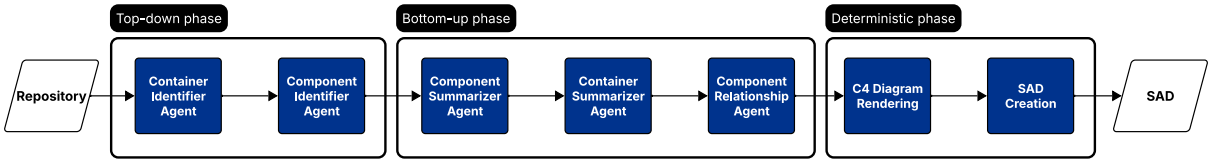


Figure 8: Simplified agent workflow diagram for the conceptual design.

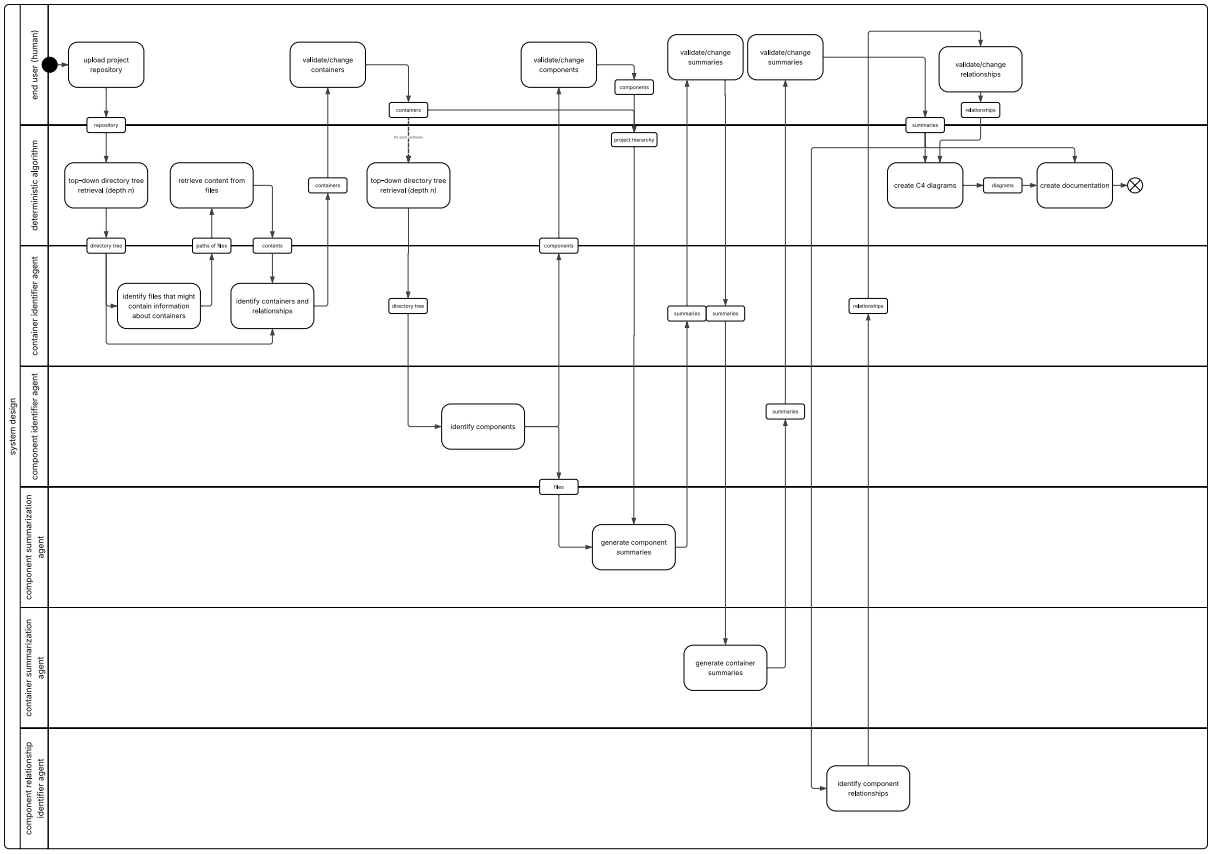


Figure 9: UML activity diagram for the conceptual design.

4.1.1 Individual steps in the design

The following is a global overview of the individual steps in the sequential pipeline. The first two steps encompass the top-down container and component identification stage. The next three steps encompass the bottom-up summarization stage. In the final two steps, C4 diagrams and SAD are created deterministically.

Phase 1: Top-down identification

Step 1: Container identification agent (including container relationship identification)

When the user uploads a project repository (directory with source code), a deterministic algorithm similar to the Unix `tree` command retrieves the top n (hyperparameter set in a configuration file) layers of its directory tree. This algorithm truncates every directory at a layer that is deeper than layer n in the root directory. Figure 10 contains an example of an output of this algorithm. The output of this algorithm is then truncated to a prompt, which is used by the first agent. This agent aims to identify files that might contain information about the project's containers/services, e.g., Docker Compose files, Kubernetes manifests, or service definition files. The agent returns the paths of such files, and the content is then retrieved deterministically. A second agent then uses these file contents, together with the previously retrieved directory tree, to identify C4-level containers and their underlying relationships. Finally, these containers and relationships are presented to the user, who can adjust or delete them, or manually create new ones. A container instance has the following properties: (1) `id`, (2) `name`, (3) `type` (e.g., frontend, backend, database, microservice, etc.), (4) `description`, (5) `root_paths` (paths to the root directories of the container), and (6) `technologies` (a list of used programming languages/frameworks). A relationship instance has the following properties: (1) `id`, (2) `source_container`, (3) `target_container`, (4) `label`, and (5) `bidirectional` (a boolean bidirectional flag; one arrowhead versus two arrowheads in a C4 diagram).

```
c4-gen [directory]
|-- LICENSE.md
|-- README.md
|-- plantuml-server [directory]
|   `-- docker-compose.yml
|-- pyproject.toml
|-- requirements.txt
`-- src [directory]
    `-- backend [directory]
        |-- README.md
        |-- api.py
        `-- c4_diagram_builder [directory]
            |-- c4_diagram_builder.py
            `-- utils[directory][truncated]
```

Figure 10: Example output of the deterministic `tree` algorithm ($n = 3$). Every directory deeper than n is truncated. We explicitly add the `[directory]` tag to directories. We explicitly add the `[truncated]` tag to truncated directories.

Step 2: Component identification agent (excluding component relationship identification)

When the C4-level containers have been identified automatically and are validated by the user, the C4-level components can be identified for individual containers. The first step of this is to run the deterministic *tree* algorithm again (with a new value for n, set in the configuration file) for all the root directories of a container. These root directories have been identified in the previous step. For every container, a third agent builds a prompt by aggregating the container's directory trees and providing the container information that has been identified and validated in the container identification phase. The agent sends this prompt to an LLM, which returns a list of identified components. The user then validates these components. A component has the following properties: (1) `id`, (2) `name`, (3) `description`, and (4) `file_paths` (a list of paths to files that the component is made of). During this phase, the underlying relationships between components are not identified yet.

Phase 2: Bottom-up summarization and component relationships

When the previous step has finished, all containers and components are identified automatically, and these identifications have been validated by the user. The next step is to generate summaries that encompass the functionalities/roles of these identified objects. We do this in a bottom-up way by first summarizing individual components, and then using the summaries of components to generate summaries of the parent containers, similar to the approach by Rukmono et al. [57] as described in Section 2.7.2.

Step 3: Component summarization agent

For each individual component, we retrieve the contents of its files deterministically. We then aggregate this content and use it to build a prompt which the agent sends to an LLM. This prompt also includes the component's properties that have been identified in the previous step, as well as the properties and directory trees of its parent container. The LLM returns a summary of the component. This summary is a textual string of 200-400 words, which the user can validate/change. Figure 11 shows an example of a generated summary.

```
**Purpose:**
The C4 Diagram Builder component is responsible for constructing C4 diagrams, which are
    used to visualize the architecture of software systems. This component plays a crucial
    role in the backend service by enabling the generation of visual representations of
    system components and their interactions, aiding in the understanding and documentation
     of the system's architecture.

**Key functionalities:**
The C4 Diagram Builder provides functionalities to create and manage C4 diagram items and
    relationships. It allows for the definition of diagram elements such as containers,
    components, and their relationships, using attributes like name, technology, and
    description. The component can generate PlantUML text representations of these elements
    , facilitating the creation of diagrams that can be rendered using PlantUML tools. This
     capability is essential for producing clear and standardized architectural diagrams
    that can be easily shared and understood by stakeholders.

**Relationships with other components:
The C4 Diagram Builder operates within the backend container and interacts with various
    components to fulfill its role. It is likely to be utilized by the Diagrams Route
    component, which handles API requests related to diagram operations, to generate and
    serve C4 diagrams as part of the backend's API offerings. Additionally, it may interact
     with the Documentation Generator component to incorporate generated diagrams into
    system documentation. However, specific interactions with other components are not
    explicitly detailed in the provided information.
```

Figure 11: Example of a generated summary for a software component, generated using `gpt-4o`.

Step 4: Container summarization agent

When the generated component summaries have been validated by the user, we subsequently use these summaries to generate summaries of the parent containers (*summary-of-summaries* approach, similar to the approach by Rukmono et al. [57] as described in Section 2.7.2). We do this for each individual container by aggregating the child components' summaries and building a prompt that includes these summaries, together with the container's properties and the directory tree of the container. The agent then sends the prompt to an LLM, which returns a container summary. This summary is a textual string of 200-400 words, which the user can validate/change.

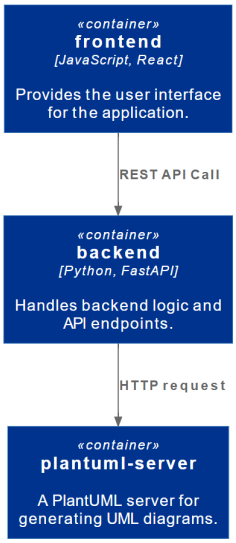
Step 5: Component relationship agent

Relationships between underlying containers have already been identified during the container identification step, but relationships between underlying components have not been identified yet. We do this identification during this step, after containers and components have been summarized. The generated summaries include information about relationships between components, which is based on the file contents of components. We add these summaries to a prompt and send the prompt to an LLM. The LLM proposes relationships, which the user can validate/change.

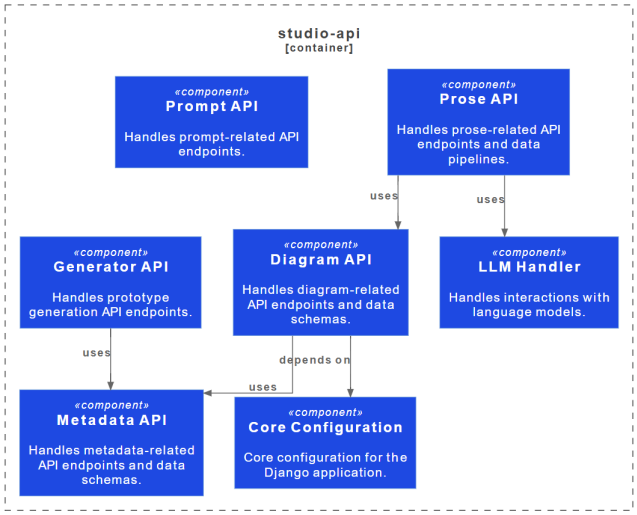
Phase 3: Deterministic SAD creation

Step 6: Deterministic diagram generation

When all containers, components, and relationships have been identified and validated, including all their corresponding properties, we can create a C4 container diagram that shows the container-level overview of the software project deterministically. Additionally, we can create a C4 component overview diagram for each individual container. We can do this by deterministically parsing the analysis results to a diagramming format that supports the rendering of C4-level diagrams, e.g., C4-PlantUML [54]. Figure 12a shows an example of such a rendered container overview diagram. Figure 12b shows an example of a rendered component overview diagram for an individual container.



(a) C4 container overview diagram for a software system (C4Gen).



(b) C4 component overview diagram for an individual container (AI4MDE API).

Figure 12: Examples of rendered C4 diagrams, built using generated analysis results.

Step 7: Deterministic SAD creation

Finally, we generate an SAD markdown document that captures the insights and diagrams that are retrieved and generated in all previous steps. This markdown file first presents the project's container overview using the generated C4 container overview diagram. Under this diagram, we present the properties and summary of each individual container. Subsequently, for each individual container, we present the component overview using the generated C4 component diagrams. We explain each component using the generated summaries.

4.1.2 Design decisions

Our proposed design is composed of multiple sequential steps, all of which can be implemented using different approaches. This section aims to motivate a selection of design decisions that have been made in our proposed solution.

Agentic approach

One of the first steps of implementing an agentic solution is deciding whether to use AI agents at all. Alternative solutions to the problem statement are:

1. Sending the entire input repository to an LLM (non-agentic) and requesting C4 diagrams.
2. Iteratively chatting with an LLM-driven chat interface, with the goal of iteratively providing information and building C4 diagrams.

The first option is not feasible because software repositories can grow large and complex. State-of-the-art LLMs have an effective context length of at most 128K tokens [29], which corresponds to roughly half a million characters (note that the *effective* context length is smaller than the marketed context length; LLMs start to lose attention beyond the effective context length). This is far too small for enterprise-grade software solutions. Additionally, this option is not traceable because of its black-box nature of providing an input repository and immediately receiving a C4 diagram without any intermediate steps. Because of this reason, users have little control over the decisions that are taken. As described in Section 3.3, we use this alternative approach as a baseline to compare the performance of our POC against.

The second option is not feasible because it lacks a standardized approach of iteratively taking intermediate decisions. Standardized prompts can be created, but end users are still required to actually use these prompts. Additionally, a chat interface can only take textual inputs and provide textual outputs, which can harm user experience and increase end-to-end runtime.

Agentic solutions can touch the middle ground of both solutions. A task can be subdivided into smaller sub-tasks, increasing maintainability and traceability, and opening up more possibilities of user control. Agents also handle all prompting logic, ensuring that end users do not have to use LLM-based chat interfaces or manually build prompts, but rather navigate through an intuitive user interface that is connected to the agents. Because of these advantages, we have chosen to implement an agentic solution.

Agentic approach of identifying containers

In our solution, the container identifier agent identifies paths of files that may specify information on what containers a system is composed of. We have opted for this strategy instead of using deterministic methods, like regex-based matching, because the agentic approach enables the detection of files in a context-aware and semantically flexible manner, instead of relying on standard naming conventions or predefined patterns. Deterministic methods like regex-based matching may fail in repositories where container-related information is not stored using common naming conventions or directory structures. By leveraging an LLM, we aim to identify files in a repository-agnostic manner.

Used agents

One of the critical steps of implementing an agentic solution is deciding what agents to implement where in the solution. As described in Section 4.1, our solution is composed of seven steps, and uses five agents (container identifier including container–container relationship identification, component identifier, component summarizer, container summarizer, component relationship identifier). We have chosen not to use agents for the last two steps (C4 diagram rendering and SAD creation) because these steps can be done fully deterministically. By not implementing agents for these steps, we save on token usage and runtime and avoid the risk of hallucinations or stochastic errors. We have chosen to split the rest of the pipeline into five agents for the following reasons:

1. Because of the parent–child relationship between containers and components, containers must first be identified and validated before their child components can be identified.
2. Container–container relationships are directly identified by the container identification agent because of the abstract, high-level nature of container–container relationships.
3. The component identifier agent does not identify component–component relationships because we noticed that these detailed relationships cannot be identified by LLMs without including the component's source code in the prompt. This agent solely identifies the file paths of components based on the directory tree, without reading the source code. Because of this, we have chosen to identify component–component relationships in a later step, after the component's source code has been retrieved.
4. Component summaries can only be given after components have been identified. This is why we implemented a component summarization agent that is called after the component identification agent has finished.
5. Container summaries are based on component summaries, which is why they can only be generated after component summaries have been generated and validated.
6. Component–component relationships are based on the source code of components, which is read by the component summarizer agent. Additionally, the parent container's summary is used as an input, which is why the component–component relationship identification agent can also only be called after container summaries have been generated and validated. This is why we have chosen to implement this step as a final, separate agent.

Tree algorithm and truncating directory trees

We implemented a deterministic algorithm that reads the directory tree of a given input directory. This algorithm is used for adding context to multiple prompts, which are subsequently used by the agents in our pipeline. We have chosen to implement such an algorithm ourselves in Python, rather than using the Unix `tree` command, because the Unix `tree` command does not allow for explicitly adding a `[directory]` and `[truncated]` suffix. Because these trees are sent as context to LLMs, it must be explicitly stated which entries are files and which are directories, because the LLM only takes textual strings as input. We have chosen to truncate directories after depth n because we noticed that for

multiple state-of-the-art LLMs the performance of our pipeline dropped if we included entire directory trees in a prompt. Additionally, including the entire directory tree drastically increased token usage, and thus computational runtime, network traffic, and costs.

Not including file contents in component identification phase

Figure 9 shows that, in order to identify containers, the content of potentially important files is given as an input. This cannot be said for the identification of components. Initially, we tried to also include the content of potentially important files in order to identify components, but the performance of our pipeline drastically worsened. We assume that this is because of too high context when including the content of low-level implementation files, which causes the LLMs to lose track of important input tokens. Due to this, we have chosen not to include file contents in this step.

Identification of component relationships after summarization steps

In a C4 container overview diagram, relationships between containers are often generally abstract and broadly defined (e.g., REST API calls, HTTP requests). We noticed that individual containers do not require full implementation details of other containers in order to retrieve these relationships. However, in C4 component diagrams, this is not the case. These relationships are often more detailed and sophisticated, and can have specific labels that can only be defined after explicitly reading source code. Due to this, we first need the details of all components in a software system before we can extract these relationships. Due to this, we have chosen to first let the summarization agent generate the summaries for all components in all containers, after which this new agent can extract the relationships between components by taking all summaries as input.

Human validation steps

Essential decisions in the process of designing a human-in-the-loop workflow are the decisions of where to integrate human interventions in the workflow. We have chosen to do this after every agent call, to ensure that judgments that are made by LLMs must always be validated before the next step in the sequential pipeline can be started. By doing this, we mitigate the risks of amplifying false information in ensemble LLM calls (the concept described in Section 2.6). We have chosen not to let the user have any control over the output of deterministic steps to prevent a decrease in the ease of use of the pipeline. However, some deterministic steps can be steered slightly by setting hyperparameters, e.g., by setting the `max_depth` parameter in the deterministic tree algorithm.

AI transparency

In Section 2.2 we mentioned the importance of trust, documentation accuracy, and AI transparency in an IT consulting context. In Section 2.6, we discussed three dimensions of AI transparency; *explainability*, *traceability* and *communication*. In our POC, which we will discuss in Section 4.2, we have decided to implement the following measures to comply with these dimensions:

1. The user can read the motivation that is provided by the LLM during the identification steps. This increases explainability of the agents' outputs.
2. During the container identification step, the user can see which source code files are read and aggregated to the agent's prompt. This increases explainability and traceability of the agent's output.
3. During the component identification step, the user can see which file paths are identified by the agent. This increases the traceability of the component summarization agent.
4. All functionality that incorporates stochastic decisions made by AI agents, explicitly mention the word *generate*, implying that LLMs are used. This increases transparency in the communication dimension.
5. The documentation of our POC explicitly states in which steps AI is used. This increases transparency in the communication dimension.
6. During all steps in the solution, judgments that are made by LLMs must be validated and can be changed or reverted. This ensures that a human is always responsible for the final output of the solution.

4.2 Proof-of-Concept: C4Gen

To be able to evaluate our proposed design, we implement the design in a POC, which is called C4Gen. C4Gen is composed of a frontend, backend, and a localized version of a PlantUML server. Figure 12a shows a C4 container overview diagram of the POC.

Frontend

The frontend is the entry point for users and provides a user interface for the POC. On the landing page, the user can upload a software repository (**.tar.gz**, **.tar**, or **.zip** file) that is to be analyzed. When the user uploads a file, the sequential pipeline is triggered in the backend. After every agentic step in the backend, a page is shown to the user in which the results of the agentic step can be seen and edited. Section 4.3 contains screenshots of these pages. On these pages, the user can edit container and component properties, as well as their corresponding summaries and relationships. In the final step, the user can download the generated SAD in markdown format. The frontend is implemented using JavaScript and React because of familiarity. A C4 component overview diagram of the frontend can be seen in Figure 25 (Appendix B).

Backend

The backend of the POC is responsible for all the retrieval, parsing, and generation logic that is described in our design. The backend is written in Python and uses FastAPI to expose a variety of endpoints related to (among others) container and component identification, relationship identification, summarization, directory tree retrieval, prompt building, and LLM handling. C4Gen can call OpenAI's models directly, but we also added functionality to call KPMG's `Workbench` API. `Workbench` is an API that can be used to securely call Azure deployments of LLMs, preventing the leakage of confidential data. By using `Workbench`, we can still use the POC for confidential software repositories. C4Gen's API endpoints are called by the frontend, in which the user can subsequently alter generated data. The backend is connected to a local PlantUML server to be able to return rendered C4 diagrams (**.svg** format). Figure 21 (Appendix B) shows a C4 component overview diagram of the backend.

Prompts used in backend

Appendix A contains the prompts that are used by the agents in the backend. These prompts are the result of iterative development, and include multiple techniques such as emotional prompting, one-shot prompting, request structuring using Extensible Markup Language (XML) tags, and response formatting (JSON and XML tags). We have implemented a total of eight prompts, split over the five different agents:

Container identification agent:

- **Container information file path identification.** This prompt aims to identify the paths of files within the input repository that are likely to contain information on what containers the repository is composed of. It takes the truncated directory tree of the repository as an input between XML tags, and expects a list of file paths as a response. We use emotional prompting techniques (e.g., *'please'*, capitalization, and exclamation marks), as well as one-shot prompting (we give an example response). Additionally, we use a CoT technique to steer the LLM to first come up with a motivation on why particular file paths might be interesting. The LLM returns this motivation between XML tags, such that it can be retrieved for traceability.
- **Container identification.** This prompt aims to identify containers and their corresponding properties. As inputs, it takes the truncated directory tree of the project, together with the contents of the files that have been identified by the previous prompt. This prompt expects a JSON string as a response that contains information on the identified containers. Just like in the previous prompt, we use emotional prompting techniques, as well as one-shot prompting and XML formatting.
- **Container relationship file path identification.** This third prompt aims to identify the paths of files that might contain information about the underlying relationships between containers. It takes the identified containers as input, as well as the project's truncated directory tree. As a

response, we expect a list of file paths between XML tags. Again, we use emotional prompting and one-shot prompting.

- **Container relationship identification.** This prompt aims to identify the relationships between containers. As an input, it takes the contents of the files that have been identified by the previous prompts, as well as the project's truncated directory tree and the properties of the identified containers. As a response, we expect a JSON string that contains information on the identified relationships. Just like in the previous prompts, we use emotional prompting and one-shot prompting techniques. Also here, we use a CoT technique for reasoning and motivation retrieval.

Component identification agent:

- **Component identification.** The purpose of this prompt is to identify the components corresponding to an individual container. We give the parent container's properties as input, together with the truncated directory trees of that individual container. We expect a JSON string that contains information on the components as an output. We use one-shot prompting, as well as CoT reasoning and motivation retrieval for traceability.

Component summarization agent:

- **Component summarization.** The purpose of this prompt is to generate a textual summary of a component. As input, we give the properties of the component, as well as the properties of the parent container. Additionally, we give the truncated directory trees of the parent container, as well as the (truncated) contents of the files of the component. As a response, we expect a textual string of 200-400 words. We used emotional prompting techniques.

Container summarization agent:

- **Container summarization.** We use this prompt to generate a textual summary of a container. We give the container's properties as input, together with the truncated directory trees of the container as well as the summaries of the container's child components, which are generated using the previous prompt. We expect a textual summary of 200-400 words as an output.

Component relationship agent:

- **Component relationship identification.** We use this prompt to propose component-component relationships for a parent container. We provide the parent container summary and child component summaries as input, which should contain information about underlying relationships, as the previous agents were instructed to include this in the summaries. As an output, we expect a JSON list of component-component relationships. We use one-shot prompting by giving an example output, as well as CoT reasoning and motivation retrieval for traceability.

LLM and hyperparameter selections

For all agents, we use an Azure deployment of `gpt-4o-2024-08-06-dzs-we`. We chose this model because of availability within the host organization, and because of the overall performance of `gpt-4o` models in architectural tasks (as described in Section 2.7.2). For all prompts, we used a temperature of 0.1. We chose this temperature by trial-and-error. We set the maximum LLM response length for prompts to 8000 tokens, and the maximum prompt length to 36,000 characters. If an agent builds a prompt with a longer length, the prompt is truncated. As can be seen in Appendix A, we provide the most important instructions and information first in all prompts. This ensures that when a prompt is truncated during prompt building, the least important information is truncated. The number n in the top-n `tree` algorithm is configurable, and is set to 5 for the identification agents, and 3 for the component summarization agent.

