FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Recovery of Software Architecture from Code Repositories

Ricardo Jorge de Araújo Ferreira

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Filipe F. Correia

October 30, 2022

Recovery of Software Architecture from Code Repositories

Ricardo Jorge de Araújo Ferreira

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Jácome Miguel Costa da Cunha

External Examiner: Prof. Florian Rademacher

Supervisor: Prof. Filipe Figueiredo Correia

October 30, 2022

Abstract

Architecture can result from multiple intangibly connected parts spread across source code and other development artefacts, making it difficult to describe the architecture without resourcing to additional documentation that puts this information together. Most of the time, this documentation is manually created, rendering it a costly process that, over time, starts to be disregarded, and the documentation becomes outdated and sometimes obsolete. There is significant inertia in producing and keeping the existing documentation updated, resulting in missing documentation, and even if it exists, it cannot be trusted.

Approaches to producing parts of the documentation in automated ways exist. However, finding a set of tools general enough to be used without effort is no easy task. Most existing tools have been built to analyse projects that obey certain conditions and formats. Moreover, the tools are built on a paradigm of "run once - produce once", meaning that the documents produced are final. The result cannot be changed and reused in subsequent runs. Imposing an extra burden on users that must, on every run, correct and complete the information that the tool is producing.

In this thesis, we explore the fact that infrastructure frameworks, like docker, terraform, and others, have well-defined structures and artefacts to propose a tool, which we name Infragenie, to generate infrastructure diagrams ready to be used in the project's documentation. By analysing these infrastructure artefacts, the tool tries to capture all the relevant elements while allowing manual changes and annotations, saved and reused on subsequent runs of the tool, generating diagrams that can evolve with the project with minimal effort. By automatically producing infrastructure diagrams and allowing the users to complete and improve the diagram with the certainty that those changes will not be lost, the documentation becomes a living part of the project. The developed tool is also compliant with the UML specification, something that is difficult to find on other tools.

To understand the usefulness and intention to use this tool, we conducted an empirical study on a heterogeneous sample of projects from GitHub. Infragenie was used to analyse the projects, and the result was sent to project owners in the form of pull requests where they were also invited to answer a short survey about the tool. The results supported the answers to this thesis's research questions and helped confirm that there is a willingness to keep the documentation updated if that is not a significant effort. Furthermore, auto-generating diagrams lead to an increased adoption, specially if the diagram is showing relevant and complete information. More than 55% of the enquirees were willing to use the developed tool and the produced diagrams in their projects. Also, editing capabilities on this type of tool are a much-appreciated feature. However, Infragenie is still in an early stage, as appointed by some users, and the fact that human intervention is needed was one of the factors in reducing the adoption of the tool. Nevertheless, the tool has room for improvement and to help projects to have Self-documenting Architecture.

Keywords: Infrastructure, Architecture, Automated documentation, UML

Resumo

A arquitectura de um projecto pode ser constituída por várias partes conectadas intangivelmente, espalhadas pelo código-fonte e outros artefactos de desenvolvimento. Isto impõe uma dificuldade na descrição da arquitectura, obrigando à consulta de recursos adicionais para reunir toda a informação. Na maioria das vezes, esta documentação é criada manualmente, um processo penoso que, com o tempo, passa a ser desprioritizado levando a que a documentação fique desactualizada e obsoleta. Há uma grande inércia em produzir e manter a documentação, resultando na falta de documentação e quando existe esta não é confiável.

Existem abordagens para produzir partes da documentação de forma automatizada. No entanto, encontrar um conjunto de ferramentas gerais o suficiente para serem usadas sem esforço não é fácil. A maioria das ferramentas existentes foram desenvolvidas para analisar projetos que obedecem a formatos específicos. Além disso, as ferramentas são construídas com um paradigma de "run once - produce once", o que significa que os documentos produzidos são finais. O resultado não pode ser alterado e reutilizado em execuções subsequentes. Impondo um ônus extra aos utilizadores que devem, a cada execução, corrigir e completar as informações que a ferramenta produz.

Nesta tese, exploramos o fato de que frameworks de infraestrutura, como docker, terraform e outras, possuem estruturas e artefatos bem definidos para propor uma ferramenta, a que chamamos Infragenie, para gerar diagramas de infraestrutura prontos para serem usados na documentação do projeto. Ao analisar esses artefatos de infraestrutura, a ferramenta tenta capturar todos os elementos relevantes, permitindo alterações e anotações manuais, que podem ser reutilizadas em futuras execuções da ferramenta, gerando diagramas que evoluem com o projeto com o mínimo de esforço. Ao produzir automaticamente diagramas de infraestrutura e permitindo que os utilizadores os completem e melhorem, com a certeza de que essas alterações não serão perdidas, a documentação torna-se uma parte viva do projeto. A ferramenta desenvolvida também é compatível com a especificação UML, ao contrário de outras ferramentas.