Local PlantUML server

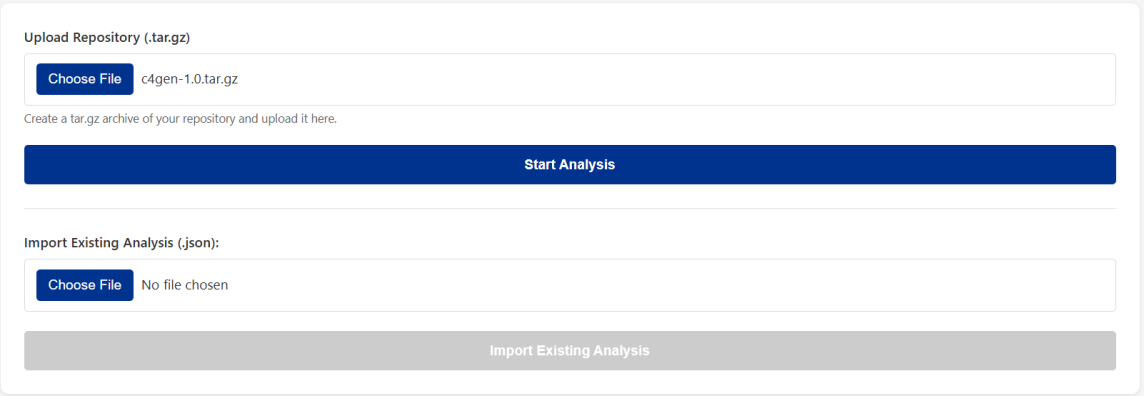
We run a local instance of a PlantUML server using the PlantUML server Docker image [55]. We use this local server to render C4 diagrams in SVG format, based on input C4-PlantUML strings. We chose to run the server locally, rather than calling a public endpoint, to ensure that we do not expose the source code data of input repositories to external endpoints. This is of importance when we run the POC on confidential source code repositories.

4.3 C4Gen: demonstration

To demonstrate how C4Gen can be used in practice, we demonstrate a worked example in this section. We show the sequential pages in the frontend in which the end user can interact with the system.

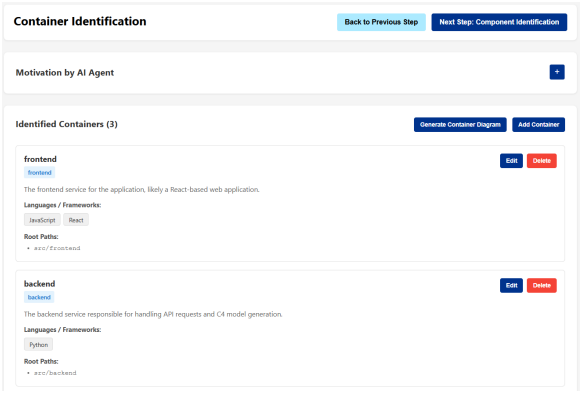
Step 1: uploading a repository

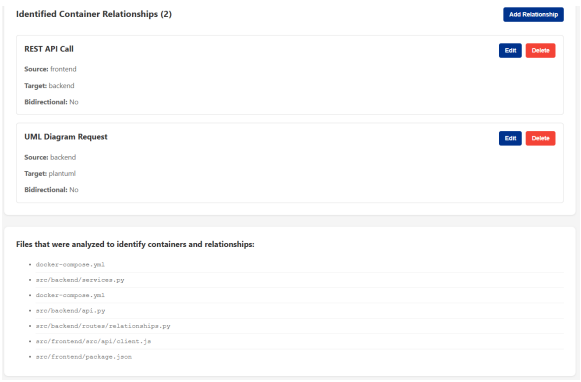
On the landing page, the user uploads a compressed archive format of a repository (which can be retrieved easily using most git-based platforms). This is the software repository for which C4-level SAD must be generated. Alternatively, the user can upload a JSON export file from a prior analysis. In that case, the next steps will show the information that was generated during that prior analysis.



Step 2: validating identified containers and container-container relationships

When the user uploads a repository, the container identification agent is called in the backend. When this agent has completed the analysis, identified containers and identified container–container relationships are shown to the user. On this page, the user can see and edit these containers and relationships, or create new ones. The user can click on the *"Generate Container Diagram"* button to get a preview of what the container diagram will look like in the SAD that will be created in the final step. Additionally, the user can see a textual motivation of why the agent has identified the specific containers and relationships in the *"Motivation by AI Agent"* tab. At the bottom, the user can see a list of files whose content was retrieved and used by the agent to come up with a judgment.

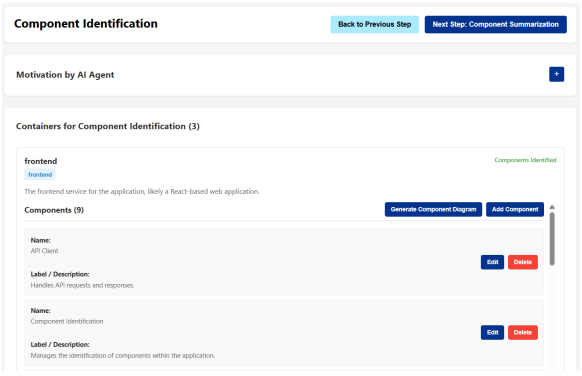


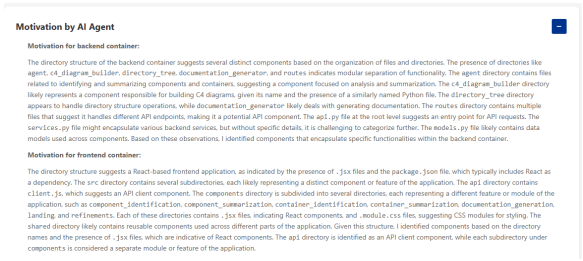


Step 3: generating and validating components for individual containers

When the identified containers have been validated, the user can click on an *"Identify Components"* button for individual containers. When the user presses this button, the component identifier agent is called in the backend for the corresponding parent container. This agent returns a list of components for the parent container. Subsequently, the user can see and edit these components. Additionally, the user can manually create new containers that are not identified by the agent. Like in the previous step, the user can see a preview of the component diagram that will be generated in the final step. Unlike in the previous step, these diagrams do not show relationships because they have not been identified yet. When

the user opens the *"Motivation by AI Agent"* tab, the user can again see a textual motivation about why the agent has identified which components.

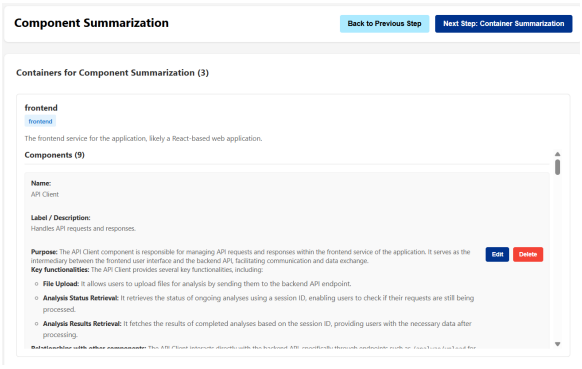


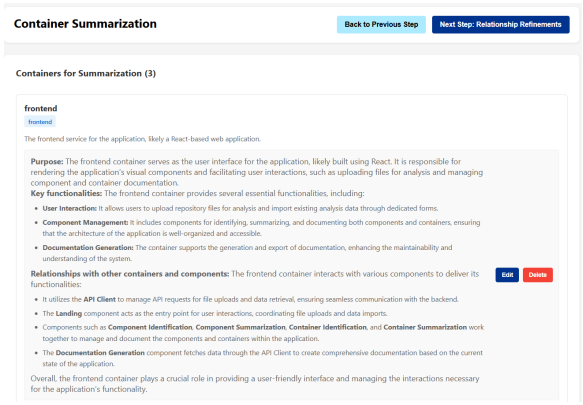


Steps 4 and 5: generating and validating component and container summaries

When all components have been identified and validated, the user can click on a *"Generate Summary"* button for individual components. When the user clicks on this button, the component summarizer agent is called in the backend. This agent returns a textual summary, which the user can see and edit. Alternatively, the user can bypass this agent and manually write summaries for components.

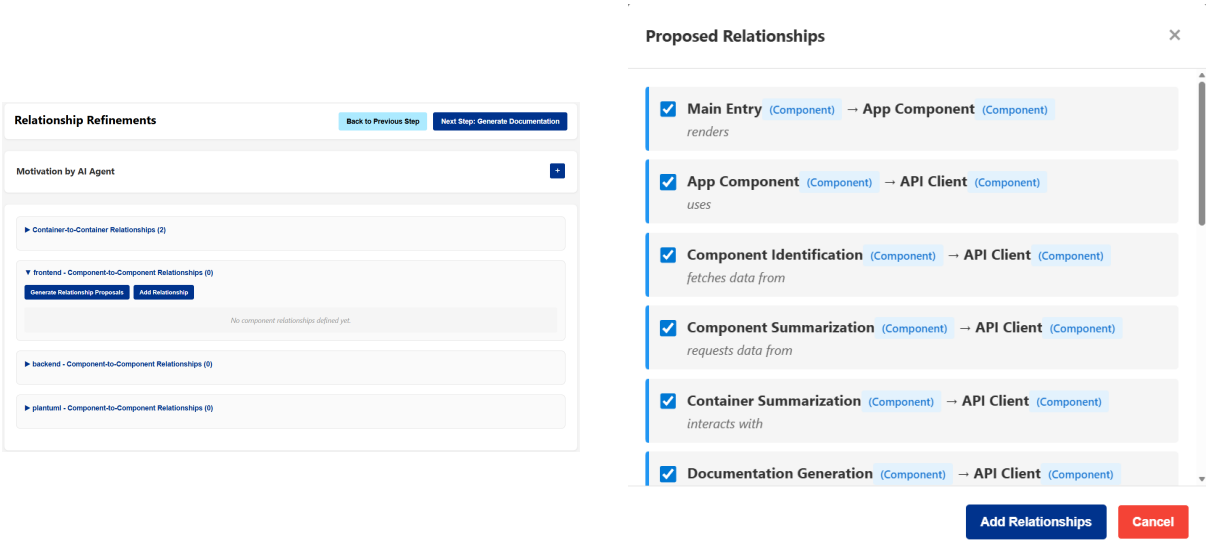
The next step is to generate summaries for containers. Like for components, the user can press a button, which calls the container summarizer agent in the backend. Also, over here, the user can bypass the agent by manually writing a container summary.





Step 6: generating and validating component-component relationships

When all summaries have been generated or written, the next step is to identify component–component relationships. The user can click on a *"Generation Relationship Proposals"* button for each parent container, which calls the component relationship identifier agent for the parent container. This agent proposes component–component relationships for all components within a parent container. The user can see the proposed relationships and choose whether to include them or not. Additionally, the user can manually create new relationships or edit proposed relationships. The agent provides a textual motivation for the relationship proposals, which the user can see by opening the *"Motivation by AI Agent"* tab. This motivation includes citations from the summaries, which the LLM used to come up with a judgment. The user can also see and edit the container–container relationships that the container identifier agent identified earlier.



Step 7: generating and exporting SAD

The final step is to generate SAD. The user can do this by navigating to the final page, which automatically calls the final deterministic algorithm in the backend. This algorithm aggregates all previously created metadata and summaries and parses them into a markdown string with diagrams. The user can view the SAD artifact (a rendered version of this markdown string) and download the SAD as a markdown file. Additionally, the user can export the analysis by pressing the *"Export Analysis"* button. When this is done, the user's web browser downloads a JSON file, which the user can upload again in the first step.

Chapter 5

Evaluation

This chapter presents the results of the evaluation of C4Gen. As described in Section 3.3, we evaluate using three dimensions of quality. First, we show the baseline results of `GitHub Copilot` in Section 5.1. Next, we show the top-line results of `Claude Code` in Section 5.2. Subsequently, we show the quantitative results of the syntactic quality evaluation for C4Gen in Section 5.3. After that, we show the quantitative results of the semantic quality evaluation for C4Gen in Section 5.4. Section 5.5 shows the results of our POC output consistency study. We provide the qualitative answers for C4Gen that were given to the pragmatic quality questions in Section 5.6. Finally, we indicate the token usage and estimated costs of C4Gen in Section 5.7. The diagrams that were used for syntactic and semantic evaluation can be found in Appendix B.

5.1 Baseline results (GitHub Copilot)

We determine a quality baseline by evaluating the syntactic and semantic quality of diagrams generated by `GitHub Copilot` using the prompt as described in Section 3.3. Appendix B contains the generated diagrams. Table 5.1 shows the baseline syntactic error rates for container diagrams that were generated for three input projects. Table 5.2 shows the syntactic error rates for generated component diagrams. Table 5.3 shows the normalized semantic scores for the diagrams generated by `GitHub Copilot`, and Table 5.4 shows the hallucination rates.

We observe the following visually in the `GitHub Copilot`-generated diagrams within Appendix B:

1. For the Abacus system, we cannot render the generated PlantUML strings due to syntax errors; we evaluate the syntactic and semantic quality of these diagrams based on the contents of the raw generated PlantUML strings
2. The container diagrams in Figures 19 and 39 show that generated container diagrams sometimes include `Person` stereotypes
3. The component diagrams in Figures 23 and 27 show that component diagrams sometimes include parent containers, but the relations with their children are not made explicit
4. In the component diagram in Figure 43, we observe a relation that is connected to an undefined node
5. The component diagrams in Figures 23, 27, 43, 47 and 51 show that generated component diagrams include component technologies

We observe the following in Tables 5.1–5.4:

1. Syntactic errors are present in the diagrams that were generated for Abacus
2. None of the input projects have an acceptable mean $S_{\text{normalized}}$
3. The mean $S_{\text{normalized}}$ for containers is acceptable, but not every individual project has an acceptable score

4. The mean $S_{\text{normalized}}$ for container–container relationships is acceptable, but not every individual project has an acceptable score
5. None of the input projects have an acceptable $S_{\text{normalized}}$ for components
6. None of the input projects have an acceptable $S_{\text{normalized}}$ for component–component relationships; all these scores are near total imperfection
7. Hallucinations are only present in the container diagram that was generated for C4Gen

Input project	n_{error}	n_{total}	*ErrorRate*
C4Gen	0	11	0.0%
AI4MDE	0	14	0.0%
Abacus	4	16	25.0%
Overall	4	41	9.8%

Table 5.1: Baseline syntactic error rates in the generated container diagram for different input projects, generated using `GitHub Copilot`.

Input project	n_{error}	n_{total}	*ErrorRate*
C4Gen	0	22	0.0%
AI4MDE	0	22	0.0%
Abacus	1	17	5.9%
Overall	1	61	1.6%

Table 5.2: Baseline syntactic error rates in generated component diagrams for different input projects, generated using `GitHub Copilot`. n_{error} and n_{total} are the aggregated number of (erroneous) elements for all generated component diagrams for an input project.

Input project	$S_{\text{normalized}}$ (cont.)	$S_{\text{normalized}}$ (cont.–cont. rel.)	$S_{\text{normalized}}$ (comp.)	$S_{\text{normalized}}$ (comp.–comp. rel.)	$S_{\text{normalized}}$ (mean)
C4Gen	0.61	0.50	0.28	0.15	0.38
AI4MDE	0.92	0.90	0.14	0.07	0.51
Abacus	0.94	0.88	0.59	0.08	0.62
mean	0.82	0.76	0.34	0.10	0.50

Table 5.3: Baseline normalized semantic quality scores for different element types in the generated diagrams (generated using `GitHub Copilot`), as evaluated by the authors. Scores that are higher than the heuristic acceptability threshold ($S_{\text{normalized}} \geq 0.75$) are given in green. Lower scores are given in red.

Input project	*hallu*% (cont.)	*hallu*% (cont.–cont. rel.)	*hallu*% (comp.)	*hallu*% (comp.–comp. rel.)	*hallu*% (mean)
C4Gen	25.0	33.3	0.0	0.0	15.0
AI4MDE	0.0	0.0	0.0	0.0	0.0
Abacus	0.0	0.0	0.0	0.0	0.0
mean	8.3	11.1	0.0	0.0	5.0

Table 5.4: Baseline hallucination rates for different element types in the generated diagrams (generated using `GitHub Copilot`), evaluated by the authors. Rates that are lower than the heuristic acceptability threshold (*hallu*% $\leq 10\%$) are given in green. Higher rates are given in red.

5.2 Top-line results (Claude Code)

As described in Section 3.3, we determine a quality top-line by evaluating the syntactic and semantic quality of diagrams generated by `Claude Code` using the prompt in Appendix A.9. Appendix B contains

the generated diagrams. Table 5.5 shows the baseline syntactic error rates for container diagrams that were generated for three input projects. Table 5.6 shows the syntactic error rates for generated component diagrams. Table 5.7 shows the normalized semantic scores for the diagrams generated by `Claude Code`, and Table 5.8 shows the hallucination rates. Table 5.9 shows the costs that are associated with running `Claude Code` for different input projects.

We observe the following visually in the `Claude Code`-generated diagrams within Appendix B:

1. All generated PlantUML strings use valid PlantUML syntax and can be rendered
2. The generated container diagrams for all three input systems (Figures 20, 32, and 40) include `Person` and `External System` stereotypes
3. The container diagram for AI4MDE (Figure 40) includes the `External Container` stereotype
4. Container–container relationships in the generated container diagrams for all input systems (Figures 20, 32, and 40) include the type of communication technology and which ports are used for communication
5. All generated component diagrams specify which technologies are used in components
6. Component diagrams include calls to external systems and containers (see Figure 24 for an example)
7. Overall, container descriptions and component descriptions specify extensive technical details about the container/component

We observe the following in Tables 5.5–5.9:

1. No syntactic errors are present in any of the diagrams
2. All input projects have acceptable semantic scores, both for element types individually as well as overall
3. The mean $S_{\text{normalized}}$ scores for input projects have a range of 0.82–0.96
4. The mean $S_{\text{normalized}}$ scores for for element types have a range of 0.87–0.92
5. Hallucinations are only present in the container diagram that was generated for AI4MDE. This diagram has unacceptable hallucination rates
6. Costs that are associated with generating the C4 diagrams using Claude Code are manageable (range \$1.85–2.93)

Input project	n_{error}	n_{total}	$ErrorRate$
C4Gen	0	15	0.0%
AI4MDE	0	29	0.0%
Abacus	0	12	0.0%
Overall	0	56	0.0%

Table 5.5: Top-line syntactic error rates in the generated container diagram for different input projects, generated using `Claude Code`.

Input project	n_{error}	n_{total}	$ErrorRate$
C4Gen	0	147	0.0%
AI4MDE	0	141	0.0%
Abacus	0	115	0.0%
Overall	0	403	0.0%

Table 5.6: Top-line syntactic error rates in generated component diagrams for different input projects, generated using `Claude Code`. n_{error} and n_{total} are the aggregated number of (erroneous) elements for all generated component diagrams for an input project.

Input project	$S_{\text{normalized}}$ (cont.)	$S_{\text{normalized}}$ (cont.–cont. rel.)	$S_{\text{normalized}}$ (comp.)	$S_{\text{normalized}}$ (comp.–comp. rel.)	$S_{\text{normalized}}$ (mean)
C4Gen	0.94	0.92	0.95	0.91	0.93
AI4MDE	0.81	0.83	0.83	0.80	0.82
Abacus	1.00	1.00	0.91	0.92	0.96
mean	0.92	0.92	0.90	0.87	0.90

Table 5.7: Top-line normalized semantic quality scores for different element types in the generated diagrams (generated using `Claude Code`), as evaluated by the authors. Scores that are higher than the heuristic acceptability threshold ($S_{\text{normalized}} \geq 0.75$) are given in green. Lower scores are given in red.

Input project	*hallu*% (cont.)	*hallu*% (cont.–cont. rel.)	*hallu*% (comp.)	*hallu*% (comp.–comp. rel.)	*hallu*% (mean)
C4Gen	0.0	0.0	0.0	0.0	0.0
AI4MDE	14.3	12.5	0.0	0.0	6.7
Abacus	0.0	0.0	0.0	0.0	0.0
mean	4.8	4.2	0.0	0.0	2.2

Table 5.8: Top-line hallucination rates for different element types in the generated diagrams (generated using `Claude Code`), evaluated by the authors. Rates that are lower than the heuristic acceptability threshold (*hallu*% $\leq 10\%$) are given in green. Higher rates are given in red.

Input project	Lines of code	Associated costs
C4Gen	14,013	$2.93
AI4MDE	33,724	$2.32
Abacus	143,616	$1.85

Table 5.9: Costs for using `Claude Code` (Opus 4.7 Thinking model) to generate C4 diagrams for different input projects. Costs are tracked using `Claude Code`'s cost tracking feature, and are based on the current API pricing ($5 per million input tokens, $25 per million output tokens).

5.3 Syntactic quality (C4Gen)

We evaluate the syntactic quality of C4Gen-generated diagrams using the error rate metric that we defined in Section 3.3. Appendix B contains the generated diagrams. Table 5.10 shows the error rates as defined in Section 3.3 for generated container diagrams. Table 5.11 shows the error rates for generated component diagrams. We observe small syntactic error rates, caused by a total of only three syntactic errors. These errors are either caused by agents copying non-concise and non-descriptive names verbatim, rather than coming up with more informative names, or by explicitly mentioning in output descriptions that the agent has made assumptions from input data. We observe no syntactic errors in the rendering process.

Input project	n_{error}	n_{total}	*ErrorRate*
C4Gen (dogfooding)	0	11	0.0%
AI4MDE	0	15	0.0%
Abacus	1*	11	9.1%
Qubus	0	11	0.0%
Overall	1	37	2.7%

Table 5.10: Syntactic error rates in the generated container diagram for different input projects, generated using C4Gen. *This error is a container description from an externally managed database container that has been inferred from migration files. The description is not *concise* and not *descriptive* because it explicitly mentions that the container has been inferred from a migration file, and might not exist at all.

Input project	n_{error}	n_{total}	$ErrorRate$
C4Gen (dogfooding)	2*	104	1.9%
AI4MDE	0	91	0.0%
Abacus	0	165	0.0%
Qubus	0	61	0.0%
Overall	2	317	0.6%

Table 5.11: Syntactic error rates in generated component diagrams for different input projects, generated using C4Gen. n_{error} and n_{total} are the aggregated number of (erroneous) elements for all generated component diagrams for an input project. *Both errors are component names that are not *descriptive*, which are names copied verbatim from source code.

5.4 Semantic quality (C4Gen)

As described in Section 3.3, input project stakeholders evaluate the semantic quality of C4Gen's output. A total of eleven evaluations are done, one for the C4Gen (dogfooding) case, four for AI4MDE, four for Abacus, and two for Qubus. Table 5.12 shows the professional roles of the input project stakeholders together with their relation to the input project. Appendix B contains the generated diagrams. The four tables in Appendix C show the classification distribution by individual stakeholders for the four different element types, together with their corresponding computed S_{raw} and $S_{\text{normalized}}$ scores and hallucination rates. Table 5.13 shows the computed $S_{\text{normalized}}$ scores for different stakeholders for all element types, grouped by input project. The columns in this table show $S_{\text{normalized}}$ for different element types, as well as the mean (equal weights). Additionally, we calculate the mean for all input projects. In the final row of the table, we provide the mean of these project means. We provide scores that adhere to the heuristic acceptability thresholds in a green color. Scores that do not adhere are in red. Table 5.14 has the same structure, but shows computed hallucination rates.

We observe the following numerically:

1. All four input projects have acceptable overall normalized mean scores
2. Across all four input projects, the range of overall normalized mean scores is 0.77–0.92
3. Different stakeholders do not always agree on a classification for the same element (see Appendix C); moreover:
   (a) Different stakeholders do not always agree on which elements are hallucinated and which are not
   (b) Different stakeholders do not always agree on missing elements (false negatives)
4. One out of four input systems does not have an acceptable mean score for container–container relationships
5. Two out of four input systems do not have acceptable mean scores for component–component relationships
6. All five mean of means scores are acceptable
7. Two out of four input systems have unacceptable hallucination rates for container–container relationships
8. The mean of means hallucination rate for container–container components is unacceptable

In qualitative feedback that was supplied together with the semantic evaluations, we observe the following:

1. Respondents for the same input project (especially respondents with different professional roles) have different expectations of architecture documentation, and sometimes classify elements differently
2. Semantic evaluation requires an understanding of the C4 model
3. In component diagrams, one respondent (**P4**) expects external dependencies on frameworks that are used

4. One respondent (**P4**) raises questions about cross-repository analysis
5. AI4MDE stakeholders agree that generated diagrams are mostly acceptable, but minor adjustments are required before they can be put into practical use
6. Abacus stakeholders agree that the generated container diagram is acceptable, but one stakeholder (**P7**) debates the acceptability of component-level elements
7. Qubus stakeholders do not agree that the generated diagrams are acceptable. One stakeholder (**P9**) requires only one minor correction in the container diagram, whilst the other (**P8**) requires major adjustments in the container diagrams and minor adjustments in the component diagram

Participant	**Professional role**	**Assigned diagrams**	**Relation to project**
P1 (author)	Student	C4Gen	Project owner
P1 (author)	Student	AI4MDE	Developer
P2	Student	AI4MDE	Developer
P3	Developer	AI4MDE	Technical architect
P4 (supervisor)	Assistant professor	AI4MDE	Project owner
P1 (author)	Student	Abacus	No relation
P5	Senior consultant/Manager	Abacus	Auditor
P6	Senior consultant	Abacus	Auditor
P7	Senior consultant	Abacus	Auditor
P8	Tech specialist	Qubus	Developer
P9	Tech specialist	Qubus	Developer

Table 5.12: Profiles for different evaluation participants.

Participant		$S_{\text{normalized}}$ (cont.)	$S_{\text{normalized}}$ (cont.–cont. rel.)	$S_{\text{normalized}}$ (comp.)	$S_{\text{normalized}}$ (comp.–comp. rel.)	$S_{\text{normalized}}$ (mean)
(C4Gen)	mean	1.00	1.00	0.82	0.59	0.85
	P1	1.00	1.00	0.82	0.59	0.85
(AI4MDE)	mean	0.94	0.79	0.84	0.73	0.83
	P1	1.00	0.57	0.83	0.82	0.80
	P2	1.00	0.57	0.86	0.65	0.77
	P3	1.00	1.00	0.84	0.69	0.88
	P4	0.75	1.00	0.84	0.75	0.83
(Abacus)	mean	0.98	0.97	0.86	0.88	0.92
	P1	1.00	1.00	0.83	0.84	0.92
	P5	1.00	1.00	0.99	0.93	0.98
	P6	0.92	0.88	0.88	1.00	0.92
	P7	1.00	1.00	0.75	0.75	0.88
(Qubus)	mean	0.75	0.65	0.83	0.86	0.77
	P8	0.50	0.67	0.66	0.71	0.64
	P9	1.00	0.63	1.00	1.00	0.91
mean of means		0.91	0.85	0.84	0.77	0.84

Table 5.13: Normalized semantic quality scores for different element types in the generated diagrams (generated using C4Gen), as evaluated by individual stakeholders. Mean scores that are higher than the heuristic acceptability threshold ($S_{\text{normalized}} \geq 0.75$) are given in green. Lower mean scores are given in red.