Para aferir sobre a utilidade e intenção de uso da ferramenta, realizamos um estudo empírico numa amostra heterogênea de projetos do GitHub. A Infragenie foi utilizada para analisar os projetos, e o resultado foi enviado aos proprietários na forma de pull-requests, onde também foram convidados a responder a um questionário sobre a ferramenta. Os resultados apoiam as respostas às questões de pesquisa desta tese e ajudaram a confirmar que existe intenção de manter a documentação atualizada se isso não for um esforço significativo. Mais ainda, diagramas auto-generados levam a uma maior adopção, especialmente se o diagram mostrar informação relevante e completa. Mais de 55% dos utilizadores mostraram interesse em utilizar a ferramenta e os diagrams produzidos. Além disso, os recursos de edição neste tipo de ferramentas são um recurso muito apreciado. Contudo, a Infragenie ainda está em fase inicial, conforme apontado por alguns utilizadores, e a necessidade de intervenção humana foi um dos fatores para a redução da adoção da ferramenta. No entanto, a ferramenta tem espaço para melhoria e apoiar Arquiteturas auto-documentadas.

Acknowledgements

This dissertation would not have happened without people I hold dear supporting me along the way, to whom I am deeply grateful. First and foremost to, my wife Marta, who supported this (almost insane) idea of starting a new course after leaving the academia so many years ago. She ensured the balance I needed to dedicate myself to the course while ensuring I had a life outside of this. She was always there, reminding me about deadlines and encouraging me to keep going but also remembering me about family time. Thank you for being there. To my five-year-old son, Benjamim, who was only one when this adventure started. Despite hearing so many times, "I cannot play with you right now", he always understood (at least for a couple of hours). He was responsible for forcing me to take a break from time to time and keeping me sane.

Finally, but not less important, to Prof. Filipe Correia, who challenged me to do this work. But most importantly, he was always available to guide and help me along the way. He helped overcome all the bumps this path had, like changing the theme after six months of work. Prof. Filipe was much more than a supervisor. He was my north star and a friend, always guiding and motivating me, helping to find the best solution to my problems and never putting any kind of pressure despite the delays that my professional commitments imposed. He had the kindness to always adapt his schedule to mine to ensure that we meet at least once a week. We had valuable discussions not only about the things gathered in this dissertation but also about methodology, research and engineering in general. I am a much better engineer now than the one I was. This more than a year and a half journey wouldn't be possible without him, and I am very grateful for that.

Ricardo Ferreira

*“Only those who will risk going too far
can possibly find out how far one can go.”*

T.S. Eliot

Contents

List of Figures

List of Tables

Abbreviations

ADT	Abstract Data Type
API	Application Programming Interface
CSV	Comma separated values
FEUP	Faculty of Engineering of the University of Porto
MEIC	Master in Informatics and Computing Engineering
MIEIC	Integrated Master in Informatics and Computing Engineering
MSc	Master of Science
Qx	Question x
TAM	Technology Acceptance Model
RQ	Research question
UML	Unified modeling language

Chapter 1

Introduction

1.1 Context

Design decisions for a software product are described by its architecture, defined in line with the business needs and significantly impact its quality. Ensuring that the implementation abides with the design decisions should be a continuous task during the software development life cycle and maintenance phases. This continuous evaluation is a challenge, requiring extensive communication and collaboration between architects, developers and other stakeholders, leading most of the time to late evaluations [18]. Using software architecture models can improve this collaboration, providing abstract descriptions of the system that are quickly understandable. Furthermore, these representations help to cement the knowledge of the architecture.

However, software architecture models or diagrams will eventually become outdated like other manually generated artefacts. This will become even more evident with the evolution of the project, with the introduction of changes that are not captured in the diagrams, lack of documenting habits or even lack of knowledge to do the necessary updates [1]. Also, there is not only one way to produce these artefacts and include them in the documentation. They can follow standard specifications or have free and flexible structures, which could difficult their analysis, as stated by Reinhold Plösch et al. in [41]:

> "Especially for software documentation written in natural language, it is often difficult to precisely specify quality requirements and to systematically evaluate its quality with respect to defined quality characteristics."