Participant		*hallu*% (cont.)	*hallu*% (cont.–cont. rel.)	*hallu*% (comp.)	*hallu*% (comp.–comp. rel.)	*hallu*% (mean)
(C4Gen)	mean	0.0	0.0	0.0	0.0	0.0
	P1	0.0	0.0	0.0	0.0	0.0
(AI4MDE)	mean	0.0	16.7	2.4	4.6	5.9
	P1	0.0	33.3	0.0	7.4	10.2
	P2	0.0	33.3	0.0	7.4	10.2
	P3	0.0	0.0	0.0	3.7	0.9
	P4	0.0	0.0	9.7	0.0	2.4
(Abacus)	mean	0.0	0.0	0.0	0.0	0.0
	P1	0.0	0.0	0.0	0.0	0.0
	P5	0.0	0.0	0.0	0.0	0.0
	P6	0.0	0.0	0.0	0.0	0.0
	P7	0.0	0.0	0.0	0.0	0.0
(Qubus)	mean	0.0	25.0	0.0	3.6	7.2
	P8	0.0	50.0	0.0	7.1	14.3
	P9	0.0	0.0	0.0	0.0	0.0
mean of means		0.0	10.4	0.6	2.1	3.3

Table 5.14: Hallucination rates for different element types in the generated diagrams (generated using C4Gen), evaluated by individual stakeholders. Rates that are lower than the heuristic acceptability threshold ($hallu\% \leq 10\%$) are given in green. Higher rates are given in red.

5.5 Output consistency (C4Gen)

As described in Section 3.3, we evaluate the consistency of C4Gen's output by running C4Gen for $k = 3$ times (end-to-end) on its own source code. This is an indicative practical check. We evaluate the consistency of the generated container and components diagrams individually by using the rubric in Table 3.3. Appendix B.4 contains the generated diagrams for all runs.

We observe the following for the generated sets of diagrams:

1. Container diagrams:
   (a) All three diagrams contain functionally identical architectural information
   (b) In the containers, there are only trivial linguistic variations
   (c) There are variations in relationship labels, but all of these are functionally correct
   (d) The diagrams can be used interchangeably
2. Component diagrams (frontend):
   (a) All three diagrams have functionally identical components
   (b) In these components, there are only trivial linguistic variations
   (c) There are minor, trivial variations in generated relationships; some relationships that are present in one diagram are omitted in another
   (d) The diagrams can be used interchangeably
3. Component diagrams (backend):
   (a) The generated diagrams are not functionally identical
   (b) The architectural viewpoint is mostly preserved across different diagrams, but critical architectural information is aggregated/separated/omitted
   (c) Separate runs partially agree on a common architecture
   (d) The diagrams cannot be used interchangeably

Based on these observations, we assign the following ratings for these specific outputs:

Set of diagrams	Assigned rating
Container diagrams	Full consistency
Component diagrams (frontend)	High consistency
Component diagrams (backend)	Low–moderate consistency

Table 5.15: Consistency ratings for the generated sets of diagrams in our consistency study.

5.6 Pragmatic quality (C4Gen)

As described in Section 3.3, we evaluate the pragmatic quality of our design by organizing feedback sessions and sending out a survey to input project stakeholders who have an IT consulting profession. Table 5.16 shows a list of pragmatic evaluation stakeholders, together with their professional role and relation to the assigned input project. Appendix D contains tables with responses to the survey questions. Table D.1 provides an overview of all responses, in which the full responses are summarized. Tables D.2–D.10 contain the full, non-summarized responses to all questions.

Participant	Professional role	Assigned project	Relation to project
P1	Consultant	Abacus	Code reviewer
P2	Senior consultant	Abacus	Code reviewer
P3	Director	Abacus	Code reviewer
P4	Consultant	Abacus	Limited knowledge on project
P5	Tech consultant	Qubus	Familiar with project

Table 5.16: Profiles for different evaluation participants.

We observe the following in Tables D.1– D.10:

1. Four out of five respondents (**P1**, **P2**, **P3**, and **P5**) have an overall positive impression of C4Gen
2. One respondent (**P4**) has a neutral overall impression
3. Four out of five respondents (**P1**, **P3**, **P4**, and **P5**) believe that the generated diagrams (partially) cover the input system's architecture
4. Two out of five respondents (**P2** and **P3**) have concerns regarding the consistency of C4Gen's output
5. All five respondents foresee themselves using C4Gen in their operations
6. Three out of five respondents (**P1**, **P3**, and **P4**) think C4Gen would improve their productivity
7. Four out of five respondents (**P1**, **P2**, **P3**, and **P5**) think C4Gen would improve their effectiveness
8. The following limitations are noticed/missing functionalities are mentioned:
   (a) It is more intuitive to only allow manual identification of elements after the creation of diagrams, rather than also allowing manual identification beforehand (**P2**)
   (b) Component–component relationships between components that are within different parent containers are missing (**P3**)
   (c) Technologies used by individual components are not shown (**P3**)
   (d) There is no option to export the analysis as a PDF/ODT/DOCX file (**P3**)
   (e) There are opportunities for advanced (interactive) visualizations (**P4**)
   (f) Functionalities that make the agentic steps more observable could be added (**P4**)
   (g) Versioning could be added (**P4**)
9. All five respondents think C4Gen is easy to use

10. Three out of five respondents (**P2**, **P3**, and **P5**) mention that UI/UX improvements are needed
11. The following improvements are suggested to make C4Gen easier to use:
    (a) Clarifying text could be added in the headers (**P3**)
    (b) Navigation could be improved (**P2**)
    (c) Loading/error states could be improved (**P2**)
    (d) Accidental deletions should be able to be reversed (**P3**)
    (e) Users should be able to see which file they have uploaded (**P5**)
    (f) Bulk editing would be helpful (**P4**)
    (g) Versioning could be added (**P3** and **P4**)
    (h) Locations of some elements could be reconsidered (**P2** and **P5**)
    (i) Sizes of some elements could be reconsidered (**P5**)
12. Three out of five respondents (**P1**, **P3**, and **P5**) have the definite intention of using C4Gen in their operations
13. The remaining two respondents (**P2** and **P4**) have a possible intention of using C4Gen in their operations

5.7 Token usage (C4Gen)

Table 5.17 shows the token usage by C4Gen for end-to-end runs for the different input projects. The token usage of a run encompasses all API calls that were used in the end-to-end SAD generation workflow (uploading an input project to creating the final SAD). We provide the number of lines of code (computed using *cloc* [15]) for the different input projects, together with the corresponding input and response tokens. We compute associated costs using current prices for the Azure `gpt-4o-2024-08-06-dzs-we` API ($2.50 per 1M input tokens, $10.00 per 1M output tokens [5]). We observe that token usage and associated costs do not scale linearly for input project size, and for all input projects, the associated costs are manageable (range $0.35–0.73).

Input project	Lines of code	Input tokens	Response tokens	Total	Associated costs
C4Gen	14,013	81,669	14,148	95,817	$0.35
AI4MDE	33,724	169,039	26,705	195,744	$0.69
Abacus	143,616	203,859	21,912	225,771	$0.73
Qubus	38,805	216,168	15,008	231,176	$0.69

Table 5.17: Token usage by C4Gen for different input projects. Token amounts are aggregated for all API calls in a single end-to-end run.

Chapter 6

Discussion

In this chapter, we discuss and interpret the results that we provided in Chapter 5 and connect them to the literature that we discussed in Chapter 2. In Section 6.1, we discuss the results of our syntactic and semantic evaluation. In Section 6.2, we discuss the limitations of our solution. After that, in Section 6.3, we discuss the results of our pragmatic evaluation and relate these findings to the practical applicability of our solution in an IT consulting context. Next, we iterate over threats to the validity of our research in Section 6.4. In Section 6.5, we reflect on our key findings related to mechanisms in AI-driven software recovery. Finally, in Section 6.6, we reflect on what our research indicates in the context of agentic approaches in software architecture.

6.1 Quality of C4Gen output

As explained in Section 3.3, we evaluate the generated diagrams using syntactic and semantic evaluation metrics. This section interprets and discusses the syntactic and semantic evaluation results that we provided in Chapter 5. First, we analyze the syntactic quality of the generated diagrams. Secondly, we analyze their semantic quality. For both quality dimensions, we compare the results with the baseline `GitHub Copilot` results and the top-line `Claude Code` results, and we relate our findings to existing literature. Additionally, we discuss the results of our consistency study.

Syntactic quality

The syntactic evaluation results (Section 5.3) show that C4Gen consistently generates diagrams with low syntactic error rates. Most generated elements adhere to the *well-formed*, *concise*, and *descriptive* properties that we defined in Section 3.3. The few observed errors are minor and only relate to the verbatim copying of non-descriptive names in source code and the inclusion of meta-information in descriptions. These errors do not impact the rendering of the diagrams and could be corrected easily during the human-in-the-loop validation steps or by refining the prompts that are used.

C4Gen shows low syntactic error rates when compared to the baseline syntactic error rate (`GitHub Copilot`, Section 5.1). This difference indicates an improvement in syntactic quality in the generated output. Moreover, some of the `GitHub Copilot`-generated diagrams contain PlantUML syntax errors and cannot be rendered. Such issues do not occur when using C4Gen because it creates PlantUML strings deterministically. This design choice ensures that stochastic LLM reasoning is separated from syntax generation.

Whereas C4Gen-generated diagrams have lower syntactic error rates than the baseline diagrams, they have higher error rates than the top-line diagrams (`Claude Code`, Section 5.2). `Claude Code` achieves zero syntactic errors across all evaluated diagrams (Tables 5.5 and 5.6), while C4Gen shows small residual syntactic errors (Tables 5.10 and 5.11). This indicates that the top-line has a slightly better performance in generating container- and component-level elements that adhere to the *well-formed*, *concise*, and *descriptive* properties. We assume this increase in performance to be caused by the fully autonomous agentic architecture of `Claude Code` in combination with an RCI prompting strategy, but it could also be caused by a stronger underlying LLM (or a combination of the two). Further research could investigate the effect of these variables.

The design choice of separating stochastic LLM reasoning from syntax generation contributes to AI transparency and reliability of the solution because the syntax of C4Gen's output is predictable, as LLMs only influence the semantic content of the output. By eliminating stochasticity in the rendering step, C4Gen reduces the risk of rendering errors and ensures that issues in the generated diagrams can be traced back to earlier stages in the pipeline. The top-line study, however, suggests that this design decision may not be a strict requirement, because `Claude Code` (a fully autonomous agentic solution) also produces fully renderable diagrams for all input systems. This suggests that iterative self-refinement in agentic solutions, for example, by using the RCI-mechanism, can mitigate syntax errors without enforcing deterministic PlantUML string creation. C4Gen's design could be altered to include RCI prompts; further research would need to be conducted to investigate whether and how such an inclusion would improve the syntactic quality of C4Gen's output. However, using the RCI-mechanism to generate PlantUML strings using LLMs is more costly than deterministically creating PlantUML syntax with LLM-generated semantics. Thus, if C4Gen's prompts could be slightly adjusted (without adding recursion/iteration) such that generated metadata is *well-formed*, *concise*, and *descriptive*, syntactic quality could improve without a substantial increase in costs.

Semantic quality

The results in Section 5.4 and corresponding classifications in Appendix C demonstrate that C4Gen can produce C4 container and component diagrams that are acceptable and architecturally meaningful, with clear improvements over the baseline. Table 5.13 shows that, across all four input systems, the mean normalized scores are higher than the heuristic acceptability threshold (0.75), ranging from 0.77 to 0.92. This is higher than the mean normalized scores that `GitHub Copilot` achieves, which range from 0.38 to 0.62 (Table 5.3) and are particularly low for elements in component diagrams. These results indicate that our solution improves the semantic quality of generated diagrams when compared to a simple zero-shot approach. Compared to the top-line, `Claude Code` achieves slightly higher mean normalized semantic scores overall (0.82–0.96 across projects, Table 5.7) than C4Gen, showing that C4Gen performs similarly to a state-of-the-art fully agentic solution, but still trails it, especially regarding component-level relationships.

Table 5.13 shows that C4Gen-generated container diagrams tend to have higher semantic quality scores than their corresponding component diagrams. Container-level elements achieve high scores, and many of the individual stakeholders give a perfect semantic quality score (1.00) for these container-level elements. This means that these individual stakeholders believe all the elements are perfect matches, and there are no hallucinations or missing elements. This indicates that high-level (container-level) system decomposition can be inferred by our first agent, which suggests that low-level source code analysis may not be needed for the extraction of container-level insights. Component-level elements are also acceptable, but show more variability in the semantic quality scores. This increase in variability in lower-level elements resonates with the results of our consistency study and with qualitative feedback that is given by evaluation stakeholders.

We observe a pattern that relationships perform worse overall when compared to structural elements. In Table 5.13, both container–container and component–component relationships show a lower mean of means semantic score than their structural counterparts. We can partially explain this pattern by the dependency of relationships on their structural counterparts. If an erroneous container/component is present in a diagram, the error propagates to connected relationships. For example, if a container is classified as a hallucination, by definition, all connected container–container relationships will also be classified as a hallucination. This resonates with the increased hallucination rates for relationships that we observe in Table 5.14.

In this table, we observe hallucination rates for container–container relationships that are higher than the heuristic acceptability threshold. We believe that these high rates are caused by how our solution identifies this type of relationship. As described in Section 4.1, the container identification agent infers container–container relationships, and only takes the repository's directory tree and the contents of specific high-level files (e.g., Docker Compose or configuration files) as input data. Unlike the component–component relationship identifier agent, which identifies component–component relationships based on the summaries of actual source code, source code is not included as context by the container identifier agent, which might explain why the hallucination rates for container–container relationships are higher than component–component relationships.

We can also explain the relatively low performance for relationships compared to structural elements by the more abstract nature of relationships. Whereas containers and components are concrete artifacts that can often be mapped to source code or directory structures, relationships are interactions between artifacts, which can be implicit or distributed across multiple files. As a result, the identification of relationships introduces uncertainty, leading to possible lower semantic scores and higher hallucination rates compared to structural elements.

Additionally, there is a realistic possibility in LLM-based approaches that relationships (or even structural elements) are inferred from *"what is likely to be there"* instead of strictly deriving them from source code evidence. For example, if backend and database containers are present in a system, an LLM might infer the presence of a database query relationship, even if there is no strict evidence of such queries in the source code. Especially for higher-level abstractions such as container–container relationships, LLMs may construct relationships that are logically consistent but not present in reality. Such potential gaps emphasize the importance of human validation of AI-generated diagrams.

In the top-line results, hallucination rates are lower on average (mean 2.2% in Table 5.8), but there is a notable exception for AI4MDE at the container level. The hallucinated elements in the corresponding container diagram in Figure 40 (*Generated Prototypes Filesystem* container and the connected relationship) do not exist in AI4MDE in the exact way as described by `Claude Code`. In reality, AI4MDE's logic that corresponds to these container-level elements is implemented in a messy, non-standard manner. `Claude Code` might have difficulties with correctly describing this logic because it is a hacky and non-standard workaround. This illustrates that even state-of-the-art fully autonomous agentic solutions might struggle with extracting more abstract container-level insights, especially if more detailed and nuanced information is needed to correctly capture these insights.

Missing elements (false negatives) are present across all four input systems and impact semantic scores. The presence of these missings indicates that C4Gen does not always capture all expected architectural elements. This resonates with the qualitative feedback that was given by evaluation stakeholders, who state that diagrams do not always reflect some parts of the input system's architecture. The tables in Appendix C show that component-level elements have a higher number of missings than container-level missings. We can partially explain this higher number by the fact that component diagrams are usually bigger than their corresponding container diagrams, and thus, a higher number of elements is expected overall. However, some evaluation stakeholders mentioned in their qualitative feedback that they expected component diagrams to either be more detailed or less detailed than the current version of the diagram. This indicates a difficulty with component diagrams in the C4 model, as the level of abstraction that should be reflected in component diagrams might differ for different types of stakeholders.

We also observe such differences in expectations in other classification types in Appendix C. Different stakeholders do not always agree on what elements can be classified as a match, valid, irrelevant, hallucination, or missing. We can explain these differences by the concept of architectural viewpoints, which we discussed in Section 2.1.1. Different stakeholders expect different representations of a system, as they have different concerns and relations with the system. As a result, semantic quality is partially subjective, and there is no perfect ground-truth C4 representation of a system. These observations resonate with findings in Pex's study that we elaborated upon in Section 2.7.2, who mentioned that *"a 'good explanation' is not universal and depends on the personality of the person"* [53].

We also observe different semantic quality scores and hallucination rates between different input systems. Abacus achieves the highest overall normalized mean score, whereas Qubus achieves the lowest. This indicates that the semantic quality of C4Gen's output may depend on the characteristics of the input system (e.g., programming language, small / large system, consistent / inconsistent architecture, standard / niche architecture). However, the pool size of input projects and evaluation stakeholders is too small to draw any definitive conclusions regarding such dependencies.

Finally, we must note that we only evaluate the semantics of generated diagrams, and not of the additional textual summaries that C4Gen provides. Moreover, the human-in-the-loop mechanism was not used during the generation of the diagrams that we use for semantic evaluation. We choose to ignore this mechanism so that we can evaluate the semantic quality of generated diagrams without any human intervention.

Overall, C4Gen has a strong performance in the identification of structural C4 elements and in the extraction of container-level insights. The identification of relationships and the extraction of component-level insights is less accurate, but still above the heuristic acceptability threshold.

Output consistency

The findings in Section 5.5 highlight a difference between output consistency in the container- and component-level output of C4Gen.

The consistency study indicates that C4Gen's output is stable at the container level. A *full consistency* rating is justified because the three different container diagrams in Figure 53 show functionally identical architectural information. The only differences are linguistic (e.g., the description *"a microservice for generating UML diagrams using the PlantUML server"* versus *"a service for generating UML diagrams using PlantUML"*). The same containers and container–container relationships are present in all three container diagrams. All container names, container descriptions, relationships, and relationship labels can be used interchangeably, which is a necessity for the *full consistency* rating.

The generated component diagrams show lower consistency. In the three component diagrams that are generated for the frontend container (Figure 54), the same nine components are present in each diagram. These components only have trivial linguistic variations, similar to the linguistic variations in the generated container diagrams. However, the diagram in Figure 54c has two additional component–component relationships (outgoing relationships from the *Refinements* component) that are omitted in the other two diagrams. These variations are minor, trivial variations, because only detailed relationships are included/excluded in different diagrams, but the overall architectural viewpoint is maintained. The generated diagrams can be used interchangeably. These findings justify a *high consistency* rating.

The diagrams that are generated for the backend container (Figure 55) show an even lower consistency. Firstly, there is a great difference in the number of components (21 in the first run, 8 in the second run, 9 in the third run). These variations show a difference in architectural depth that is represented by the diagrams. For example, the diagram in Figure 55a explicitly defines an individual *Route* component for different functionalities, whereas the diagram in Figure 55c aggregates all these components under one *API Routes* component. The different diagrams share the same architectural viewpoint but differ in the level of depth that is represented. The different diagrams partially agree on a common architecture, but the diagrams are too different to be able to be used interchangeably. Because of these observations, a moderate consistency cannot be fully justified, which is why a low–moderate rating is assigned.

The findings and ratings indicate that C4Gen appears to be reliable for high-level (containers) architectural recovery but becomes more unpredictable at lower abstraction levels (components). The high consistency at the container level makes the tool reliable for retrieving quick, high-level overviews of input systems. Variability increases at the lower abstraction level, which increases the importance of manual human validation at this level. Manual human validation is of extra importance in the case of variability because a variety of architectural representations can all be factually correct, but if human preferences are taken into account, some representations could be 'better' than others. By enforcing human validation, a human can choose the 'better' option, rather than letting an algorithm/machine choose between several variations. The findings and implications of the consistency study should be interpreted with caution because of the limited number of runs, and because we only evaluate C4Gen's consistency using one input system.

6.2 Solution limitations

In this section, we will highlight the limitations of our solution. This list is not complete and only focuses on the most important limitations.

Only container and component insights are extracted

As explained in Section 2.1.2, the C4 model is composed of four hierarchical abstraction levels, but our design only generates diagrams for the two levels that describe a single system's static structure (container and component level). A perfect design would capture architectural insights from all abstraction levels, including the system context and code levels. Our current design does not allow such analyses, but as can be read in Section 5.6, some pragmatic evaluation stakeholders do expect such abstraction levels.

This limitation is a consequence of our research's scope and design decisions during development. During technical due diligence operations on an individual software system, container- and component-level insights provide the most immediate insights as they illustrate the static structure of an individual software system. Moreover, system context diagrams are more difficult to create using source code as

the only input data and require more business-oriented data, such as process documentation, stakeholder interview transcriptions, or active monitoring of system usage in a dynamic environment. If users want a system context-level understanding of a set of software systems, C4Gen is not applicable in its current state.

Only using source code as input data

One of the crucial limitations of our solution is that it only allows source code as input data. As described in Section 2.1.1, ADs are architecture views that take into account stakeholder concerns that are relevant to the viewpoint of the stakeholder. Our solution only takes into account the technical details of an input project by conducting an agentic static source code analysis, and omits any non-technical, stakeholder-oriented details that might be relevant for setting up an architecture view. For example, stakeholder concerns may determine which containers/components are to be expected in a specific viewpoint for that individual stakeholder, as big systems can grow complex, and individual stakeholders might only be interested in a subset of the static representation of the system.

System decomposition instead of pattern recovery

Another key limitation of C4Gen is its inability to decompose systems in a way that not only exposes structural elements, but also architectural tactics that realize (non-functional) requirements. As described in Section 2.1.2, the C4 model is a notation that can be used to decompose the static structure of a system into hierarchical abstractions. While C4 diagrams describe what containers and components a system is composed of, they do not *explicitly* show whether the system follows specific architecture patterns (e.g., MVC, event-driven, layered). Our current solution does not extract whether such patterns are in the source code. As a result, the solution's output only supports the understanding of an input system's building blocks, but not of intended and recurring architectural design patterns.

External dependencies are not included in diagrams

Our current solution design only retrieves component–component relationships between components that are within the same parent container (see Figure 22 for an example). Moreover, generated component diagrams do not show external components/containers that reside in the input system's source code, and neither container nor component diagrams show any relationships with external systems (e.g., calls to API endpoints that do not reside in the input system's source code). A perfect design would also capture these dependencies and give the end-user the option to include such relationships in generated diagrams. As can be read in Section 5.6, some pragmatic evaluation stakeholders expect such options.

Our chosen design simplifies the identification process of relationships, but comes at the cost of the completeness of the generated diagrams. In an IT consulting context, stakeholders may be particularly interested in external dependencies, e.g., because of security or governance constraints. This limits the usefulness of C4Gen in its current state. The diagrams that were generated using `Claude Code` do show such relationships, which indicates that the exclusion of such relationships is a limitation of C4Gen, and not a general limitation of AI-driven tools.

Component technologies are not included in component diagrams

As can be seen in the C4Gen-generated container diagrams (e.g., Figure 18), container nodes specify which technologies (programming languages/frameworks) are used in a container. However, generated component diagrams (e.g., Figure 22) do not show these technologies. We refrained from implementing such logic in our component identification logic to prevent the component identification agent from becoming too complex (in our current design, identifying components is a more complex task than identifying containers). However, in hindsight, as can be read in Section 5.6, some pragmatic evaluation stakeholders expect these technologies to be shown explicitly. The `Claude Code`-generated component diagrams include component technologies, which indicates that this is a C4Gen-specific limitation.

By not including component technologies in generated component diagrams, the usefulness of these diagrams is impacted directly when users are particularly interested in the technologies that are used within lower-level parts in a software system. This limitation is the result of design decisions during development, and only came to light after pragmatic evaluation of the solution. This limitation is the easiest to resolve, e.g., by extending the prompt of the container identification agent.

Lack of different architectural viewpoints in diagrams

In Section 2.1.1, we explained the ISO formalization of an Architecture Descriptions (AD). This formalization specifies that different stakeholders have different architectural viewpoints and concerns, and thus expect different visualizations of software architectures. In its current state, C4Gen only produces a single, technical-centric viewpoint of an input system, which might resonate more with the architectural viewpoint of technical-oriented system stakeholders. As mentioned in Section 5.4, we observe that some semantic evaluation stakeholders resonate less with generated diagrams than other stakeholders of the same input system, especially when professional roles are different. Moreover, the results show that different individuals classify elements in generated diagrams differently, even though they are evaluating the same set of diagrams. This indicates that there is no fixed consensus on what elements should belong in the architectural diagrams of a specific software system.

These observations are consistent with the findings of Pex [53], which we mentioned in Section 2.7.2, who observed that the quality of AI-generated software explanations is subjective and dependent on the expectations of individuals. The absence of options to steer C4Gen's output to match an individual's expectations limits the pragmatic applicability of the tool in an IT consulting context.