Jie-Cherng Chen et al. in [10] show that the most significant impacts on software maintainability are: *documentation obscure or untrustworthy*, *documentation inadequate, incomplete or non-existent*, *documentation not consistent* and *documentation lacking traceability*.

Give the need and importance of co-evolving related documentation artefacts, automatically generating them is a way to promote consistency and good documentation practices [11]. Due to the repeatable nature of automatic processes, the artefacts produced by such methods will enforce a specific pattern and consistent representations turning the interpretation easier. In fact, tooling to automate the recovery of architectures, producing models and diagrams to document the architecture exists [24]. Although, they tend to have limited applicability and be challenging to use. They often do not comply with representation standards like UML, C4Model and others whilst offering proper notations which require auxiliary materials to understand them.

1.2 Motivation

A lot of works have delved into how to best design and architect software systems [19, 9, 31]. With the popularity of cloud computing [45, 7, 25], some have more recently focused specifically on cloud-native systems [8, 61, 52, 53, 54, 55, 56, 2, 29]. A common approach in the context of such systems is the adoption of tools to describe infrastructures as code [51, 34, 44, 43]. Tools like docker-compose, ansible, chef cookbooks, terraform, and others define a well-structured set of artefacts used to describe the architecture. These artefacts usually live alongside other parts of the source code, in code repositories. We see new opportunities in the widespread use of such tools for mining software repositories and, namely, for recovering software architecture. Automating such a process would allow to keep track of the changes introduced during the different development cycles. Combining these processes with standard representations would produce valuable pieces of information that can be used to document the application and improve the overall knowledge.

Finding a tool or a set of tools, well accepted by the software development community, which are easy to use and suitable to be included in recurrent practices, will remove the burden of documentation and become just another automated step in the development pipeline. Besides this, if the tool provides a simple process to allow changes to the generated artefacts it will render those diagrams more complete. Although such changes can lead to inconsistencies between the source code and the diagram, such manual changes may render the diagram more readable and human-oriented, showing details that make sense for the development team. Therefore, we expect that the value of the tool and diagram would increase. Despite increasing the knowledge of the system, such a tool can also potentiate more informed changes in the architecture and accelerate the development and change.

1.3 Objectives and contributions

The main objective of this thesis is to contribute a tool, named Infragenie, that will:

1. extract architectural-significant information from code repositories, namely from resources such as docker-files;
2. use of the extracted information to synthesise architectural diagrams that will be kept in sync with the code repositories;

3. support the mechanisms that will allow a team to supply additional details to the architectural model that cannot be inferred directly from the repositories.

We will present an approach to developing such a tool, using as background the state-of-the-art review of available methodologies and tools where their weaknesses and strengths have been leveraged.

To evaluate the value of the tool, this thesis also contributes with an empirical validation to assert on the tool value, acceptance by the software development community and willingness to use. We also explore the benefits brought by the tool and if the ability to include changes in the auto-generated artefacts is, in fact, increasing its adoption. More specifically, this work will try to answer the following questions:

- **RQ1:** *To what extent does the adoption of Infragenie lead to better and updated documentation?*
- **RQ2:** *To what extent does the adoption of Infragenie improve the knowledge about the system architecture?*
- **RQ3:** *What is the value and willingness to use Infragenie?*

Thus we expect that this tool leads to better and updated documentation, enhanced by the simplicity of usage and reduced burden on the team members. We expect to find evidence that a tool like the one proposed can be part of the development pipeline, seen as just any other step in the product development cycles.

1.4 How to read this dissertation

The target audience for this work is software engineers, architects, developers, or anyone involved in the definition and implementation of software architectures. We also target this work to researchers.

The remaining of this document is organized into six chapters. These are:

- Chapter 2: Gives a quick background on software architectures, architectural models, and software documentation practices. It can be skipped by experts in those subjects.
- Chapter 3: Presents an analysis of the current state of the art around automated tools, methodologies and processes to automatically recover architecture artefacts like models, diagrams and others. Focusing on leveraging the pros and cons of each method.
- Chapter 4: Describes the problem that this work tries to solve and formulate the thesis hypothesis. Concludes with the methodology to prove the hypothesis and the estimated work plan.

- Chapter 5: Presents the proposed tool, the development approach, its architecture and principal features.
- Chapter 6: Describes the empirical study conducted to evaluate the proposed tool and validate the thesis hypotheses by finding answers to the research questions formulated in Chapter 4.
- Chapter 7: Gives a quick overview of this work and proposes improvements to the tool and additional enhancements to add to future replications of the study.