Lack of automation possibilities

Even though C4Gen partially automates the process of extracting architectural insights and representing these in C4 diagrams, the tool (in its current state) cannot be integrated into automation pipelines. The current design forces users to upload archived source code repositories using C4Gen's frontend, and users must manually validate generated insights after every step. These human-in-the-loop enforcements mitigate the risk of inclusion of inaccurate information in generated diagrams, but come at the cost of not being able to integrate the tool into version control systems, build pipelines, or other software delivery pipelines. Consequently, the generated diagrams only illustrate a snapshot of the current state of an input system. These limitations make C4Gen less applicable in a software development context, in which continuous integration is more important than in technical due diligence or IT audit operations.

Additionally, the final output can only be exported in markdown format. As observed during pragmatic evaluation (Section 5.6), some stakeholders expect different output formats, e.g., PDF or ODT. Such concerns reflect that, also in the context of technical due diligence, C4Gen should be interoperable with other file formats, before it can fully provide value.

6.3 Applicability of C4Gen in IT consulting

In this section, we discuss the applicability of C4Gen in an IT consulting context by interpreting the results of our pragmatic evaluation that are given in Section 5.6, and combine these findings with the semantic findings discussed in Sections 5.4 and 6.1. We structure the discussion using the dimensions of the Technology Acceptance Model, which we mentioned in Section 3.3. Based on these dimensions, we discuss how the strengths and weaknesses of our solution affect its applicability in an IT consulting context and briefly reflect on its applicability in other contexts.

Perceived usefulness

As indicated in our pragmatic evaluation, professionals in the IT consulting industry generally perceive C4Gen as useful. Most professionals indicate that the diagrams that are generated by the tool provide a valid architectural view of the input system. This aligns with the semantic evaluation results. Additionally, a majority of professionals mention that their operational effectiveness could be improved by using the tool, and productivity could be increased. However, professionals mention that C4Gen is primarily useful during the early-stage analysis of existing software repositories, for example, in a technical due diligence engagement. Limitations such as missing relationships and inconsistencies in the generated output impact the usefulness of the tool, as the output must be validated before it can be used. As a result, professionals mention that they see C4Gen as a supporting tool instead of a replacement for manual software analysis.

Perceived ease of use

All pragmatic evaluation subjects consider C4Gen easy to use. Some subjects mention that a number of minor UI/UX improvements are needed to improve the usability of the tool, including clearer guidance, improved navigation, better error handling, and a minor reconsideration of the tool's layout.

Attitude towards use

The overall attitudes of the pragmatic evaluation subjects towards C4Gen are positive. They find value in the tool's ability to automatically produce architectural visualizations based on source code, which reduces manual effort. However, there are concerns about inconsistencies and hallucinations, which lead subjects to treat the tool's output with caution. These hallucinations and inconsistencies are also reflected in our semantic evaluation and consistency study results. This indicates that C4Gen should be used as a supporting tool that can be used for architecture exploration and the drafting of architectural artifacts, and cannot be used as a fully reliable tool without human-in-the-loop intervention.

Behavioral intention

As mentioned in Section 5.6, all five pragmatic evaluation subjects indicate an intention to use C4Gen, with three of them indicating a definitive intention. Possible use cases that are described by the subjects are consistent: quickly generating architectural overviews and supporting technical due diligence tasks. This indicates that C4Gen aligns with consulting activities, in which speed and high-level overviews are important.

Practical feasibility and organizational implications

C4Gen is a lightweight tool that can be run locally on ordinary devices. The tool's source code is Dockerized and can be run in a Linux environment. As seen in Table 5.17, token usage and associated costs are manageable. The tool does require an LLM API to be connected to, e.g., the OpenAI API or an Azure deployment of an LLM. This makes the tool technically feasible in bigger and smaller organizations, as it can be deployed with relatively low operational effort.

An important practical consideration concerns how organizations use LLM-driven tools, and what their AI policies are. As discussed in Section 6.1, `Claude Code` achieves high performance. However, such a fully autonomous agentic solution relies on outsourcing architectural reasoning to external systems, which might introduce data governance concerns and an excessive dependency on a third-party provider. In organizations with mature governance structures, such concerns can limit the adoption of systems like `Claude Code`.

C4Gen positions itself differently. Whereas `Claude Code` has full autonomy, C4Gen uses a human-in-the-loop mechanism in which intermediate results are validated by humans. Moreover, the tool can be hosted on local devices and can be connected to enterprise-managed LLM endpoints like KPMG `Workbench`. This allows organizations to retain control over data flows and reasoning. These factors make C4Gen more suitable in contexts where transparency and compliance are important.

However, the mandatory human validation steps limit automation opportunities and position C4Gen as a tool that is suitable for incidental analysis activities, instead of as a tool that can be integrated into continuous workflows. This highlights that the tool is more feasible in an IT consulting context than in a software development context, because the tool cannot be used to reliably generate SAD in a CI/CD pipeline.

Furthermore, respondents mention limitations that impact the applicability of the tool, including missing relationships, limited export options, a lack of advanced diagram interaction, and version control. These limitations reduce the applicability of C4Gen in more complex engagements.

6.4 Threats to validity

In this section, we enumerate the most important threats to the validity of our research.

Single runs during evaluation

During syntactic and semantic evaluation of C4Gen's output, only one set of diagrams was evaluated per system because diagrams were only generated once. We chose to generate diagrams only once because of the amount of manual effort that is needed to evaluate diagrams. As mentioned in Section 4.1.2, a temperature of 0.1 was used in our implementations, which introduces stochasticity in the POC output. As mentioned in Section 2.6, this stochasticity increases in agentic solutions because of an ensemble effect. To rule out whether this ensemble effect has a significant effect on the quality of our POC output, multiple sets of diagrams should have been generated and evaluated for all input projects, which increases internal validity of the research. To partially counteract this threat, we included the consistency study in our methodology, which indicated the consistency of our POC output.

Directory truncation and prompt length limits

As mentioned in Chapter 4, we implemented context limits in the prompt building logic of C4Gen, such as directory truncation in the *tree* algorithm, file truncation, and maximum prompt and response lengths. We implemented these limits to save on token usage and runtime, and because we noticed that the quality of the POC dropped for prompts with high context. Regardless, by truncating directories, files, and prompts, essential information about the input project could be omitted, which could result in the agents misclassifying containers, components, or relationships. This is a threat to internal validity, because misclassifications might not be a result of LLMs not being effective, but rather from missing input data. To rule out the effect of truncation, an A/B experiment could have been conducted in which POC output is evaluated with and without the use of truncation. We assume that the impact of these truncations is not significant, because we put important information at the beginning of prompts (as described in Section 4.1.2).

Stakeholder and project pool sizes

A limited stakeholder pool size was used during semantic evaluation (nine participants, eleven evaluations) and pragmatic evaluation (five participants). With such a small pool size, individual biases can have a strong influence on the average, which directly influences internal validity. Likewise, we only evaluated a total of four input projects. This is a threat to external validity because a pool size of four input projects does not represent a wide variety of architecture types, technology stacks, and domains. By conducting research on a wider project pool, it can be investigated how C4Gen performs on systems with different properties (programming language, small versus large, consistent versus inconsistent architecture, standard versus niche architecture). By conducting research on a wider stakeholder pool, the influence of stakeholder preferences and expectations can be weakened.

Evaluation responsibilities

A threat to internal validity arises because the baseline and top-line diagrams were evaluated by the authors, while C4Gen's semantic evaluation also includes viewpoints from external stakeholders. This introduces potential rater bias, which can affect the comparability of semantic scores. We mitigate this risk by interpreting the baseline and top-line scores primarily as reference bounds instead of as strict acceptability bounds.

Agentic threats to validity

As mentioned above and in Section 2.6, agentic solutions can propagate and magnify errors due to an ensemble effect. This effect may cause another threat to internal validity in C4Gen, because of a summary-of-summary approach in the container summarization agent implementation. Containers are summarized using their child components' summaries, rather than providing the container's source code directly to the agent. Because of this bottom-up approach, errors that are introduced by the component summarization agent may be propagated and magnified by the container summarization agent, which makes it difficult to evaluate where and why errors are introduced in the pipeline.

Moreover, the pipeline's agents are not evaluated individually. To thoroughly evaluate the performance of C4Gen and identify potential bottlenecks, individual agents of the POC's sequential pipeline could have been evaluated in an agent ablation study. In this study, the influence of individual agents on the

POC output should be analyzed. This would increase the internal validity of the research because errors can be diagnosed and attributed to specific steps in the pipeline.

Additionally, we did not investigate the human-in-the-loop effect in our research. The diagrams that were generated for syntactic and semantic evaluation were the results of running the pipeline without human intervention, which is a threat to internal validity. To thoroughly conclude whether C4Gen could be used as a human-in-the-loop framework for automatic SAD, the quality of generated diagrams could additionally have been evaluated with the inclusion of human interventions between agentic steps.

Single LLM with fixed hyperparameters

As mentioned in Section 4.2, we only evaluated the performance of C4Gen using one LLM with fixed hyperparameters. This is a threat to external validity, because we cannot conclude whether our solution is effective using LLMs overall. To rule this out, a study could have been conducted in which C4Gen is evaluated using different LLMs and configurations. Only if this is done can it be concluded in general whether LLMs can be used for automatic SAD creation.

Abacus semantic evaluation omissions

For the Abacus input system, only the container overview diagram and one component diagram (frontend) were evaluated semantically. We have chosen to evaluate only one component diagram to encourage external stakeholders to help with evaluation, without overloading them with manual work. This limitation is a threat to external validity, as we cannot generalize C4Gen's performance in creating component diagrams. In an ideal world, Abacus was evaluated semantically using more generated component diagrams.

Limitations and assumptions of used metrics

The metrics that we have defined in Section 3.3 pose several threats to construct validity.

Firstly, we used the heuristic acceptability thresholds $S_{\text{normalized}} \geq 0.75$ and $hallu\% \leq 10\%$. These thresholds have been reverse-engineered using the survey responses in Appendix C, and should be seen as guidelines rather than strict boundaries. The lack of more formal boundaries poses a threat to construct validity.

Secondly, we calculate the overall $S_{\text{normalized}}$ score for a generated diagram by taking the mean of the diagram's $S_{\text{normalized}}$ scores for individual element types in the diagram. By following this method, all element types have an equal share in the overall score of the diagram. This is based on the assumption that containers, components, container-to-container relationships, and component-to-component relationships all share an equal share in the semantic quality of generated diagrams. In reality, however, these weights might differ and are based on subjective opinions by stakeholders. This poses a risk to the construct validity of our research.

Another threat to construct validity is posed by the classification type descriptions that we introduced in the scoring rubrics in Tables 3.1 and 3.2. These scoring rubrics force respondents to classify individual elements according to these descriptions and do not take into account (technical) nuances that might be present around these classifications. For example, a generated element could be hallucinated in various ways (e.g., incorrect name, incorrect technology, incorrect label, incorrect source/target, descriptions that contain partially false information). By classifying all such errors under the same *hallucination* type, valuable insights might be lost. Similarly, generated elements can only be classified as a *match* or a *valid*, even though in reality a generated element may lie somewhere in between.

Lastly, our introduced *hallu*% metric assumes that all types of hallucinations have an equal share in the semantic quality of a diagram. The metric divides the number of hallucinations by the number of elements to calculate a percentage. In reality, however, some hallucinations might be destructive for the semantic value of a diagram (e.g., adding a container that does not exist) whilst others might be rather unimportant (e.g., adding inaccurate information to a container). By assuming that all hallucinations are equal, a threat to construct validity is introduced.

6.5 Reflection: mechanisms for AI-driven architecture recovery

In this research, we investigated multiple mechanisms for extracting and structuring software architecture from source code using LLMs. C4Gen uses a top-down identification mechanism, a bottom-up summarization mechanism, agentic decomposition, and human-in-the-loop validation. Additionally, our baseline methodology incorporates a zero-shot prompting strategy, and our top-line methodology incorporates the RCI-mechanism and fully autonomous agentic planning and reasoning. In this research, indicative insights are gained about the strengths and weaknesses of these mechanisms.

Our research indicatively shows that a top-down mechanism appears to be especially useful for recovering the high-level structural decomposition of software systems, as it can identify containers and container–container relationships in a relatively stable manner. However, the high-level abstraction level also appears to be a limitation, as more complex, nuanced relationships are not always captured sufficiently. In contrast, the bottom-up summarization mechanism appears to be more useful for grounding ADs in implementation details and for generating detailed component–component relationship insights. However, because this mechanism depends on sequential summarization (*summary-of-summaries* approach), it also seems more sensitive to variability.

Additionally, the results provide indicative insights into the strengths and weaknesses of orchestration and prompting mechanisms. C4Gen incorporates sequential agentic decomposition to structure the recovery process into smaller and more traceable sub-tasks, thereby avoiding a single black-box generation step. The decomposition into a sequential pipeline improves observability, but comes at the cost of error propagation if inaccuracies are not corrected in time. In this context, human-in-the-loop validation appears to be an important mechanism for maintaining traceability and mitigating hallucinations, although it simultaneously limits the degree of automation that can be achieved in the pipeline.

The baseline methodology indicates that zero-shot prompting might be useful as a simple starting point with little to no development costs, but in our experimental setup it did not provide reliable output for architecture recovery tasks. In contrast, the combination of a fully autonomous agentic solution with RCI prompting appears to improve performance in architecture recovery, but this comes at the cost of reduced transparency and reduced user control over intermediate judgments.

Taken together, these findings suggest that there is no one-size-fits-all mechanism for AI-driven architecture recovery. The suitability of a mechanism appears to depend on the specific use case and on the trade-offs between output quality, autonomy, traceability, observability, and user control.

6.6 Reflection: software architecture in the age of agentic AI

When we move beyond the specific context of the C4 model and reflect more broadly on agentic AI in software architecture, our findings indicate that LLM-driven approaches enable the automatic extraction and structuring of architectural insights. These approaches reshape how architectural knowledge is constructed, interpreted, and communicated.

In conventional, human-driven approaches, software architecture is not an objective representation that is discovered by humans, but rather an abstraction that is constructed to support communication among stakeholders. ADs are compressions of a system's underlying implementation, guided by specific concerns and viewpoints (as reflected in the ISO 42010 standard). From this perspective, the outputs of agentic approaches such as C4Gen or `Claude Code` 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 incomplete signals (available input data) and agentic reasoning.

This is reflected in the variability that we observe in C4Gen's output. Different runs of the tool result in different architectural representations of the same input system. This variability can be explained as alternative generations of architectural abstractions that are formed by agentic stochastic reasoning, comparable to different viewpoints. In this way, agentic AI does not discover a single architecture, but produces multiple candidate representations that should be validated by a stakeholder.

This shift has implications for the role of a software architect. 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 an architect may shift towards a role that is more focused on the evaluation and refinement of AI-generated artifacts.

From this perspective, architecture recovery becomes a more dialogical process, in which architects iteratively interact with agentic systems to explore and refine architectural views. Human-in-the-loop mechanisms become essential, especially from the perspective of AI transparency. With these new roles, agentic AI reshapes software architecture in a field in which humans collaborate with AI, rather than a field in which humans are fully replaced by AI.

Chapter 7

Conclusions

This research aimed to develop a framework that enables the reliable use of AI-driven source code analysis techniques for generating structured and traceable C4 model representations of software systems in an IT consulting context. To achieve this goal, we designed and developed the human-in-the-loop agentic tool C4Gen and evaluated it on multiple real-world software systems using syntactic, semantic, and pragmatic quality metrics. By comparing C4Gen's output against a baseline and a state-of-the-art top-line approach, we assessed the performance and applicability of C4Gen. The results from these evaluations were interpreted to gain insights into the strengths and limitations of the proposed solution.

In this chapter, we provide brief answers to the research questions and propose possible directions for future work.

7.1 Answers to the research questions

SQ1. *"What raw data can be extracted from a software system by Large Language Models to establish a C4-level understanding of the system?"*

The results of our research indicate that LLMs can extract a variety of raw data from software repositories, which can be used to establish C4-level insights. Our proposed solution demonstrates that directory structures and the contents of high-level service definition or configuration files (e.g., Docker or infrastructure files) can be provided to LLMs as context, which can subsequently be used by LLMs to identify containers, technologies, and container–container relationships. On the component-level, our solution demonstrates that directory structures and parent-container information can be used to identify components, whilst summaries of actual source code are needed to reliably extract low-level component–component relationships. Additionally, our research shows that LLMs can retrieve file paths that link architectural views to underlying implementation (e.g., linking a backend container to corresponding implementation files). The extracted data can be structural (containers/components) or relational. Together, these data sources enable the establishment of container- and component-level insights, which can be used for generating diagrams in at least two of the four C4 abstraction levels.

SQ2. *"How can extracted data be structured into an intermediate and traceable representation that links architectural concepts to specific source code locations?"*

Our research demonstrates that extracted data can be structured into an intermediate representation that links architectural elements to source code artifacts, such as file paths and directory structures. In C4Gen, containers, components, and relationships are structured objects with properties such as identifiers, descriptions, technologies, associated file paths, and foreign keys (e.g., a key of a parent container or child component, or a relationship with source and target keys). This representation is expressed in a machine-readable format (JSON), which ensures interoperability across the entire pipeline. Traceability is ensured by providing explicit references between architectural concepts and source code locations. Additionally, agents provide their reasoning process to the user, and the user can validate intermediate results after each step, which increases traceability across the pipeline.

SQ3. *"How can the automatic process of creating a structured and intermediate representation of architectural information be adapted into a reliable human-in-the-loop framework?"*

A human-in-the-loop framework can be created for the automatic process of establishing C4-level insights by decomposing the process into sequential agentic steps and enforcing human validation after each agentic step. In C4Gen, we did this by implementing a sequential pipeline of five AI agents and explicitly giving the user the option to validate and refine generated outputs. Deterministic methods are used for diagram rendering and SAD generation, which ensures that LLM-driven stochastic reasoning is separated from output formatting. This separation ensures that LLMs can only introduce semantic errors, and not syntactic rendering errors. These design decisions mitigate risks of hallucinations and error propagation in agentic systems. As a result, our solution combines LLM-driven automation with a human-in-the-loop mechanism, which enables reliable use in an IT consulting context.

SQ4. *"To what extent does the proposed human-in-the-loop framework improve a consultant's workflow of creating architectural documentation?"*

The results of the pragmatic evaluation of our solution indicate that the solution improves a consultant's workflow primarily by reducing the initial need for manual source code navigation, which speeds up the process of gaining architectural insights. The rapid generation of container- and component-level overviews supports analysis tasks in technical due diligence engagements. The human-in-the-loop mechanism ensures that generated output can be validated and corrected, which increases trust and accuracy in the solution's final output. Evaluation results indicate that IT consultants perceive C4Gen as useful, easy to use, and beneficial for productivity and effectiveness. However, inconsistencies at lower abstraction levels, missing relationships, UI/UX improvement areas, and missing functionalities limit the applicability of the tool in more complex engagements. The framework is perceived as a supportive tool that improves a consultant's workflow, but does not replace it.

RQ. *"How can a framework be developed that uses AI-driven analysis techniques to generate a structured and traceable C4 model representation of architectural structure, that supports consultants in producing reliable software architecture documentation?"*

Our research shows that a framework can be developed by combining LLM-driven source code analysis with a sequential, multi-phase, human-in-the-loop pipeline that produces intermediate representations. The proposed solution, C4Gen, combines top-down identification of containers and components with bottom-up summarization of source code to capture architectural insights. Human validation is embedded after each agentic step, which increases the transparency and traceability of the solution. The use of deterministic rendering separates semantic reasoning from diagram generation, which mitigates syntactic errors. Evaluation results show that the solution achieves low syntactic error rates and acceptable semantic quality, comparable to a state-of-the-art fully autonomous agentic system. C4Gen indicates that a reliable framework can be created by combining agentic AI with human insights, which enables the production of reliable SAD.

7.2 Future work

Cross-repository analysis and C4 system context diagram generation

As mentioned in Section 5.4, semantic evaluation respondents raised the question of how C4Gen could be used for cross-repository C4 analysis. The design of C4Gen requires the user to upload an input project as an archived repository file, but in practice, systems do not always follow a mono-repository structure. Future research could address this limitation and investigate how our design could be changed or extended to allow for more complex input projects.

Mutually, future research could investigate how the solution could be extended to also include the generation of C4 system context diagrams. As we mentioned in Section 6.2, the C4 model is composed of four abstract hierarchical levels, but our design only generates diagrams for the two levels that describe a single system's static structure (container and component level). The other two levels are not included in the output of our solution. Code diagrams are not included because of their limitations and infrequent use. System context diagrams were left out of this research's scope, but are particularly interesting for future research because they could be used to address the questions that are raised by our semantic evaluation stakeholders. Future research could investigate how our design can be extended to take into account an input's system position within an environment with external dependencies, and generate system context diagrams accordingly.

Dynamic C4 diagrams

Moreover, our current solution only analyzes the static structure of an input system via source code analysis and generates static C4 diagrams. Dynamic behavior is not analyzed. Future research could investigate how our design could be changed or extended to allow for behavioral analysis of running software systems, rather than static analysis of source code. In addition, our design could be extended to also generate behavioral C4 diagrams (dynamic and deployment diagrams).

External dependency scanning

C4Gen's current state only enables the extraction of internal component–component relationships (relationships between components inside the same parent container). External component calls or external system calls are not integrated. Our top-line study shows that it is possible for agentic systems to include such external dependencies in generated diagrams. Future research could investigate how our design can be improved such that external dependencies are included, just like in `Claude Code`'s output, whilst the human-in-the-loop mechanism of C4Gen is still maintained.

Feedback mechanism or custom instructions

Our evaluation results show that different stakeholders expect different diagrams as an output. Additionally, our current solution makes mistakes that can only be resolved by manually changing generated data. Future research could investigate how reinforcement learning-related concepts could be integrated into our design, e.g., by integrating a feedback system that users can use to fine-tune the solution's behavior. For example, one user might prefer many component-to-component relationships in generated component diagrams, whilst others might only expect the most important relationships. Likewise, one user might always expect the inclusion of databases, whilst others do not. By integrating a feedback system or a way of adding custom instructions to the agents, the solution could be instructed to behave according to specific provided guidelines or feedback.

LLM comparison and hyperparameter study

In this research, our solution has only been evaluated using one LLM and one set of hyperparameters (given in Section 4.1). Future research could investigate the syntactic and semantic quality of C4Gen's output and its applicability in an IT consulting context using different LLM and hyperparameter configurations.

Human-in-the-loop Claude Code solution

Our top-line study shows that `Claude Code` generates C4 diagrams with no syntactic errors and with a better semantic quality than C4Gen-generated diagrams. However, this comes at the cost of little observability and no human-in-the-loop mechanisms. Future work could investigate how a solution could be designed that uses `Claude Code` as the main engine for identifying architectural insights, but that also integrates a human-in-the-loop mechanism similar to that in C4Gen, such that the `Claude Code`-based solution can be used reliably in organizations with mature governance and risk-averse AI policies.

Bibliography

[1] Mohamad Abou Ali, Fadi Dornaika, and Jinan Charafeddine. Agentic ai: a comprehensive survey of architectures, applications, and future directions. *Artificial Intelligence Review*, 59(1), November 2025. ISSN 1573-7462. doi: 10.1007/s10462-025-11422-4. URL http://dx.doi.org/10.1007/s10462-025-11422-4.

[2] Domenico Amalfitano, Marco De Luca, Tiziano Santilli, Patrizio Pelliccione, and Anna Rita Fasolino. Automated software architecture design recovery from source code using llms. In *Software Architecture: 19th European Conference, ECSA 2025, Limassol, Cyprus, September 15–19, 2025, Proceedings*, page 73–89, Berlin, Heidelberg, 2025. Springer-Verlag. ISBN 978-3-032-02137-3. doi: 10.1007/978-3-032-02138-0_5. URL https://doi.org/10.1007/978-3-032-02138-0_5.

[3] Anthropic. Claude code. https://claude.com/product/claude-code, 2026. Accessed: 2026-05-13.

[4] Menaka Pushpa Arthur. Automatic source code documentation using code summarization technique of nlp. *Procedia Computer Science*, 171:2522–2531, 2020. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2020.04.273. URL https://www.sciencedirect.com/science/article/pii/S1877050920312655. Third International Conference on Computing and Network Communications (CoCoNet'19).

[5] Microsoft Azure. Azure openai service pricing. https://azure.microsoft.com/en-us/pricing/details/azure-openai/, 2026. Accessed: 2026-03-25. Pricing for GPT-4o-2024-08-06 (Global): $2.50 per 1M input tokens, $10.00 per 1M output tokens.

[6] Liudmila Bagdoniene and Rasa Jakstaite. Trust as basis for development of relationships between professional service providers and their clients. *Economics and management*, (14):360–366, 2009.

[7] Simon Brown. The c4 model for software architecture. *InfoQ*, June 2018. URL https://www.infoq.com/articles/C4-architecture-model/. Reviewed by Thomas Betts.

[8] Simon Brown. C4 Model, 2026. URL https://c4model.com/. Accessed: 2026-01-07.

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

[10] Julieth Castellanos Ardila, Barbara Gallina, and Faiz UL Muram. Compliance checking of software processes: A systematic literature review. *Journal of Software: Evolution and Process*, 34, 03 2022. doi: 10.1002/smr.2440.

[11] Vikas Chomal and Jatinderkumar Saini. Significance of software documentation in software development process. *International Journal of Engineering Innovation & Research*, 3:410–416, 07 2014.

[12] Saaniya Chugh and Aditya Vilas Deshpande. Opportunities and challenges of agentic ai in finance. *Journal of Emerging Technologies and Innovative Research*, February 2025. doi: 10.2139/ssrn.5538799. URL https://dx.doi.org/10.2139/ssrn.5538799. Accessed: 2026-01-13.

[13] CISA. Software bill of materials (sbom). Cybersecurity and Infrastructure Security Agency (CISA) official website, 2025. URL https://www.cisa.gov/sbom. Accessed: 2025-12-22; provides an overview of SBOM concepts, resources, and guidance on software component transparency and supply chain security efforts by CISA.

[14] Consultancy.uk. Big four fined more than £100 million by frc over five years. *Consultancy.uk*, June 2025. URL https://www.consultancy.uk/news/40601/big-four-fined-more-than-100-million-by-frc-over-five-years. Accessed: 2026-01-12.

[15] Al Danial. cloc: Count lines of code. https://github.com/AlDanial/cloc, 2026. GitHub release v2.08, accessed April 20, 2026.

[16] Fred Davis. User acceptance of information systems : the technology acceptance model (tam). 01 1987.

[17] Rudresh Dwivedi, Devam Dave, Het Naik, Smiti Singhal, Rana Omer, Pankesh Patel, Bin Qian, Zhenyu Wen, Tejal Shah, Graham Morgan, and Rajiv Ranjan. Explainable ai (xai): Core ideas, techniques, and solutions. *ACM Comput. Surv.*, 55(9), January 2023. ISSN 0360-0300. doi: 10.1145/3561048. URL https://doi.org/10.1145/3561048.

[18] Matteo Esposito, Xiaozhou Li, Sergio Moreschini, Noman Ahmad, Tomas Cerny, Karthik Vaidhyanathan, Valentina Lenarduzzi, and Davide Taibi. Generative ai for software architecture. applications, challenges, and future directions, 2025. URL https://arxiv.org/abs/2503.13310.

[19] EU. Regulation (eu) 2024/2847 of the european parliament and of the council of 23 october 2024 on horizontal cybersecurity requirements for products with digital elements and amending regulations (eu) no. 168/2013 and (eu) 2019/1020 and directive (eu) 2020/1828 (cyber resilience act). Official Journal of the European Union, L 2847, 20 November 2024, pp. 1–(end), November 2024. URL https://eur-lex.europa.eu/eli/reg/2024/2847/oj/eng. Accessed: 2025-12-22.

[20] European Commission. Cyber Resilience Act: Shaping Europe's digital future. https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act, 2026. Accessed 8 January 2026.

[21] European Commission. Cyber Resilience Act – Conformity Assessment. https://digital-strategy.ec.europa.eu/en/policies/cra-conformity-assessment, 2026. Accessed 8 January 2026.

[22] European Commission. Cyber Resilience Act – Manufacturers. https://digital-strategy.ec.europa.eu/en/policies/cra-manufacturers, 2026. Accessed 8 January 2026.

[23] European Commission. Cyber Resilience Act – Reporting. https://digital-strategy.ec.europa.eu/en/policies/cra-reporting, 2026. Accessed 8 January 2026.

[24] European Commission High-Level Expert Group on AI. Ethics Guidelines for Trustworthy AI. https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai, 2019. Accessed 9 January 2026.

[25] Hugh Evans. How to use ai to write documentation that actually works. https://hughevans.dev/how-to-use-ai-to-write-documentation-that-actually-works/, February 2025. Accessed: 2026-01-14.

[26] Stephen D. Gantz. Chapter 1 - it audit fundamentals. In Stephen D. Gantz, editor, *The Basics of IT Audit*, pages 1–19. Syngress, Boston, 2014. ISBN 978-0-12-417159-6. doi: https://doi.org/10.1016/B978-0-12-417159-6.00001-8. URL https://www.sciencedirect.com/science/article/pii/B9780124171596000018.

[27] R. Stuart Geiger, Nelle Varoquaux, Charlotte Mazel-Cabasse, and Chris Holdgraf. The types, roles, and practices of documentation in data analytics open source software libraries: A collaborative ethnography of documentation work. *Computer Supported Cooperative Work (CSCW)*, 27(6):767–802, 2018. doi: 10.1007/s10606-018-9333-1.

[28] GitHub, Inc. Github copilot. https://github.com/features/copilot, 2026. Accessed: 2026-01-14.

[29] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models?, 2024. URL https://arxiv.org/abs/2404.06654.

[30] Brian Hu, Paul Tunison, Bhavan Vasu, Nitesh Menon, Roddy Collins, and Anthony Hoogs. Xaitk: The explainable ai toolkit. *Applied AI Letters*, 2(4):e40, 2021. doi: https://doi.org/10.1002/ail2.40. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/ail2.40.

[31] W. Irwin, C. Cook, and N. Churcher. Parsing and semantic modelling for software engineering applications. In *2005 Australian Software Engineering Conference*, pages 180–189, 2005. doi: 10.1109/ASWEC.2005.36.

[32] ISO/IEC/IEEE. ISO/IEC/IEEE 42010:2022 – software, systems and enterprise — architecture description. International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), Institute of Electrical and Electronics Engineers (IEEE), November 2022. URL https://www.iso.org/standard/74393.html. Edition 2, Standard number: ISO/IEC/IEEE 42010:2022.

[33] Anton Jansen, Paris Avgeriou, and Jan Salvador van der Ven. Enriching software architecture documentation. *Journal of Systems and Software*, 82(8):1232–1248, 2009. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2009.04.052. URL https://www.sciencedirect.com/science/article/pii/S0164121209001046. SI: Architectural Decisions and Rationale.

[34] Nils Johansson, Mauro Caporuscio, and Tobias Olsson. Mapping source code to software architecture by leveraging large language models. In *Software Architecture. ECSA 2024 Tracks and Workshops: Luxembourg City, Luxembourg, September 3–6, 2024, Proceedings*, page 133–149, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-70945-6. doi: 10.1007/978-3-031-71246-3_13. URL https://doi.org/10.1007/978-3-031-71246-3_13.

[35] Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agents that matter, 2024. URL https://arxiv.org/abs/2407.01502.

[36] Kiesraad. Abacus c4 diagrams. https://github.com/kiesraad/abacus/blob/main/documentatie/softwarearchitectuur/diagrammen, 2025. Image from the Abacus software documentation.

[37] Kiesraad. Abacus. https://kiesraad.github.io/abacus-documentatie/, 2026. Accessed: 2026-01-15.

[38] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks, 2023. URL https://arxiv.org/abs/2303.17491.

[39] Philippe Kruchten. Architectural blueprints—the "4+1" view model of software architecture. In *Proceedings of the IEEE Software*, pages 42–50, 1995. URL https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf.

[40] LIACS. AI4MDE v1.0.2. https://github.com/ai4mde/studio, April 2025.

[41] Odd Lindland, Guttorm Sindre, and Arne Sølvberg. Understanding quality in conceptual modeling. *IEEE Software*, 11:42–49, 01 1994.

[42] Llama Project. Llama 4 Models. https://www.llama.com/models/llama-4/, 2026. Accessed 8 January 2026.

[43] Nishchai Jayanna Manjula and Akhilesh Dube. Harnessing generative ai to create and understand architecture diagrams. *International Journal of Scientific Research and Applications*, 13(2):2601, 2024. doi: 10.30574/ijsra.2024.13.2.2601. URL https://doi.org/10.30574/ijsra.2024.13.2.2601.

[44] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. Prompt engineering in large language models. In I. Jeena Jacob, Selwyn Piramuthu, and Przemyslaw Falkowski-Gilski, editors, *Data Intelligence and Cognitive Informatics*, pages 387–402, Singapore, 2024. Springer Nature Singapore. ISBN 978-981-99-7962-2.

[45] Nature Machine Intelligence Editorial Team. Multi-agent ai systems need transparency. *Nature Machine Intelligence*, 8(1):1–1, 2026. ISSN 2522-5839. doi: 10.1038/s42256-026-01183-2. URL https://doi.org/10.1038/s42256-026-01183-2.

[46] Object Management Group. OMG Unified Modeling Language—Version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF, December 2017. URL https://www.omg.org/spec/UML/2.5.1/PDF. Formal specification; normative PDF.

[47] OpenAI. How ChatGPT and Our Foundation Models Are Developed. https://help.openai.com/en/articles/7842364-how-chatgpt-and-our-foundation-models-are-developed, 2025. Accessed 9 January 2026.

[48] OpenAI. Introducing GPT-5.2. https://openai.com/index/introducing-gpt-5-2/, December 2025. Accessed 9 January 2026.

[49] Oracle. *Java Platform, Standard Edition Javadoc Guide, Release 25*. Oracle, September 2025. URL https://docs.oracle.com/en/java/javase/25/javadoc/index.html. Accessed: 2026-01-14.

[50] Oracle Corporation. Javadoc tool. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html, n.d. Accessed: 2026-01-12.

[51] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24: 45–77, 01 2007.

[52] Matjaž Perc. Counterfeit judgments in large language models. *Proceedings of the National Academy of Sciences*, 122(48):e2528527122, 2025. doi: 10.1073/pnas.2528527122. URL https://www.pnas.org/doi/abs/10.1073/pnas.2528527122.

[53] Floris M. J. Pex. Exploring software explanation using retrieval augmented generation. Master's thesis, Eindhoven University of Technology, August 2024. URL https://research.tue.nl/en/studentTheses/exploring-software-explanation-using-retrieval-augmented-generati/.

[54] PlantUML. C4-plantuml. https://github.com/plantuml-stdlib/C4-PlantUML, 2026. Version v2.13.0.

[55] PlantUML. Plantuml server. https://github.com/plantuml/plantuml-server, 2026.

[56] Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm, 2021. URL https://arxiv.org/abs/2102.07350.

[57] Satrio Adi Rukmono, Lina Ochoa, and Michel 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)*, pages 7–12, 2023. doi: 10.1109/ICoDSE59534.2023.10292037.

[58] Satrio Adi Rukmono, Lina Ochoa, and Michel 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, page 92–96, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705007. doi: 10.1145/3639476.3639776. URL https://doi.org/10.1145/3639476.3639776.

[59] Johannes Schneider. Generative to agentic ai: Survey, conceptualization, and challenges, 2025. URL https://arxiv.org/abs/2504.18875.

[60] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences, 2019. URL https://arxiv.org/abs/1704.02685.

[61] Mohamed Soliman, Elia Ashraf, Kamel M. K. Abdelsalam, Jan Keim, and Ashwin Prasad Shivarpatna Venkatesh. Llms for software architecture knowledge: A comparative analysis among seven llms. In Vasilios Andrikopoulos, Cesare Pautasso, Nour Ali, Jacopo Soldani, and Xiwei Xu, editors, *Software Architecture*, pages 99–115, Cham, 2026. Springer Nature Switzerland. ISBN 978-3-032-02138-0.

[62] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source code summarization in the era of large language models, 2025. URL https://arxiv.org/abs/2407.07959.

[63] Swimm. Swimm: Application understanding platform. https://swimm.io/, 2026. Accessed January 14 2026.

[64] Kamil Szczepanik and Jarosław A. Chudziak. Collaborative llm agents for c4 software architecture design automation, 2025. URL https://arxiv.org/abs/2510.22787.

[65] Rayden Tseng. Exploring large language models in it audit: Improving efficiency in control testing. Master's thesis, Leiden University, Leiden Institute of Advanced Computer Science (LIACS), 2025.

[66] Alan Turing. *Computing Machinery and Intelligence (1950)*. 09 2004. ISBN 9780198250791. doi: 10.1093/oso/9780198250791.003.0017.

[67] Lorenzo Vainigli. Plantuml generator. https://github.com/lorenzovngl/plantuml-generator, 2022. GitHub repository.

[68] Sven Yannick van Dam. C4gen prompts, May 2026. URL https://doi.org/10.5281/zenodo.20391799.

[69] Dimitri van Heesch. Doxygen. https://www.doxygen.nl/index.html, n.d. Accessed: 2026-01-12.

[70] Rasita Vinay, Giovanni Spitale, Nikola Biller-Andorno, and Federico Germani. Emotional prompting amplifies disinformation generation in ai large language models. *Frontiers in Artificial Intelligence*, Volume 8 - 2025, 2025. ISSN 2624-8212. doi: 10.3389/frai.2025.1543603. URL https://www.frontiersin.org/journals/artificial-intelligence/articles/10.3389/frai.2025.1543603.

[71] Zhiyuan Wan, Yun Zhang, Xin Xia, Yi Jiang, and David Lo. Software architecture in practice: Challenges and opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1457–1469, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3616367. URL https://doi.org/10.1145/3611643.3616367.

[72] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

[73] Wware Consulting. Gituml. https://www.gituml.com/, n.d. Accessed: 2026-01-26.

[74] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. Hallucination is inevitable: An innate limitation of large language models, 2025. URL https://arxiv.org/abs/2401.11817.

[75] Dayu Yang, Antoine Simoulin, Xin Qian, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, and Grey Yang. Docagent: A multi-agent system for automated code documentation generation, 2025. URL https://arxiv.org/abs/2504.08725.

Appendix A

Prompts

Lengthy parts of prompts are omitted for brevity. Full prompts can be found on Zenodo. [68]

A.1 Container information file paths identifier prompt

```
Based on the following directory structure of a software project (shown in tree command
format below), identify the file paths that are most likely to contain information about
the system's containers or services (C4 model level containers, e.g. backend, frontend, or
database).

Please return a list of file paths (relative to the repository root) that are most likely
to contain information on what containers/services the system is composed of. We will
analyze the files that you identified such that we will gain more information on the system
's containers. Examples of files that may be interesting for this analysis are:
[[FEW-SHOT EXAMPLES]

Note that input projects may not always explicitly define a database, because it could be a
 managed service (e.g., AWS RDS) or defined in infrastructure-as-code files (e.g.,
Terraform, CloudFormation). However, if you find files/directories (e.g. .sqlx) that
suggest the presence of a database container/service, please include them as well.

Important rules for your response structure:
[[FORMATTING RULES]]

Example response format:
<MOTIVATION>
[[ONE-SHOT EXAMPLE]]
</MOTIVATION>
<FILEPATHS>
[[ONE-SHOT EXAMPLE]]
</FILEPATHS>

Here is the directory structure for the software project:
<PROJECT_DIRECTORY_TREE>
[[INPUT DIRECTORY TREE]]
</PROJECT_DIRECTORY_TREE>
```

A.2 Container identifier prompt

```
Based on the files from a software project given below, identify and extract information
about the containers/services (C4-level containers) that make up the system. A container is
 a deployable unit (e.g., a backend, database, frontend, API, microservice, etc.).

For each container you identify, provide the following information:
- name: The name of the container/service
- type: The type of container (e.g. backend, database, frontend, API, microservice, etc.)
- programming_languages: List of programming languages used in the individual container.
Also include frameworks if you detect any, but only do this if you are certain. (e.g., ['
Python', 'JavaScript', 'Django', 'React'])
- description: A brief description of what this container does
- root_paths: List of root file paths in the repository where this container's code resides
 (e.g., ['root/services/user-service', 'root/backend'])

Important rules for your response:
[[FORMATTING AND CONTENT RULES]]

Note that the input may not explicitly define a database, because it could be a managed
service (e.g., AWS RDS) or defined in infrastructure-as-code files (e.g., Terraform,
CloudFormation). If you find traces that suggest the presence of a database (e.g. the
presence of migration or database schema files), do include a container for the database
and elaborate on the evidence you found in the motivation section, even if you cannot find
root paths for the database code.

Example response format for a container in a 3-tier system:
<EXAMPLE_RESPONSE>
<MOTIVATION>
[[ONE-SHOT EXAMPLE]]
</MOTIVATION>
<CONTAINERS>
[[ONE-SHOT EXAMPLE]]
</CONTAINERS>
</EXAMPLE_RESPONSE>

This is the directory tree of the repository (note that some directories may be truncated
to prevent forming very large trees):
<PROJECT_DIRECTORY_TREE>
{[[INPUT DIRECTORY TREE]]
</PROJECT_DIRECTORY_TREE>

Another agent has identified the following possible important file(s) in the repository
that may contain container information:
<FILES_CONTENT>
[[INPUT FILES CONTENT]]
</FILES_CONTENT>
```

A.3 Container relationship information file paths identifier prompt

```
You will be provided with the directory structure of a software project (shown in tree
command format below), along with a list of containers/services (C4 model level containers,
 e.g. backend, frontend, or database) that have been identified in the system.

Your task is to identify the file paths that are most likely to contain information about
the relationships and interactions between these containers. Examples of files that may be
interesting for this analysis are:
[[FEW-SHOT EXAMPLES]
```

```
Please return a list of file paths (relative to the repository root) that are most likely
to contain information on how the identified containers interact with each other.

Important rules for your response structure:
[[FORMATTING RULES]]

Example response format:
<FILEPATHS>
[[ONE-SHOT EXAMPLE]]
</FILEPATHS>

You need to do this for the following containers:
<CONTAINERS_INFO>
[[INPUT CONTAINERS INFORMATION]]
</CONTAINERS_INFO>

Directory structure of the repository:
<PROJECT_DIRECTORY_TREE>
[[INPUT DIRECTORY TREE]]
</PROJECT_DIRECTORY_TREE>

Only include the XML object in your response, no other text.
```

A.4 Container relationship identifier prompt

```
You will be provided with the directory structure of a software project (shown in tree
command format below), along with a list of containers/services (C4 model level containers,
 e.g. backend, frontend, or database) that have been identified in the system. Additionally
, you will receive the contents of files that are likely to contain information about the
relationships and interactions between these containers.
Your task is to identify and extract information about the relationships and interactions
between these containers. For each relationship you identify, provide the following
information:
- source_id: The ID of the source container/service in the relationship
- target_id: The ID of the target container/service in the relationship (this is the
container to which the arrowhead points)
- label: A brief label or name for the relationship (e.g., "REST API Call", "Database Query
", "Message Queue Communication")
- bidirectional: A boolean value indicating whether the relationship is bidirectional (true
) or unidirectional (false)

Important rules for your response:
[[FORMATTING AND CONTENT RULES]]

Example response format for relationships between containers:
<EXAMPLE_RESPONSE>
<MOTIVATION>
[[ONE-SHOT EXAMPLE]]
</MOTIVATION>
<RELATIONSHIPS>
[[ONE-SHOT EXAMPLE]]
</RELATIONSHIPS>
</EXAMPLE_RESPONSE>

You need to do this for the following containers:
<CONTAINERS_INFO>
[[INPUT CONTAINERS INFORMATION]]
```

```
</CONTAINERS_INFO>

Directory structure of the repository:
<PROJECT_DIRECTORY_TREE>
[[INPUT DIRECTORY TREE]]
</PROJECT_DIRECTORY_TREE>

Another agent has identified the following possible important file(s) in the repository
that may contain information about container relationships (note that these files may not
include relevant information, so be critical when analyzing them):
<FILES_CONTENT>
[[INPUT FILES CONTENT]]
</FILES_CONTENT>
```

A.5 Component identifier prompt

```
Based on the files from a container/service (C4-level container) in a software project
given below, identify and extract information about the components (C4-level components)
that make up the container. A component is a modular part of a container that encapsulates
specific functionality or services.

For each component you identify, provide the following information:
- name: The name of the component
- type: The type of component (e.g. module, API component, authentication service, data
access layer, etc.)
- programming_languages: List of programming languages used in the INDIVIDUAL component.
Also include frameworks if you detect any, but only do this if you are certain. (e.g., ['
Python', 'JavaScript', 'Django', 'React'])
- description: A brief description of what this component does. Do NOT make up descriptions
, only use information you can find in the provided files.
- file_paths: List of file paths in the repository where this component's code resides (e.g
., ['/path/to/file.py', '/another/path/module.js']). If you are uncertain about some file
paths, do not include them. Provide the empty list '[]' if you cannot find any file paths
for the component. Do not include directory paths, only file paths.

Important rules for your response:
[[FORMATTING AND CONTENT RULES]]

Note that the input may not explicitly define all components, so only include components
that you can clearly identify from the provided files.

Example response format for components in a container:
<EXAMPLE_RESPONSE>
<MOTIVATION>
[[ONE-SHOT EXAMPLE]]
</MOTIVATION>
<COMPONENTS>
[[ONE-SHOT EXAMPLE]]
</COMPONENTS>
</EXAMPLE_RESPONSE>

You are provided with the following information about the parent container:
<CONTAINER_INFO>
[[INPUT PARENT CONTAINER INFORMATION]]
</CONTAINER_INFO>

This is the directory tree of the parent container (note that some directories may be
truncated to prevent forming very large trees):
```

```
<PROJECT_DIRECTORY_TREE>
[[INPUT PARENT CONTAINER DIRECTORY TREE]]
</PROJECT_DIRECTORY_TREE>

Now provide the JSON response with all components you can identify.
```

A.6 Component summarization prompt

```
You are provided with information about a specific component (C4 model level component)
within a container/service (C4 model level container) of a software system, along with the
directory trees of the container's codebase and aggregated contents of important files
related to the component.
Your task is to generate a concise and informative summary of the component, highlighting
its purpose, functionality, and relationships with other components within the same parent
container, or with other external components.

Important guidelines for the summary:
[[FORMATTING AND CONTENT RULES]]

You will need to do this for the following component:
<COMPONENT_INFO>
[[INPUT COMPONENT INFORMATION]]
</COMPONENT_INFO>

Here is the information you have about the container in which the component resides:
<CONTAINER_INFO>
[[INPUT PARENT CONTAINER INFORMATION]]
</CONTAINER_INFO>

Here is the directory structure of the container:
<CONTAINER_DIRECTORY_TREE>
[[INPUT PARENT CONTAINER DIRECTORY TREE]]
</CONTAINER_DIRECTORY_TREE>

Use the following formatting and headings:
<EXAMPLE_RESPONSE>
[[ONE-SHOT EXAMPLE]]
</EXAMPLE_RESPONSE>

Here is a list of all components that have been identified in the same parent container,
along with their descriptions and file paths:
<COMPONENTS_LIST>
[[INPUT COMPONENTS INFORMATION]]
</COMPONENTS_LIST>

Here is the aggregated content of important files related to the component:
<AGGREGATED_FILES_CONTENT>
[[INPUT FILES CONTENT]]
</AGGREGATED_FILES_CONTENT>
```

A.7 Container summarization prompt

```
You are provided with information about a specific container/service (C4 model level
container) within a software system, along with the directory trees of the container's
```

```
codebase and summaries of the C4-level components that make up the container.
Your task is to generate a concise and informative summary of the container, highlighting
its purpose, functionality, relationships with other containers and components, and any
other relevant details that would help a developer understand its role within the overall
system.

Important guidelines for the summary:
[[FORMATTING AND CONTENT RULES]]

You will need to do this for the following container:
<CONTAINER_INFO>
[[INPUT CONTAINER INFORMATION]]
</CONTAINER_INFO>

Here is the directory structure of the container:
<CONTAINER_DIRECTORY_TREE>
[[INPUT CONTAINER DIRECTORY TREE]]
</CONTAINER_DIRECTORY_TREE>

Use the following formatting and headings:
<EXAMPLE_RESPONSE>
[[ONE-SHOT EXAMPLE]]
</EXAMPLE_RESPONSE>

Here are the summaries of the components within the container:
<COMPONENT_SUMMARIES>
[[INPUT CHILD COMPONENTS SUMMARIES]]
</COMPONENT_SUMMARIES>
```

A.8 Component relationship identification prompt

```
You are provided with a list of identified C4 model components within a parent C4 container
 in a software system.
Your task is to analyze the provided information and propose component-component
relationships between the identified components. For each proposed relationship, provide
the following information:
- source_id: The ID of the source component in the relationship
- target_id: The ID of the target component in the relationship (this is the component to
which the arrowhead points)
- label: A brief label or name for the relationship (e.g., "uses", "calls", "provides data
to", "sends messages to")
- bidirectional: A boolean value indicating whether the relationship is bidirectional (true
) or unidirectional (false)

Note that the proposed relationships should be based on the analysis of the provided
information, and should not be speculative. Only include proposed relationships that you
can reasonably infer from the provided data.

Important rules for your response:
[[FORMATTING AND CONTENT RULES]]

Example response format for relationships between components:
<EXAMPLE_RESPONSE>
<MOTIVATION>
[[ONE-SHOT EXAMPLE]]
</MOTIVATION>
<RELATIONSHIPS>
[[ONE-SHOT EXAMPLE]]
```

```
</RELATIONSHIPS>
</EXAMPLE_RESPONSE>

The components reside in the following parent container:

<CONTAINER_INFO>
[[INPUT PARENT CONTAINER INFORMATION]]
</CONTAINER_INFO>

You need to do this for the following components:
[[INPUT COMPONENTS INFORMATION]]
```

A.9 Claude Code top-line comparison prompt

```
You are an autonomous software architecture recovery agent.
Your task is to analyze this repository and produce high-quality C4 architecture diagrams.
You must work fully autonomously and plan your own steps.

---

OBJECTIVE

Generate:
1. A C4 container diagram
2. A C4 component diagram for each container

Use C4-PlantUML.

---

PROCESS REQUIREMENTS (IMPORTANT)

You MUST follow an iterative improvement process:
1. Explore the repository
   - Identify relevant files (e.g. configs, services, APIs, modules)
   - Understand system structure

2. Generate an initial version of:
   - Container diagram
   - Component diagrams

3. Critically evaluate your own output:
   - Identify missing elements
   - Identify hallucinations (elements not supported by code)
   - Identify weak or incorrect relationships
   - Identify poor naming or abstraction

4. Improve your diagrams:
   - Remove unsupported elements
   - Add missing elements where justified
   - Refine relationships
   - Improve clarity and correctness

5. Repeat steps 3-4 UNTIL:
   - No major issues remain
   - You are confident the diagrams accurately reflect the system

---
```

```
STRICT RULES

- Only include elements supported by evidence in the repository
- Do NOT fabricate containers, components, or relationships
- Prefer under-specification over hallucination
- Use clear and concise naming
- Follow C4 modeling principles

---

STOP CONDITION

Stop iterating when:
- Additional improvements are minimal
- No clear hallucinations or major omissions remain

---

OUTPUT FORMAT (FINAL ONLY)

Return ONLY:
1. Final C4 container diagram (PlantUML)
2. Final C4 component diagrams (PlantUML)

Do NOT include intermediate reasoning or earlier drafts in the final output.

---

Begin your analysis.
```

Appendix B

C4 Diagrams

B.1 C4Gen

Container overview

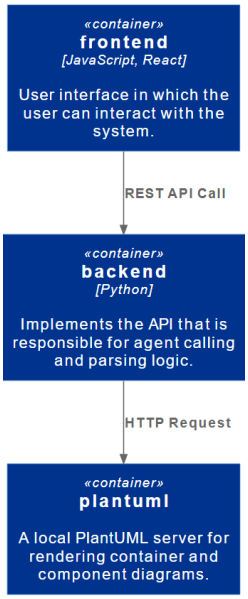


Figure 17: Reference container diagram for the C4Gen POC, manually created by the authors.