Chapter 2

Background

This chapter introduces Software architectures and Software architecture models. It also introduces the concept of Living documentation, showing why it is one of the motivations for the usage of architecture recovery tools and describes the role of DocOps in automating documentation creation. The chapter can be skipped by experts in these subjects.

2.1 Software architectures

Finding a good and succinct definition for software architectures is difficult because the scope is large [15]. The book Building Evolutionary architectures uses the following quote:

> "the important stuff (whatever that is)"

This quote clearly shows how broad the definition is. But, at the same time addresses the relevance of architectures in software products and the role of architects in describing all the components of a software product.

Many definitions can be found in the literature or even with simple web searches. Len Bass et al. in [5] proposed the following:

> "Software architectures must live within the system and enterprise, and increasingly it is the focus for achieving the organization's business goals.They are concerned with major elements taken as abstractions, the relationships among the elements, and how the elements together meet the behavioral and quality goals of the thing being built."

This definition provides a structural aspect of software architectures, showing that architectures are not only a model of the system but also the principles that guide its design decisions.

Another worth mentioning definition is the one used by Banani Roy et al. in [47]:

> “A software architecture is an abstraction of the runtime elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.”

Showing that architectures are abstractions of the system, hiding some details to identify its properties better. Complex systems will contain many levels of abstraction, described by architectures, guiding the implementation of the different interfaces of each element. This level of abstraction can lead to different architectures for the same problem, each bound to subjectivity and interpretation. There is no such thing as inherently good or bad architecture. Architectures are either more or less fit for some purpose [5].

2.2 Software architectural models

A model is a representation that shows information about a system. In the same way, an architectural model can be seen as a visualization of a software architecture viewpoint, expressing its structure and design [5].

An important distinction to be made is between diagram and model. While a model can be represented using a diagram, a diagram alone is not a model [5]. A model is aimed to provide complete and detailed information to describe the modelled viewpoint. The information cannot be vague or subject to interpretation. It should be rigorous in a way that two different people will have the same understanding. Here, the usage of standards plays an important role, in the sense that they allow communication with everyone that knows the standard [33]. Some accepted modelling languages in use are UML and the C4Model.

A model should avoid aggregating too much information despite rigour and completeness being essential in an architectural model. For that reason, it is often better to break a model into different components, which can be diagrams, instead of modelling everything together. This is another crucial aspect of a model that sets it apart from a diagram. A model can use different elements aggregated, not necessarily similar, in a package to provide the full view of the architecture.

To sum up, an architectural model provides clear, rigorous information about an architecture, usually fully describing the architecture. Views of single components and their interactions, while possible to be represented by a model, are more representations of design decisions than of the software architecture.

2.3 DocOps: Applying agile and automation to documentation

Time and quality affect all aspects of the software development ecosystem. Delivering new features faster is considered a competitive advantage [58]. To tackle this challenge, a DevOps culture aimed at automating repetitive tasks and defining processes is gaining popularity in the software development ecosystem. In the same way, other "*Ops" cultures are being integrated, focusing on different aspects, like technical writing, dealt with by DocOps.

DocOps promote collaboration and continuous improvement and integration of documentation. Jodie Putrino provides a good definition of a DocOps responsibility in its online article about DocOps[1].

> “a set of practices that works to automate and integrate the process of developing documentation across engineering, product, support, and, of course, technical writing teams.”

Expanding on this definition is safe to say that DocOps promote agile technical writing best practices, aligning technical writing with the business needs.

2.4 Living documentation

The term living documentation is presented by Cyrille Martraire in his book *Living Documentation: Continuous Knowledge Sharing by Design* [32] and can be resumed by one of the first quotes of the book:

> “Make very good documentation, without spending time outside of making a better software.”

This remit to the usage of methodologies, tools and other mechanisms to enforce the practice of having valuable, updated documentation that does not suffer the fate of becoming obsolete. The documentation evolves in line with the project, constantly keeping track of changes and maintaining its relevancy. Another important aspected of this concept is depicted in the following quote, also from the book:

> “ Living Documentation makes your code, its design and its architecture transparent for everyone to see. If you don’t like what you see, then fix it in the source code.”

The documentation of a project should be aligned with the project and be retrieved automatically so that the developers do not need to think about writing it. Moreover, achieving this Living documentation status, the documentation can act as the source of truth for the project, so well integrated that it could even allow for detecting problems and taking corrective measures.

Living documentation is an approach to producing documentation in an agile way, providing accurate, rigorous, easy-to-understand and constantly updated information.

[1]DocOps article in Write the Doc blog: `https://www.writethedocs.org/guide/doc-ops/`

Chapter 3

State of the art

To document a software architecture and keep it up-to-date is often complex and costly. Nevertheless, documentation is a fundamental artefact to understand the software and to improve how it is communicated and maintained [28]. Techniques to automatically or semi-automatically generate documentation are often of great value, especially for medium and high size projects. This chapter will explore approaches and methods used to automate software architecture documentation. The analysis will start with an overview of common problems and challenges in this space, then with an analysis of some software architecture recovery techniques and continues with a more thorough analysis of some tools and artefacts with the objective to answer the following questions:

- *Q1: What approaches are there to recover software architectures from the contents of their repositories?*
  - What type of files are used?
  - What type of models and/or diagrams are generated?
  - What are the models and/or diagrams used for?
- *Q2: What approaches are there for automatically documenting the architecture of a software system?*
  - Which aspects of the architecture are covered?
  - Are they based on models?
- *Q3: What tools are there to provide a visual representation of the architecture based on orchestration artefacts?*
  - What artefacts have been used? And why?

For that, a literature search was conducted on the following platforms: Google Scholar, Google default search, Scopus, IEEE Xplore and Elsevier. Some of the search queries included: *"self-documenting architecture"*, *"extract architectural information"*, *"living documentation"*, and *"software architecture reconstruction"*. As for the analysed tools, the search was in GitHub, GitLab and Google, and these tools were tested manually to check their functionality.

This chapter is the starting point to understand what is being done and how should we start our approach to solve the challenge proposed by this thesis: automatically create a way to visualize the architecture of a software using orquestration artefacts with the possibility of manually updating those visualizations.

3.1 Software Documentation Challenges

Software documentation, when properly produced, is a valuable tool to provide knowledge about the system and its related processes [1]. Documentation can be used to describe the system and detail updates and changes, to perform maintenance activities, among others. Furthermore, good documentation is proven to improve the performance of engineers and even documents that are not up to date may still provide valuable information [49] .

Despite the value of documentation, producing it is surrounded by many issues that affect its quality and usability. Statich et al. [49] conducted an empirical study to understand common problems and solutions of software documentation, and they concluded that the most common issues are:

- *Documentation being out of date.*
- *Documentation not being traceable to the changes made in code.*
- *Lack of standards and effort needed to update documents.*
- *Lack of tools for documentation maintenance and change tracking.*

Another interesting conclusion of this study is the factors determining software documentation significance. The authors state that documentation significance is determined by the size of the project, the experience of the persons who will use the documents, geographic distribution of teams and the tools used to access, search and navigate the documentation. In sum, documentation should be adapted for every project.

Addressing all the issues is no easy task, and it is where tools that automate the documentation process are introduced. Tools created under the unbrella of concepts like Living Documentation [32] or, perhaps closer to the topic of this thesis, Self-documenting Architectures [59]. Applying these concepts means that the software documentation will be constantly updated to the point that even simple changes to the project artefacts could generate updates without effort. Also, these methodologies will help reduce the learning curve for development, expose bad decisions and help to understand the impacts of changes.

Build tooling to automate documentation is not trivial and can impose significant challenges on the teams and organizations. The rest of this chapter will look into existing tools, methodologies and frameworks to achieve a satisfactory level of documentation automation, with a particular focus on the architectural and infrastructure spaces.

3.2 Software Architecture Recovery

Software maintenance represents a considerable effort in a product lifecycle. Moreover, this process is challenged by architectural erosion and technical debt [50, 6]. Development iterations will increase this problem if not adequately planned and documented, which is often the case, and sooner or later, developers are forced to employ architecture recovery techniques [20].

Architecture reconstruction is a reverse engineering activity used to recover lost or unknown aspects of software architecture, and indirectly a means to document the architecture [24].

Software architecture recovery helps to reconstruct documentation from system available artefacts, such as source code and, especially for legacy systems, are the only way to produce knowledge about the software [62].

Some techniques have been proposed to recover software architecture, [3, 20, 28, 37]. They can be classified as automatic, manual or semi-automatic. Of course, manual recovery techniques are not of great interest for this work and such, they will be left out of this analysis.

One way to recover architectures is using clustering methods, which simplistically are nothing more than group together pieces of related information in the so-called clusters [60]. There are two common ways to identify clusters in software systems:

- *Knowledge-based:* This approach tries to group functionalities in the source code, for example, modules, complementary functionalities, libraries and others.
- *Structured-based:* In this approach the software is decomposed based on the flow of data, looking into interactions between entities.