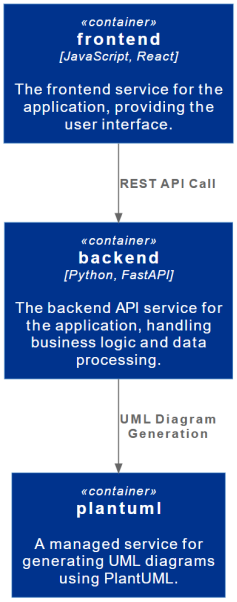


Figure 18: Container diagram for the C4Gen POC, generated by the POC.

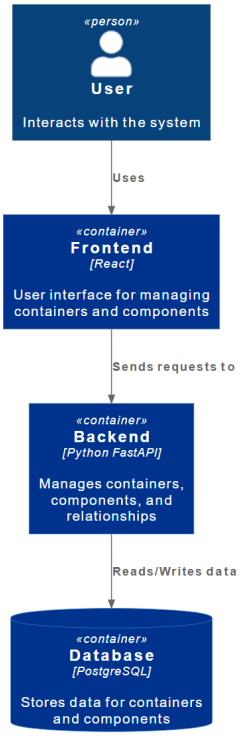


Figure 19: Container diagram for the C4Gen POC, generated by GitHub Copilot.

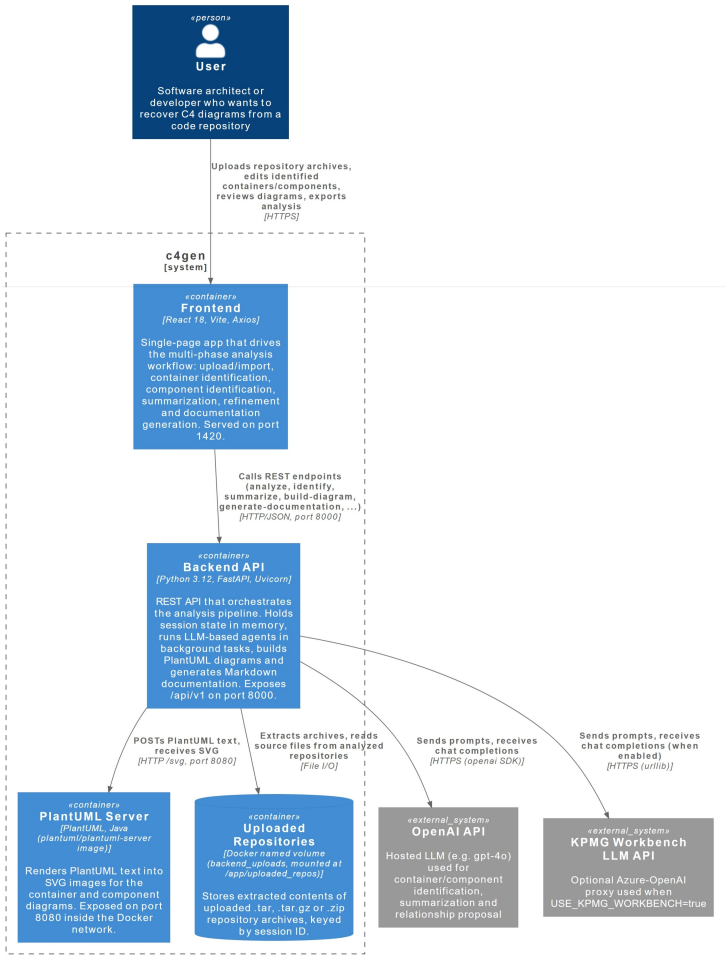


Figure 20: Container diagram for the C4Gen POC, generated by Claude Code.

Backend container

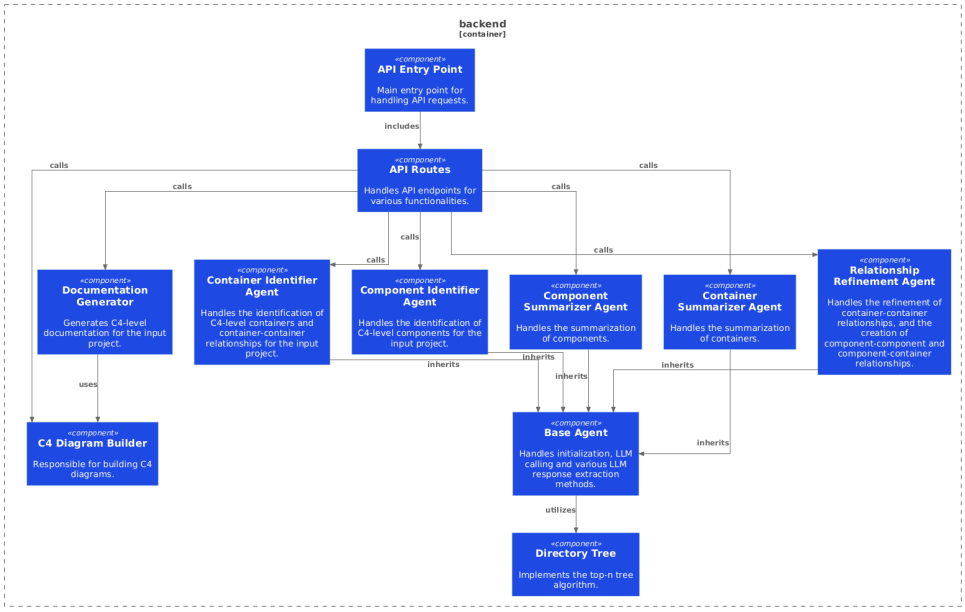


Figure 21: Reference component diagram for the backend container of the C4Gen POC, manually created by the authors.

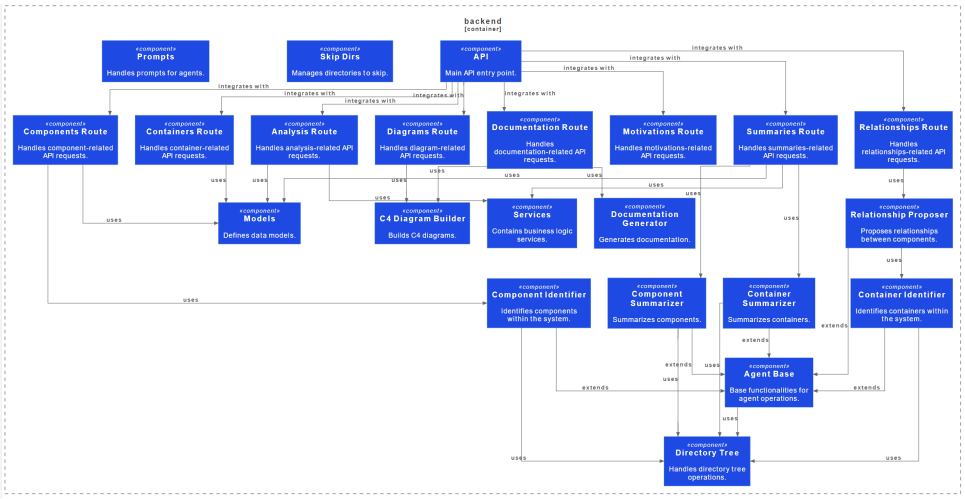


Figure 22: Component diagram for the backend container of the C4Gen POC, generated by the POC.

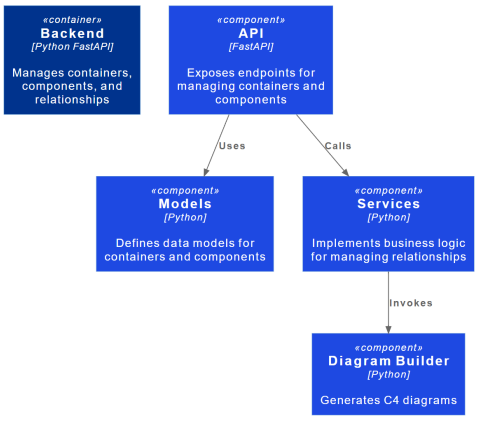


Figure 23: Component diagram for the backend container of the C4Gen POC, generated by GitHub Copilot.

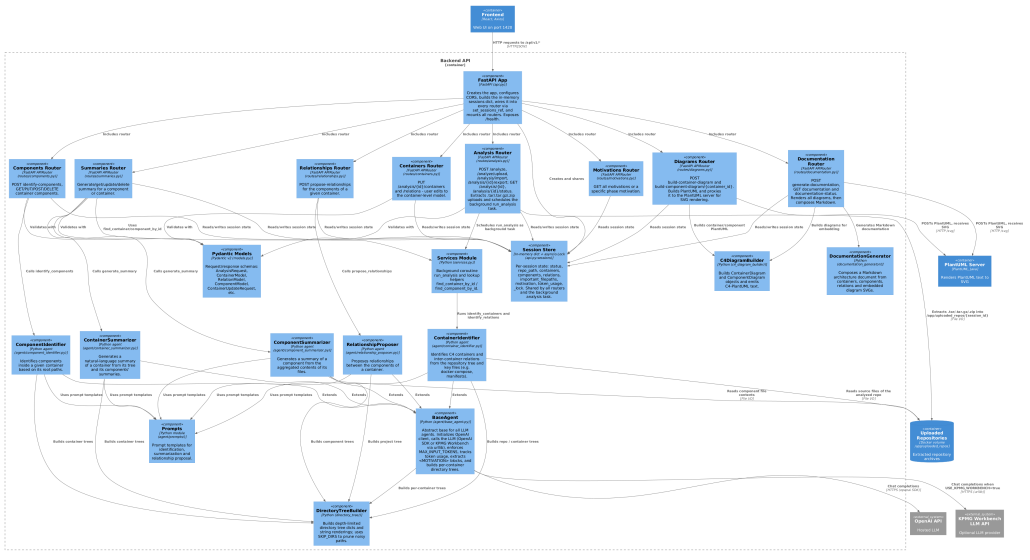


Figure 24: Component diagram for the backend container of the C4Gen POC, generated by Claude Code.

Frontend container

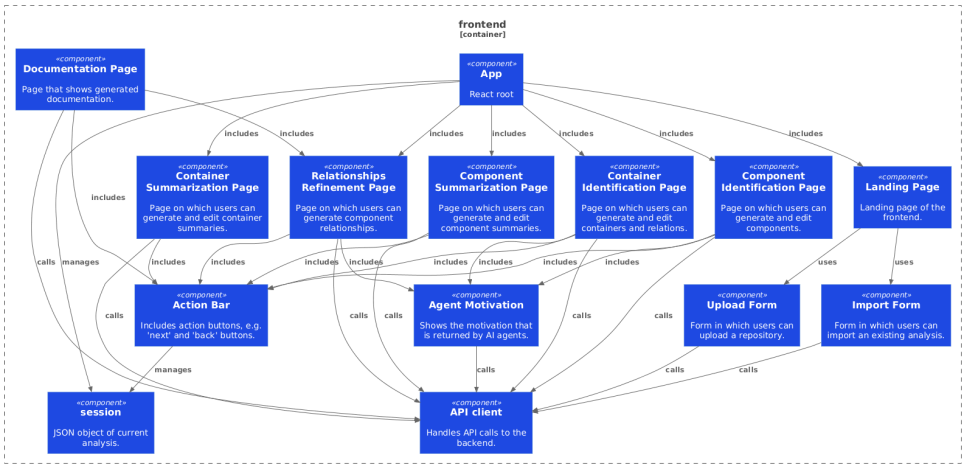


Figure 25: Reference component diagram for the frontend container of the C4Gen POC, manually created by the authors.

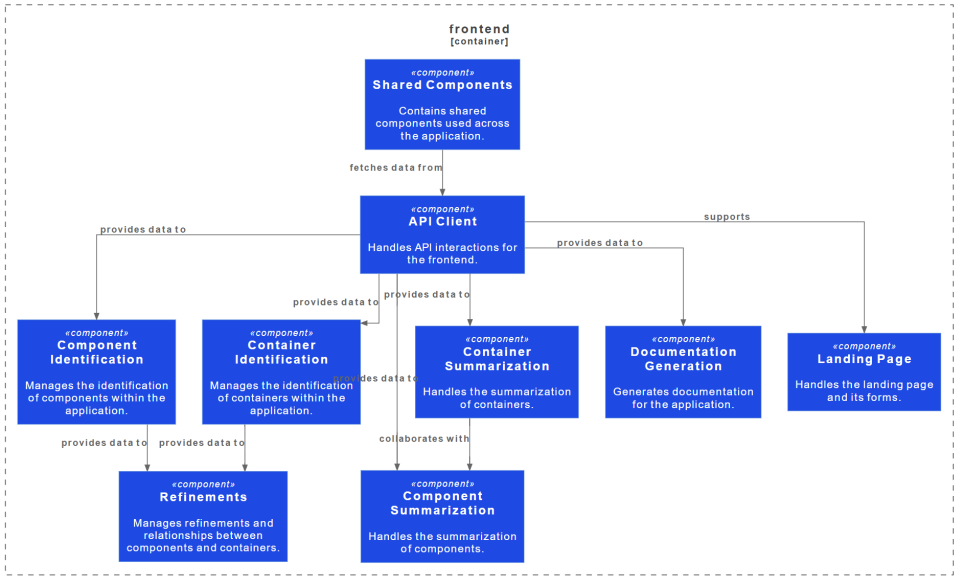


Figure 26: Component diagram for the frontend container of the C4Gen POC, generated by the POC.

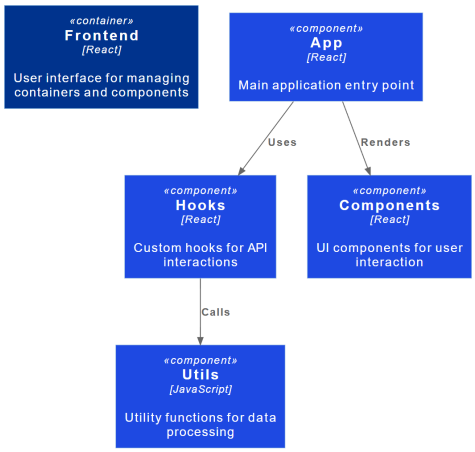


Figure 27: Component diagram for the frontend container of the C4Gen POC, generated by GitHub Copilot.

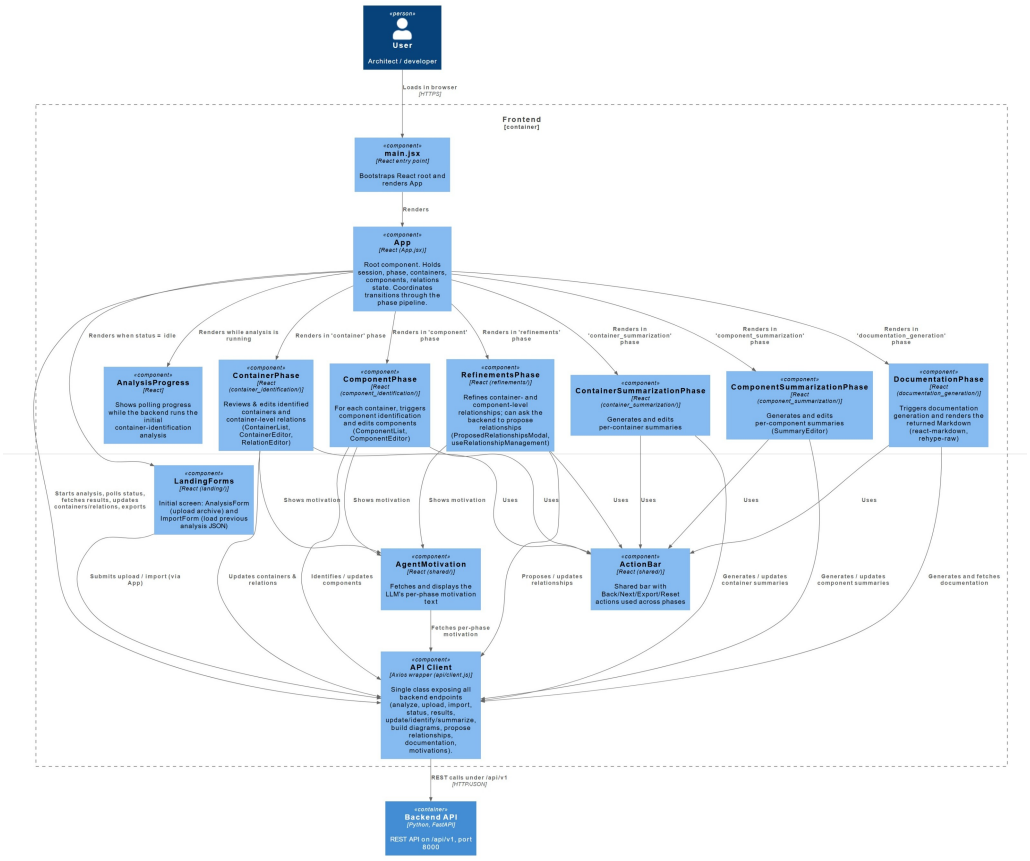


Figure 28: Component diagram for the frontend container of the C4Gen POC, generated by Claude Code.

B.2 Abacus

Container overview

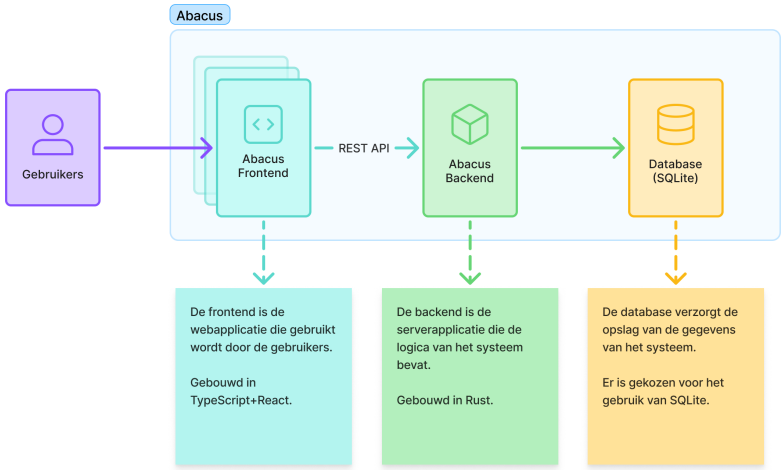


Figure 29: Reference container overview diagram for Abacus, created by the developers of Abacus. [36]

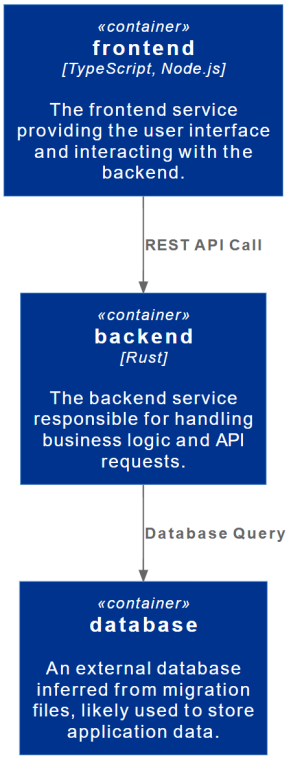


Figure 30: Container overview diagram for Abacus, generated by the POC.

```
    @startuml
    !include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/
C4_Container.puml

    LAYOUT_WITH_LEGEND()

    Person(user, "User", "Interacts with the system via the web interface")

    System_Boundary(abacus, "Abacus System") {
        Container(frontend, "Frontend", "JavaScript", "Web-based user interface")
        Container(backend, "Backend", "Rust", "Handles business logic and APIs")
        ContainerDb(database, "Database", "SQL", "Stores elections, polling stations, and
user data")
        ContainerExt(external, "External Services", "Various", "External integrations")
    }

    Rel(user, frontend, "Uses")
    Rel(frontend, backend, "Sends requests to", "REST API")
    Rel(backend, database, "Reads/Writes data to")
    Rel(backend, external, "Integrates with")

    @enduml
```

Figure 31: Container overview diagram for Abacus, generated by GitHub Copilot. This PlantUML string could not be rendered due to a syntax error (`ContainerExt` is not valid syntax; should be `Container_Ext`).

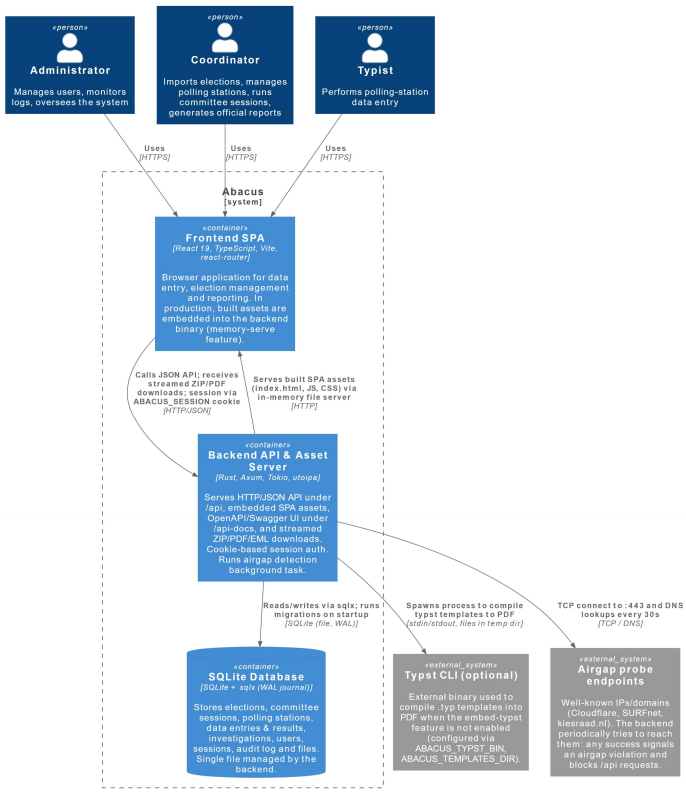


Figure 32: Container overview diagram for Abacus, generated by Claude Code

Frontend container

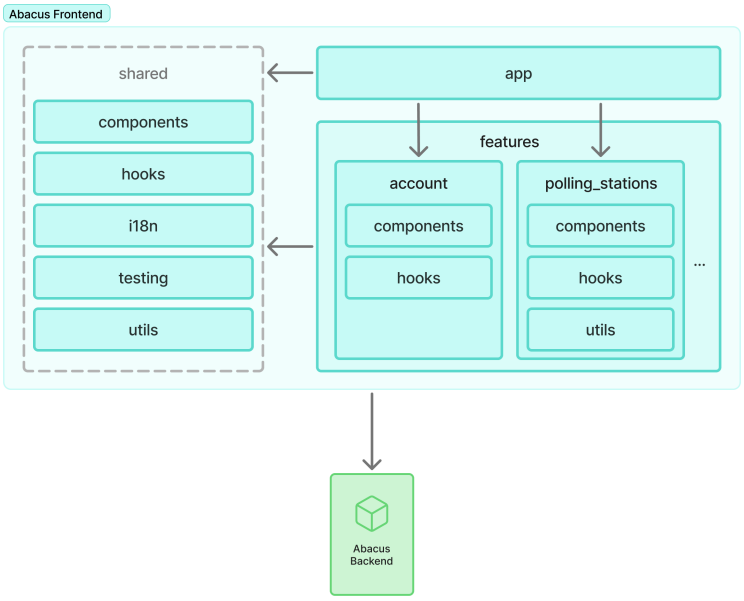


Figure 33: Reference component diagram for the frontend container of Abacus, created by the developers of Abacus. [36]

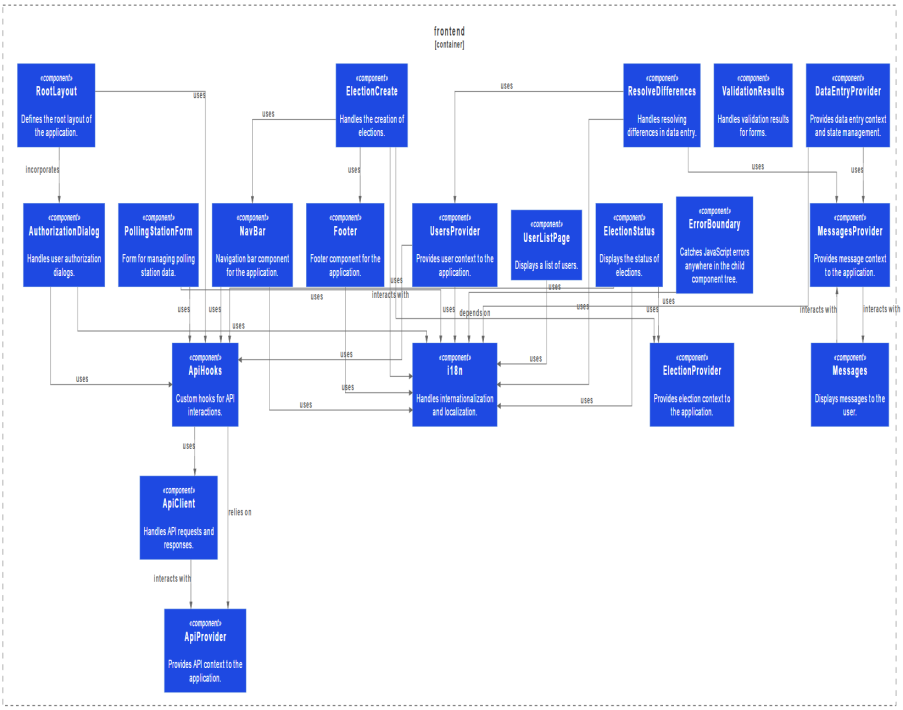


Figure 34: Component diagram for the frontend container of Abacus, generated by the POC.

```
    @startuml
    !include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/
C4_Component.puml

    Container_Frontend(frontend, "Frontend", "TypeScript", "Provides the user interface")

    Component(error_handling, "Error Handling", "TypeScript Component", "Manages error
display")
    Component(footer, "Footer", "TypeScript Component", "Displays the footer")
    Component(navbar, "Navbar", "TypeScript Component", "Manages the navigation bar")
    Component(polling_station_ui, "Polling Station", "TypeScript Component", "Handles
polling station-related UI")
    Component(committee_session_ui, "Committee Session", "TypeScript Component", "Manages
committee session UI")
    Component(data_entry_ui, "Data Entry", "TypeScript Component", "Handles data entry UI")
    Component(election_ui, "Election", "TypeScript Component", "Manages election-related UI
")

    Rel(frontend, backend, "Consumes REST APIs")
    Rel(frontend, playwright, "Uses", "End-to-End Testing")

    @enduml
```

Figure 35: Component diagram for the frontend container of Abacus, generated by GitHub Copilot. This PlantUML string could not be rendered due to a syntax error (external reference to `Container_Frontend` that does not exist).

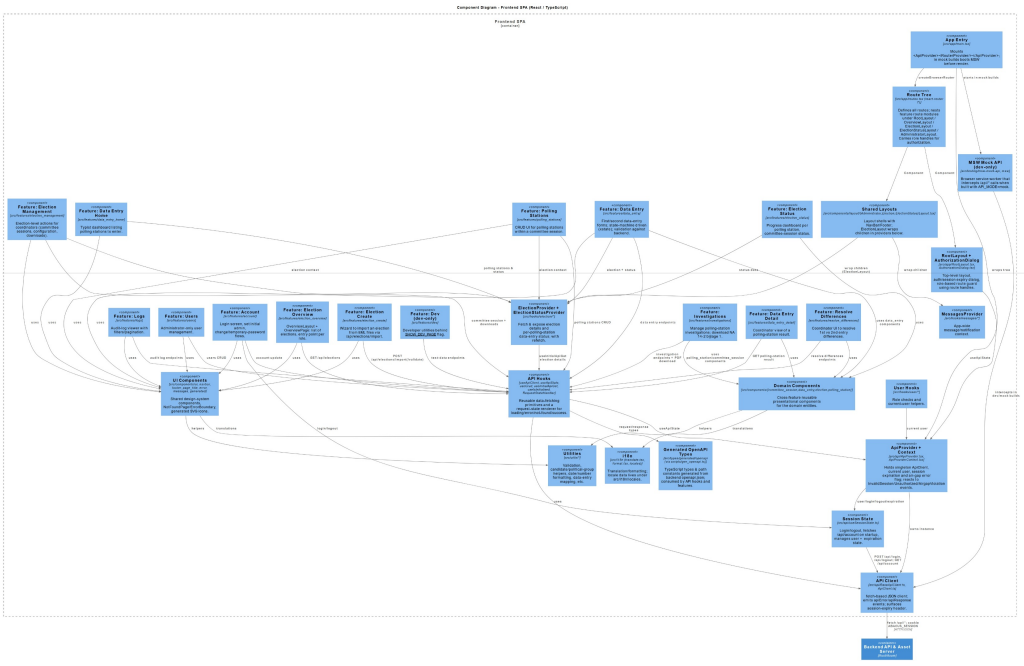


Figure 36: Component diagram for the frontend container of Abacus, generated by Claude Code.

B.3 AI4MDE

Container overview

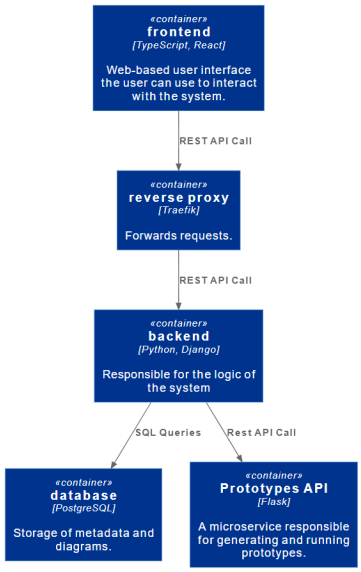


Figure 37: Reference container diagram for AI4MDE, manually created by the authors.

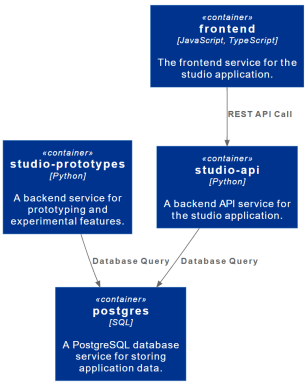


Figure 38: Container diagram for AI4MDE, generated by the POC.

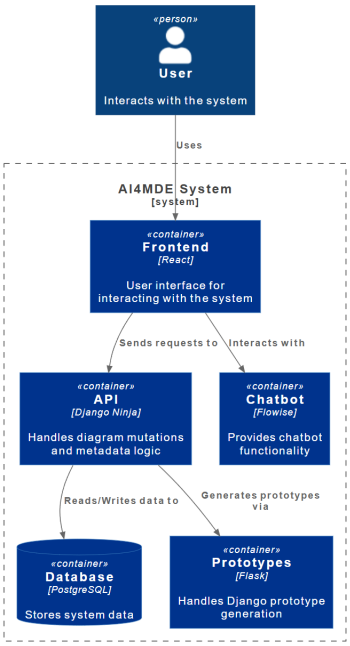


Figure 39: Container diagram for AI4MDE, generated by GitHub Copilot.

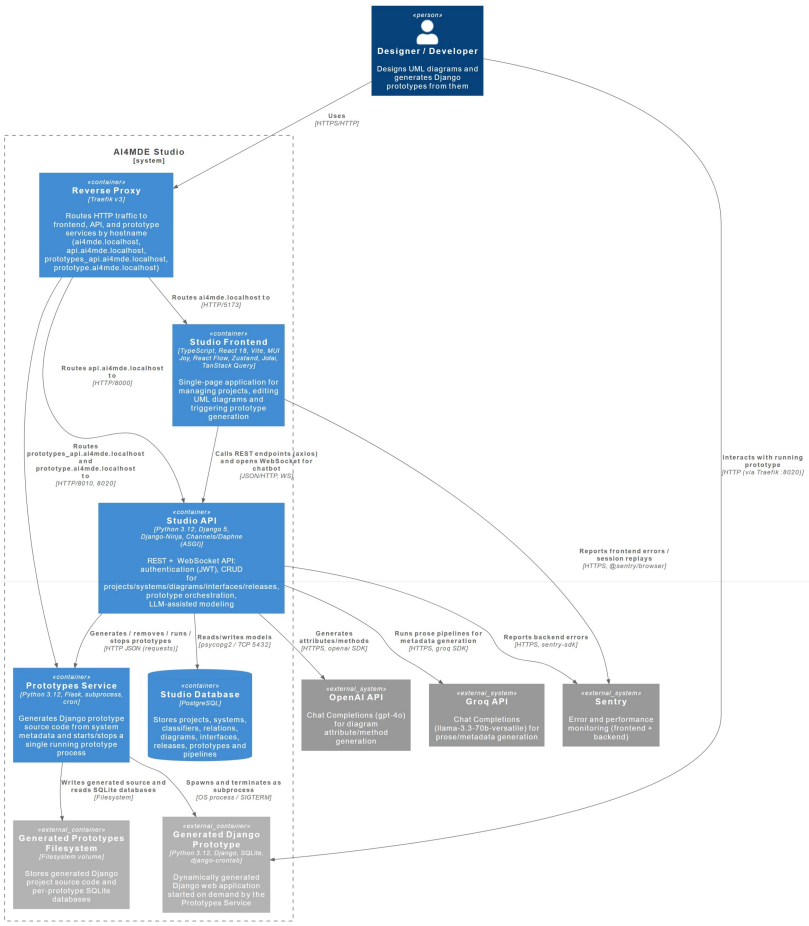


Figure 40: Container diagram for AI4MDE, generated by Claude Code.

Backend container

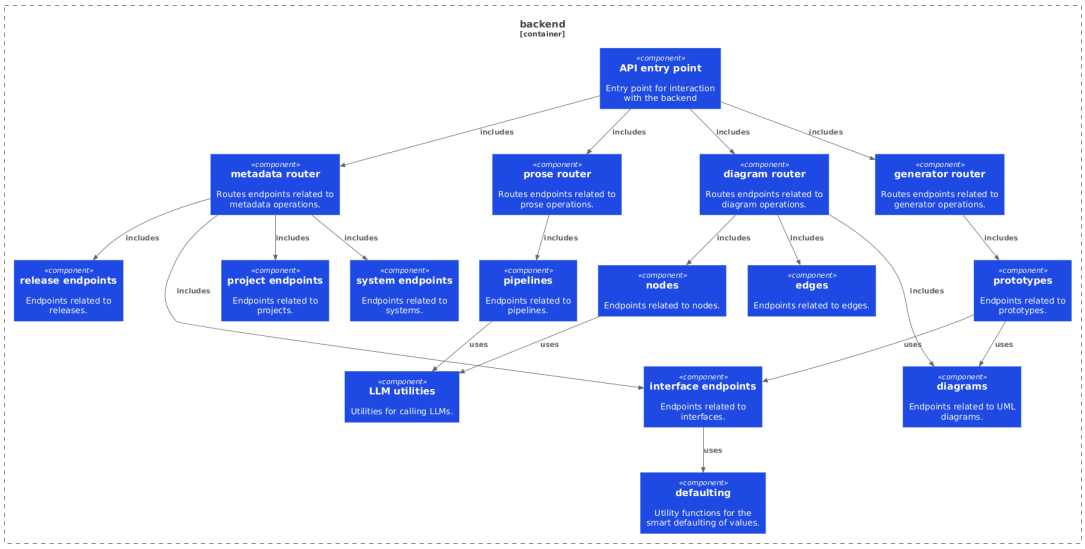


Figure 41: Reference component diagram for the backend container of AI4MDE, manually created by the authors.

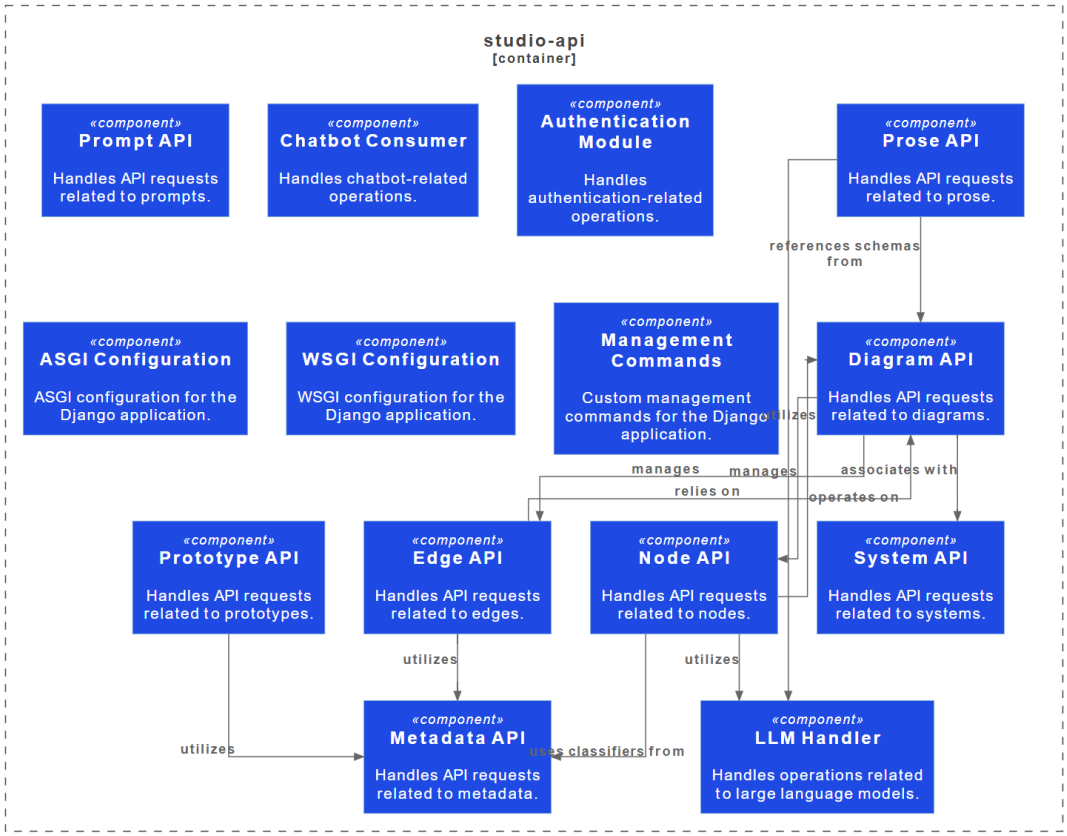


Figure 42: Component diagram for the backend container of AI4MDE, generated by the POC.

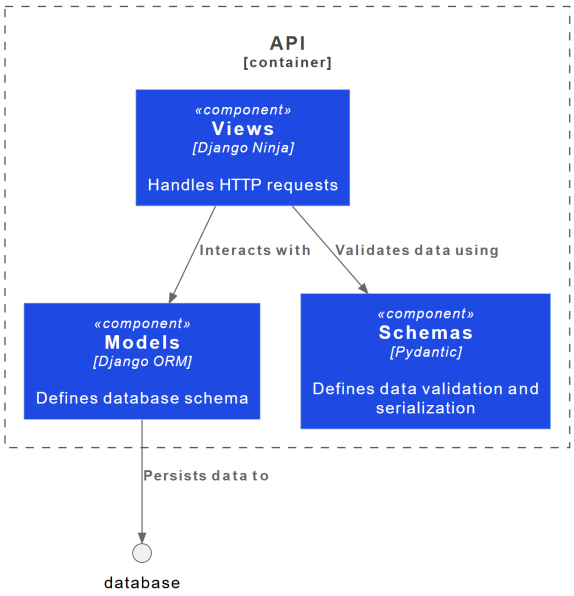


Figure 43: Component diagram for the backend container of AI4MDE, generated by GitHub Copilot.

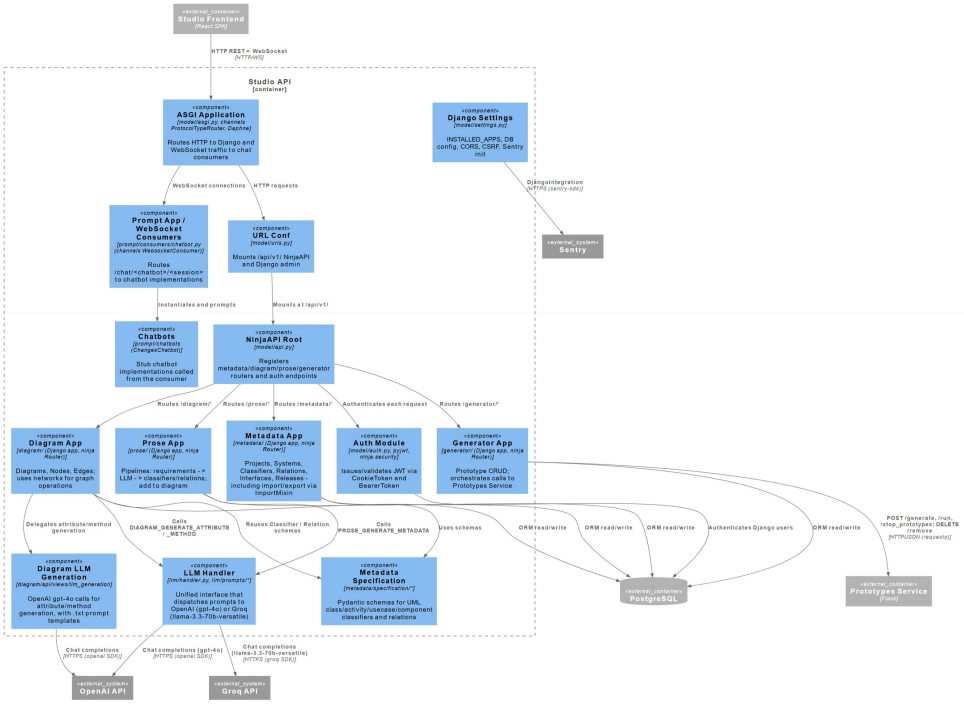


Figure 44: Component diagram for the backend container of AI4MDE, generated by Claude Code.

Frontend container

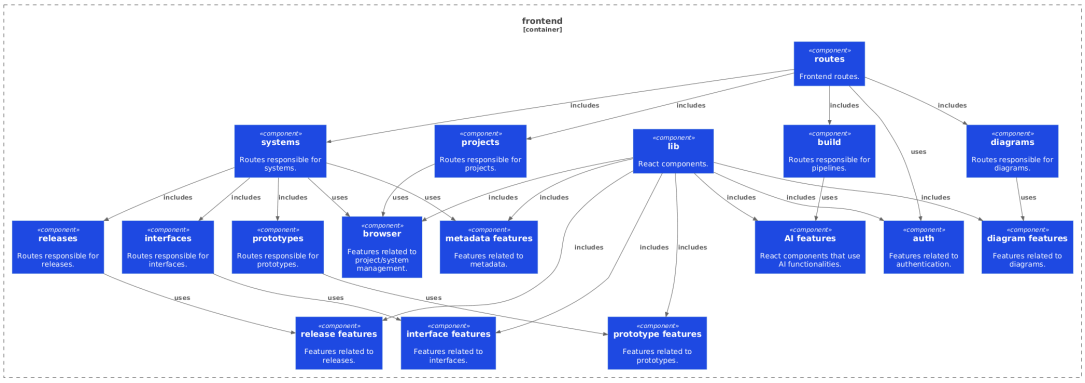


Figure 45: Reference component diagram for the frontend container of AI4MDE, manually created by the authors.

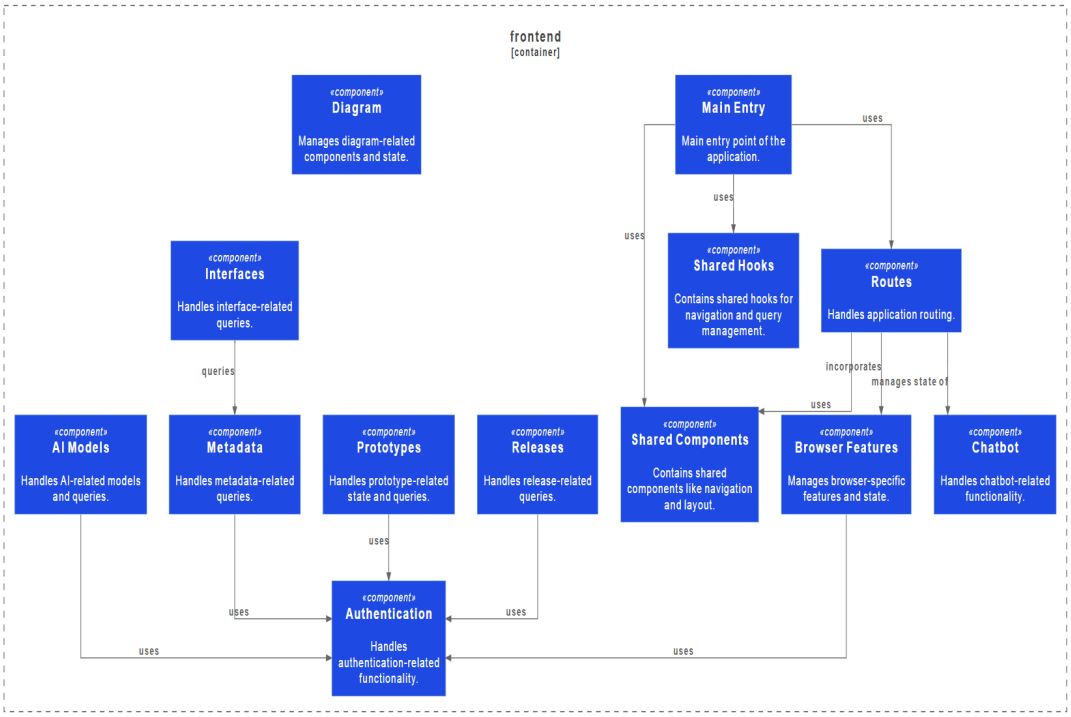


Figure 46: Component diagram for the frontend container of AI4MDE, generated by the POC.

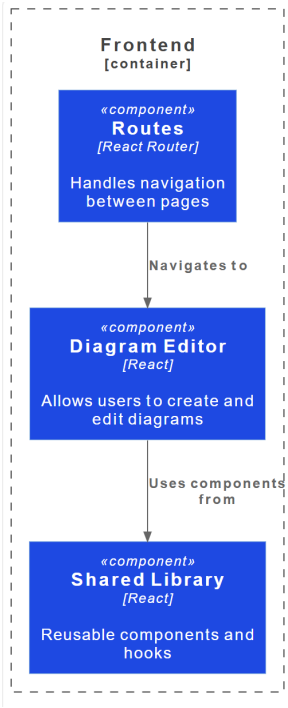


Figure 47: Component diagram for the frontend container of AI4MDE, generated by GitHub Copilot.

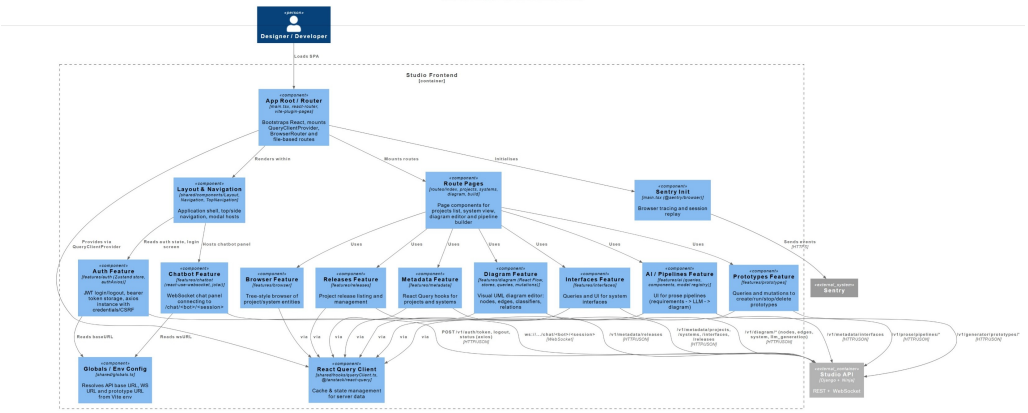


Figure 48: Component diagram for the frontend container of AI4MDE, generated by Claude Code.

Prototypes API container

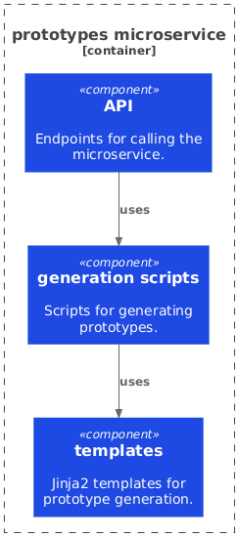


Figure 49: Reference component diagram for the prototypes API container of AI4MDE, manually created by the authors.

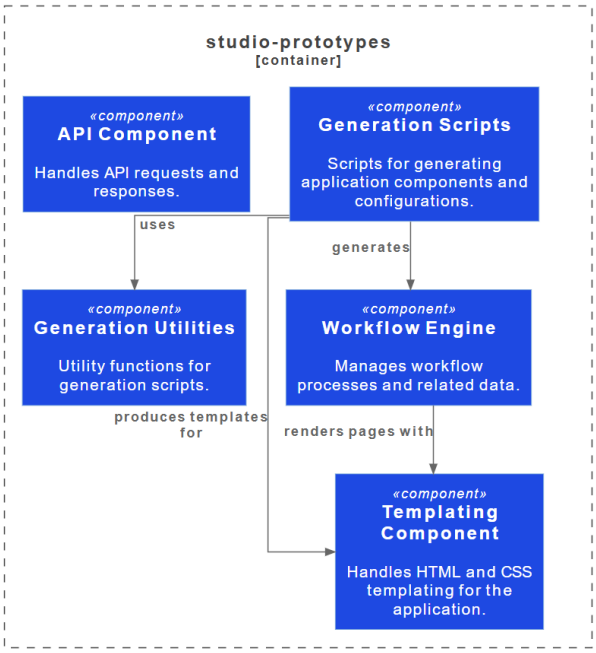


Figure 50: Component diagram for the prototypes API container of AI4MDE, generated by the POC.

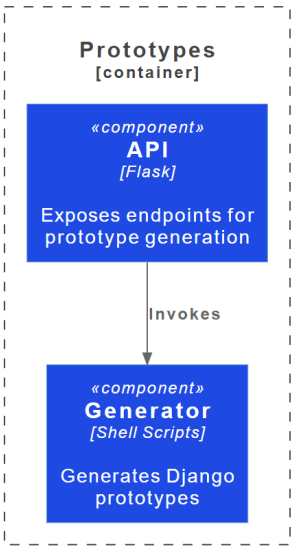


Figure 51: Component diagram for the prototypes API container of AI4MDE, generated by GitHub Copilot.

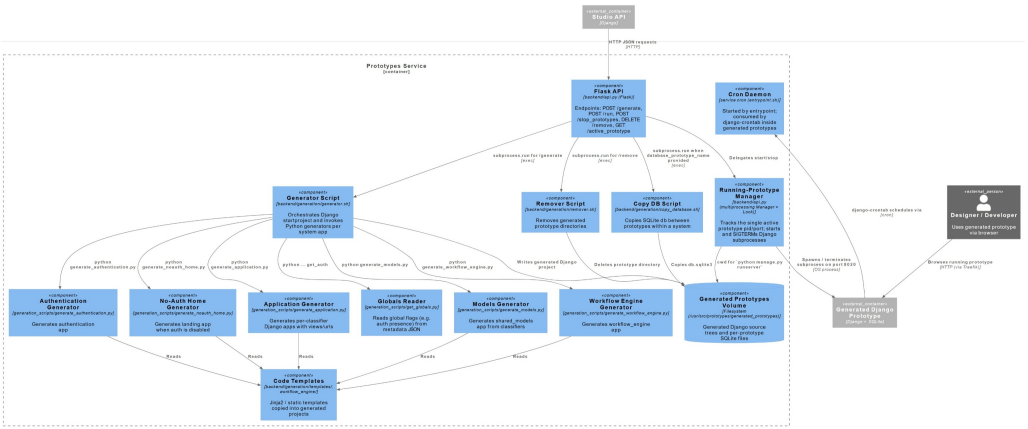
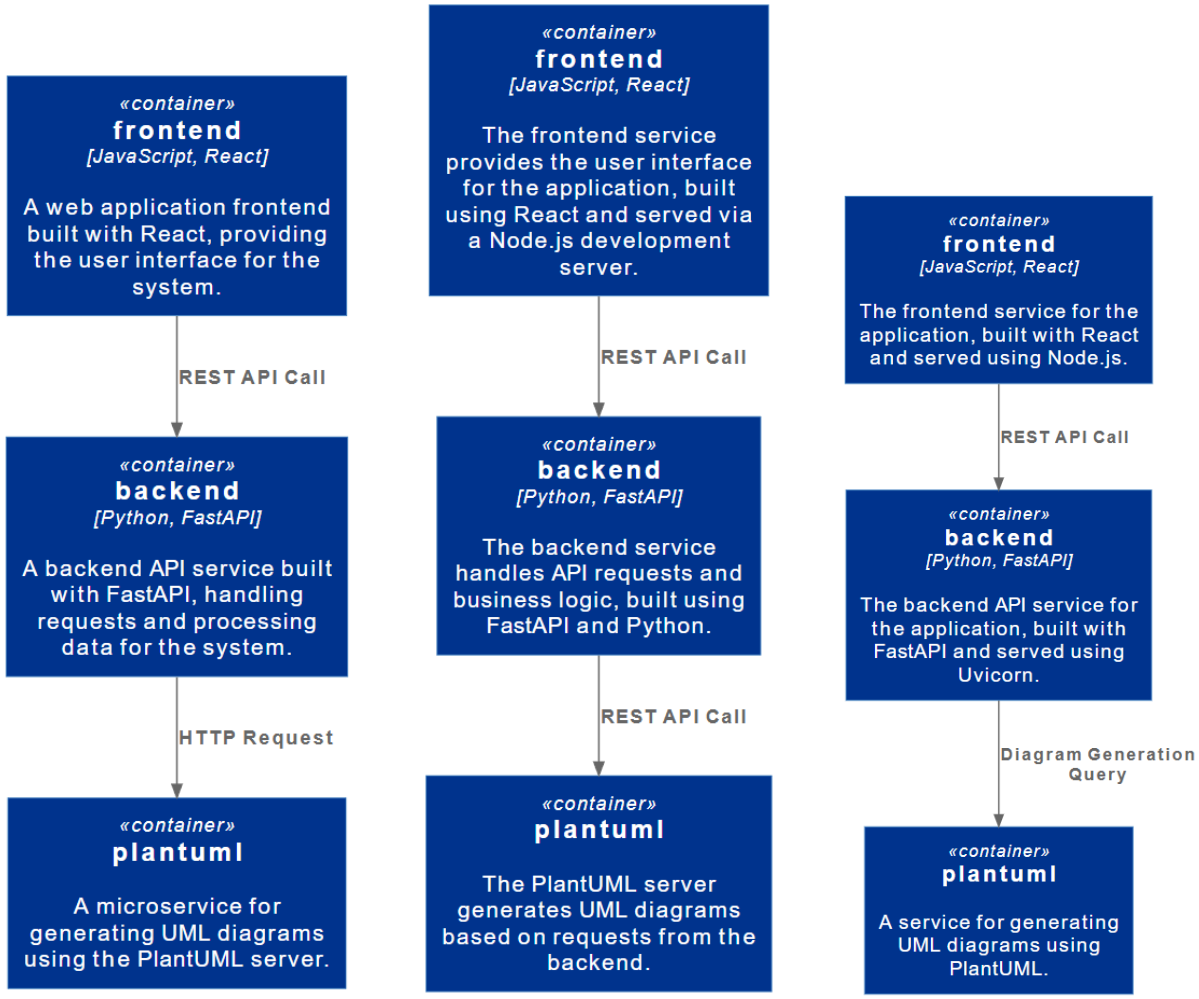


Figure 52: Component diagram for the prototypes API container of AI4MDE, generated by Claude Code.

B.4 Consistency study (C4Gen)

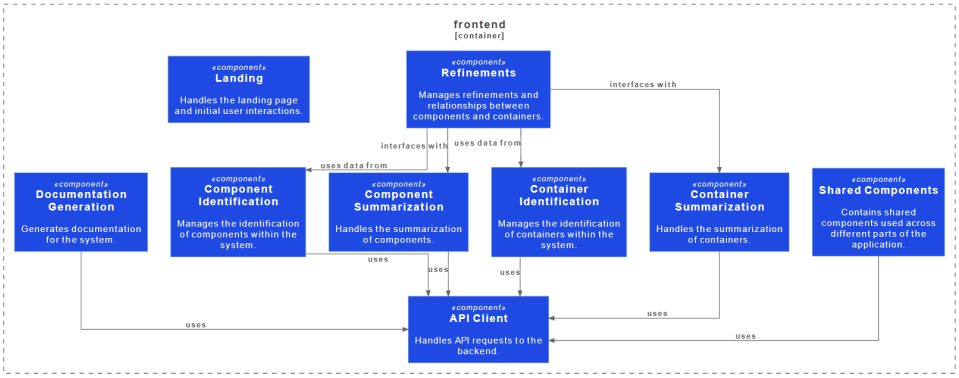
Container overview



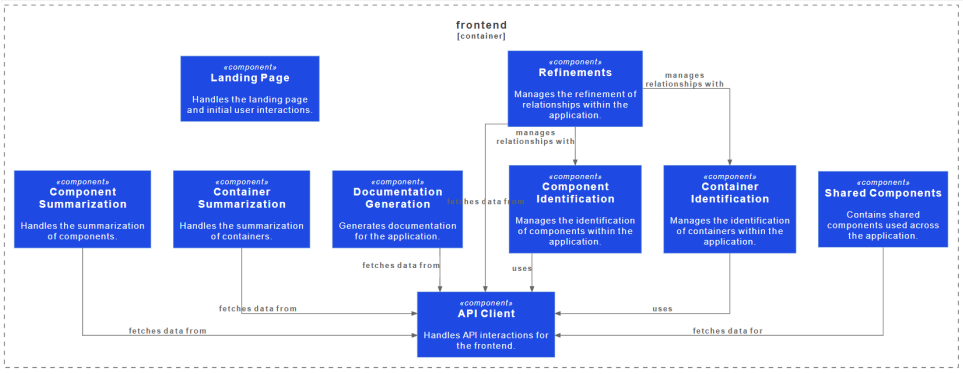
(a) First run (b) Second run (c) Third run

Figure 53: Container diagrams for the C4Gen POC across three runs, generated using the POC.

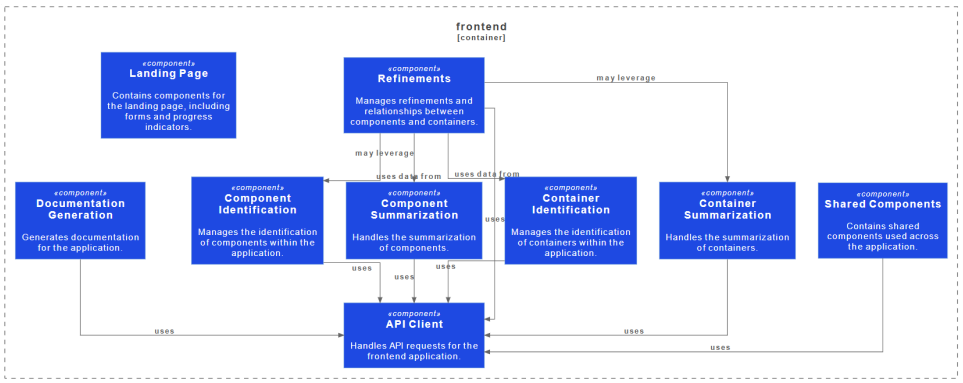
Frontend container



(a) First run



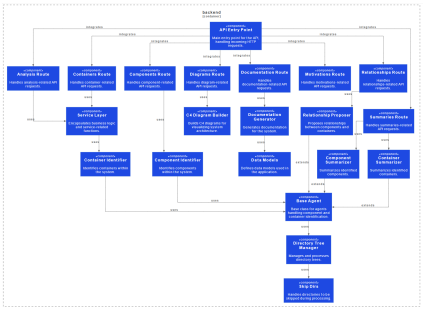
(b) Second run



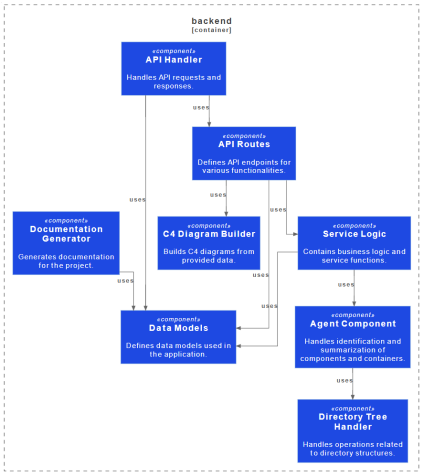
(c) Third run

Figure 54: Component diagrams for the frontend container of the C4Gen POC across three runs, generated using the POC.

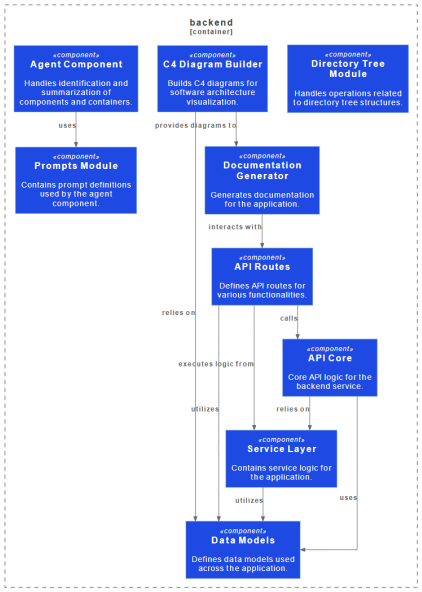
Backend container



(a) First run



(b) Second run



(c) Third run

Figure 55: Component diagrams for the backend container of the C4Gen POC across three runs, generated using the POC.

Appendix C

Semantic evaluation classifications

Containers

Participant	n_m	n_v	n_i	n_h	n_{mi}	n_{tot}	S_{raw}	$S_{\text{normalized}}$	$hallu\%$	acceptable?
P1 (C4Gen)	3	0	0	0	0	3	6	1.00	0.0	yes
P1 (AI4MDE)	4	0	0	0	0	4	8	1.00	0.0	yes
P2 (AI4MDE)	4	0	0	0	0	4	8	1.00	0.0	yes
P3 (AI4MDE)	4	0	0	0	0	4	8	1.00	0.0	yes
P4 (AI4MDE)	3	1	0	0	1	5	5	0.75	0.0	partially
P1 (Abacus)	3	0	0	0	0	3	6	1.00	0.0	yes
P5 (Abacus)	3	0	0	0	0	3	6	1.00	0.0	yes
P6 (Abacus)	2	1	0	0	0	3	5	0.92	0.0	yes
P7 (Abacus)	3	0	0	0	0	3	6	1.00	0.0	yes
P8 (Qubus)	1	0	2	0	0	3	0	0.50	0.0	no
P9 (Qubus)	3	0	0	0	0	3	6	1.00	0.0	yes

Table C.1: Container classifications for generated container diagrams, as evaluated by different stakeholders. The *acceptable* column shows whether the stakeholder believes the corresponding container diagram is acceptable. This *acceptable* column is used to reverse-engineer heuristic acceptable $S_{\text{normalized}}$ and $hallu\%$ thresholds.

Container–container relationships

Participant	n_m	n_v	n_i	n_h	n_{mi}	n_{tot}	S_{raw}	$S_{\text{normalized}}$	$hallu\%$	acceptable?
P1 (C4Gen)	2	0	0	0	0	2	4	1.00	0.0	yes
P1 (AI4MDE)	2	0	0	1	1	4	0	0.57	33.3	yes
P2 (AI4MDE)	2	0	0	1	1	4	0	0.57	33.3	yes
P3 (AI4MDE)	3	0	0	0	0	3	6	1.00	0.0	yes
P4 (AI4MDE)	3	0	0	0	0	3	6	1.00	0.0	partially
P1 (Abacus)	2	0	0	0	0	2	4	1.00	0.0	yes
P5 (Abacus)	2	0	0	0	0	2	4	1.00	0.0	yes
P6 (Abacus)	1	1	0	0	0	2	3	0.88	0.0	yes
P7 (Abacus)	2	0	0	0	0	2	4	1.00	0.0	yes
P8 (Qubus)	1	0	0	1	0	2	0	0.67	50.0	no
P9 (Qubus)	1	0	1	0	0	2	1	0.63	0.0	yes

Table C.2: Container–container relationship classifications for generated container diagrams, as evaluated by different stakeholders. The *acceptable* column shows whether the stakeholder believes the corresponding container diagram is acceptable. This *acceptable* column is used to reverse-engineer heuristic acceptable $S_{\text{normalized}}$ and $hallu\%$ thresholds.

Components

Participant	n_m	n_v	n_i	n_h	n_{mi}	n_{tot}	S_{raw}	$S_{\text{normalized}}$	$hallu\%$	acceptable?
P1 (C4Gen)	18	13	0	0	3	34	43	0.82	0.0	yes
P1 (AI4MDE)	24	3	4	0	2	33	43	0.83	0.0	yes
P2 (AI4MDE)	24	5	2	0	2	33	47	0.86	0.0	yes
P3 (AI4MDE)	25	4	2	0	3	34	46	0.84	0.0	partially
P4 (AI4MDE)	28	0	0	3	4	35	42	0.84	9.7	partially
P1 (Abacus)	10	8	2	0	0	20	26	0.83	0.0	yes
P5 (Abacus)	19	1	0	0	0	20	39	0.99	0.0	partially
P6 (Abacus)	14	6	0	0	1	21	32	0.88	0.0	yes
P7 (Abacus)	0	20	0	0	0	20	20	0.75	0.0	debatable
P8 (Qubus)	7	3	0	0	4	14	9	0.66	0.0	partially
P9 (Qubus)	10	0	0	0	0	10	20	1.00	0.0	yes

Table C.3: Component classifications for generated component diagrams (sum of all component diagrams of an input project), as evaluated by different stakeholders. The *acceptable* column shows whether the stakeholder believes the corresponding component diagrams are acceptable. This *acceptable* column is used to reverse-engineer heuristic acceptable $S_{\text{normalized}}$ and $hallu\%$ thresholds.

Component–component relationships

Participant	n_m	n_v	n_i	n_h	n_{mi}	n_{tot}	S_{raw}	$S_{\text{normalized}}$	$hallu\%$	acceptable?
P1 (C4Gen)	19	18	4	0	16	57	20	0.59	0.0	yes
P1 (AI4MDE)	22	2	1	2	3	30	35	0.82	7.4	yes
P2 (AI4MDE)	16	1	8	2	3	30	15	0.65	7.4	yes
P3 (AI4MDE)	22	3	1	1	9	36	26	0.69	3.7	partially
P4 (AI4MDE)	27	0	0	0	9	36	36	0.75	0.0	partially
P1 (Abacus)	11	19	0	0	0	30	41	0.84	0.0	yes
P5 (Abacus)	22	8	0	0	0	30	52	0.93	0.0	partially
P6 (Abacus)	30	0	0	0	0	30	60	1.00	0.0	yes
P7 (Abacus)	0	30	0	0	0	30	30	0.75	0.0	debatable
P8 (Qubus)	11	2	0	1	4	18	14	0.71	7.1	partially
P9 (Qubus)	14	0	0	0	0	14	28	1.00	0.0	yes

Table C.4: Component–component relationship classifications for generated component diagrams (sum of all component diagrams of an input project), as evaluated by different stakeholders. The *acceptable* column shows whether the stakeholder believes the corresponding component diagrams are acceptable. This *acceptable* column is used to reverse-engineer heuristic acceptable $S_{\text{normalized}}$ and $hallu\%$ thresholds.

Appendix D

Survey responses

Question	P1	P2	P3	P4	P5
Q1	Positive and helpful	Generally positive, but background knowledge is required	Positive	Neutral, some confusion	Positive
Q2	Yes	Output quality is inconsistent	Valid base, but missing relationships and inconsistency concerns	Somewhat, some hallucinations	Yes
Q3	Yes, definitely	Yes, possibly	Definitely	Yes, if observability is improved	Yes
Q4	Yes, saves time	No speed-up, but can be used for validation	Yes, but only for quick overviews. Also some UI flaws.	Yes, to get a first draft	Not per se
Q5	Yes	Yes	Yes	Only speed-up in time, no boost in effectivity	Yes
Q6	None	One missing	Three missing	Some higher-level improvements are needed	Yes, but minor improvements needed
Q7	Yes	Yes, but minor improvements needed	Yes, but minor improvements needed	Yes	Yes
Q8	None	Improvement needed in quality of diagrams	Clarifying header texts needed	None in C4Gen's current form	Minor UI improvements
Q9	Yes, if allowed	Possibly	Definitely	Possibly	Yes

Table D.1: Generic summarized content of pragmatic quality question responses for different survey participants.

Participant	**Response Q1** – *"What is your overall impression of the POC?"*
P1	"I liked it very much since when doing code reviews, we often encounter technologies in which we can't easily extract any form of documentation or architecture diagrams. This tool would help us/me to immediately have an overview of what the structure of the source code looks like."
P2	"In general it functions well. To understand the use for the product some background knowledge of the C4 model is required. The quality of the output for the front-end and back-end did differ."
P3	"Overall a good tool to gain a quick insight in the application architecture."
P4	"Fine UI/UX for editing agent results. Unclear what information exactly goes into the agents, what is validated, and what not. Not sure what my human-input and feedback does for the end result. Lots of text at times."
P5	"Works well, minor styling or logic feedback."

Table D.2: Full survey participant responses to Q1 in the pragmatic quality evaluation survey.

Participant	**Response Q2** – *"Do you believe that the generated diagrams cover the input system's architecture?"*
P1	"In this case for Abacus it does, I am curious how it works with an older codebase with bad naming conventions etc."
P2	"For the back-end the diagrams gave a good overview of the system's architecture. The front-end had levels of details that I wouldn't expect, while some components I was expecting were left out."
P3	"I believe the structured approach to provide a valid base to draw up the architecture. However, there may be some component-to-component relationships missing due to the fact that C4Gen currently only discovers intra-container relationships. I do have a doubt whether the output is consistent - when running the LLM multiple times on the same application, does it generate the same overview (although texts might differ slightly) or is there a chance of deviations with every generation."
P4	"Somewhat. It gladly hallucinated an explanation for a "fake" container I added. I also removed a possibly crucial container, but it was unclear how this broke (or not) the diagram."
P5	"Yes."

Table D.3: Full survey participant responses to Q2 in the pragmatic quality evaluation survey.

Participant	**Response Q3** – *"Could you foresee yourself using the POC in any of your operations within your organization?"*
P1	"I would like to use it! Mostly for helping me create a visual to understand the codebase in a faster and more easy way."
P2	"I could, it can give an indication of where to look for further exploration in the source code. Especially the dependencies between things are useful."
P3	"Definitely"
P4	"With better observability, yes. It seems decent at digesting a folder-tree into something slightly more semantic. I'd probably use it after doing my own interpretation of a code base to get a visual representation of the system, a "lay of the land" if you will to use as future reference. Essentially I see it as an easier way to get a visualization at a greater level of depth I would usually draft one. I would not use C4Gen before having looked at a code base myself, since everything seem very believable, but I do not yet trust it enough to give me correct info. I see it more as a black box that can sometimes output a detailed visualization that can be easily verified if you have sufficient context on a system already."
P5	"Yes"

Table D.4: Full survey participant responses to Q3 in the pragmatic quality evaluation survey.

Participant	**Response Q4** – *"Do you think the POC would improve the productivity in your operations?"*
P1	"It does, since building a diagram from scratch in an unknown codebase costs a lot of time."
P2	"It would not speed me up a lot I think, I can use it to validate or generate insights mainly."
P3	"Yes, as it provides software quality reviewers with a quick(er) insight in the architecture of the application under review. There is a risk that a practitioner unfamiliar with the source code / application misses some of the manual containers, components or relationships, and places too much trust on what the LLM identifies. For a quick overview, this helps. For the more in-depth analysis, I feel a manual approach is still necessary. Current flaw in the UI may hinder productivity though, as mistakes (e.g. accidental deletion of a relationship) cannot be undone or re-generated."
P4	"Somewhat. Visuals make collaboration easier, because you simply "point" to a piece of the system and people will know what you are discussing at least. It can also be a good reference point to record more information in a structured way. Making visuals by hand, and annotating with extra data, usually requires quite some effort, especially to get a first working draft that everyone can agree on. Having that done automatically and somewhat verifiable is helpful."
P5	"Not per se."

Table D.5: Full survey participant responses to Q4 in the pragmatic quality evaluation survey.

Participant	**Response Q5** – *"Do you think you can be more effective by using the POC?"*
P1	"Yes."
P2	"I think it can benefit the quality of my output."
P3	"Yes, as it provides an initial overview of the architecture. Users can further explore the application source code with the output of C4Gen as guidance, and modify / shape the architecture further while doing so."
P4	"Maybe it saves some time here and there. It certainly feels convenient. I don't think it will help me get better at actually analyzing architectures. If using it to get an initial view of a system before you actually conceive of a view yourself it will probably result in bias short-term and de-skilling long term. In that sense it's similar to SonarQube, which also makes it easy to focus on the "wrong" thing by virtue of not showing the "right" thing at times, except C4Gen is less transparent making the failure-modes less tractable. C4Gen also removes a key component of building system-level understanding, which is browsing module/file hierarchies, since that is roughly the highest level of information that conveys meaning about the system structure at a technical level. I can imagine that hurts the learning-effect."
P5	"Yes."

Table D.6: Full survey participant responses to Q5 in the pragmatic quality evaluation survey.

Participant	**Response Q6** – *"What functionalities do you think that the POC is lacking?"*
P1	"For this stage none, it does what I would want it to do and that is to create a basic visualisation of the codebase to help me understand it in a better way."
P2	"To me it makes more sense to be able to add relations/components/etc. after the generation. It is a bit harder to do so before having seen the diagrams that were generated."
P3	"Inter-container relationships, overview of technologies used per component, option to export as a document (e.g. PDF, ODT, DOC)"
P4	"There's a lot of opportunity for advanced visualizations, interactivity and observability. Essentially a better modality for surfacing information. But that's in a very different domain in terms of research and implementation. In general "visualizing" a code base in an interactive way is not an easy problem at all. Improving observability should be easier, but probably will be textual in nature if not also addressing visualization. Another direction would be making the entire agentic process more systematic and process-like to improve quality of information retrieved."
P5	"No major functionalities, only UI improvements."

Table D.7: Full survey participant responses to Q6 in the pragmatic quality evaluation survey.

Participant	**Response Q7** – *"Do you think the POC is easy to use?"*
P1	"Yes."
P2	"It can use some UI/UX improvements (loading/error states/positioning of the navigation), but overall it is good."
P3	"Yes, although some clarifying texts at the header may be useful to explain what is expected from the user."
P4	"Quite easy. It requires almost no effort to produce some output that is parseable at a glance, and can be validated with some apriori knowledge of a system. In that sense it's very accessible."
P5	"Yes."

Table D.8: Full survey participant responses to Q7 in the pragmatic quality evaluation survey.

Participant	**Response Q8** – *"Do you have any suggestions for improvements that would make the solution easier to use?"*
P1	"No."
P2	"Mostly quality of the diagrams is important, otherwise see previous comments :)"
P3	"See above."
P4	"Not really, at least not in the current form. You press some buttons, you type out some fields and that's that. There are some general workflow things like bulk editing, versioning and all that blabla, but that's really more of a product-thing. Improving capabilities in information retrieval and surfacing is probably a more important direction"
P5	"Provided feedback during session like: card size, button location."

Table D.9: Full survey participant responses to Q8 in the pragmatic quality evaluation survey.

Participant	**Response Q9** – *"Do you intend on using this solution in your operations?"*
P1	"I am going to, if the engagement allows it."
P2	"I will give it a try if it's available for my operations."
P3	"Definitely."
P4	"Possibly, at least I will try it a handful of times since it requires little effort."
P5	"Yes."

Table D.10: Full survey participant responses to Q9 in the pragmatic quality evaluation survey.