Although both approaches are valid and can lead to good results, for large systems, the Knowledge-based approaches do not perform well, and such structure-based techniques are more popular [60].

Clustering is a documented and structured approach followed by some authors. However, most of the time, due to the singularities of the project and recovery objectives, the best approach is to develop tailored methodologies. In general, these methods return better results than clustering. After all, they have been put together to respond to a specific need.

Most of the tools and frameworks to recover or reverse engineering architectures are born from the need to have models or diagrams of the software to help understand the impact of future changes, to plan evolutions of the product, to keep documentation updated at low costs, or to acquire more knowledge about the system. These were the motivations behind the approach proposed by Claudio Riva [46]. His work is a good starting point for the analysis carried out in this chapter. Instead of a tool, Claudio proposes a flow with tasks to accomplish his recovery objectives and chooses some tools to help with those tasks. This methodology shows the diversity of ways in which it is possible to extract architecture models from software systems. The approach taken is depicted in the diagram in Figure 3.1.

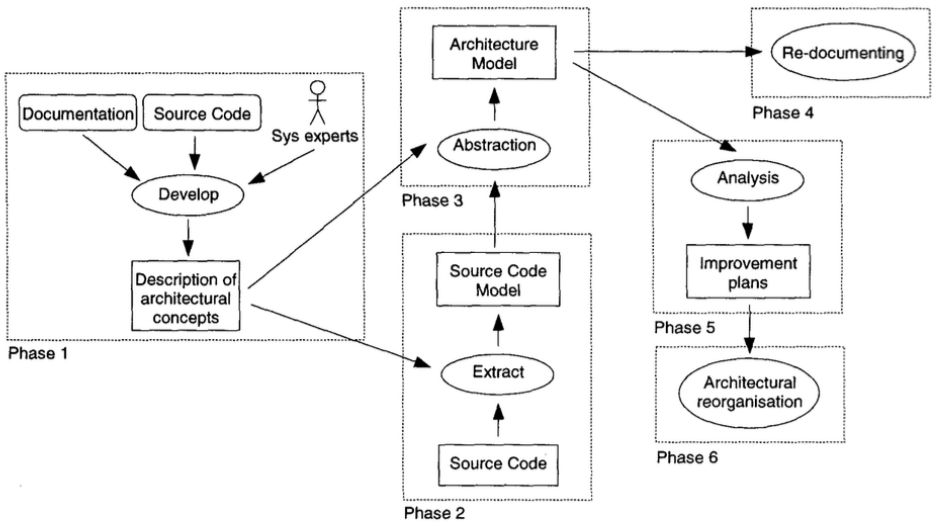


Figure 3.1: Reverse architecting process flow chart proposed by Claudio Riva [46].

The flow is divided into six different phases. The first phase is probably the most important one because architectural concepts are defined. Meaning, in this phase, the author defines what the architecture is and its expectations and expected results from the recovery process, guiding the following steps and the tools to use. The second phase consists of extracting the source code model, which can be done using automatic tools. Then, phases three and four are assisted by manual intervention and consist in identifying abstractions taken during development that the automated tools are not capable of modelling, resulting in a refined model that can be used to update the documentation. The last two phases exist to evaluate the results and reorganize the model and the source code. Although this flow relies heavily on human interaction can be seen as a good set of stages that a fully automated tool needs to do and will be the base to analyze the tools on Table 3.1. The analyzes will be divided into the following subjects, type of analysis, the object of the analysis, level of automation, inputs and outputs.

3.2.1 Type of analysis

Software architecture recovery can be applied using static, dynamic or a mix of both analyses. Most of the tools in Table 3.1 use static methods or a mix of both.

Usually, static analysis is faster and requires less effort. The projects files are usually enough [36]. On the other hand, dynamic analysis is more precise [4] but requires a running environment which can sometimes be difficult to set up.

Each type of analysis is employed with different goals in mind. The tool MicroArt [21] for example, uses static analysis to recover a model of the physical architecture, where the components

Table 3.1: Software architecture recovery tools and frameworks.

Tool	Analysis	Technologies	Inputs	Outputs	Allow changes	Manual steps	Limitations
MicroArt [21]	S & D	Java, Docker files, Logs	Source code	Component model	No	No	Running env., Technology-bound
X-Trace [17]	D	Java, C/C++, Ruby	Metadata	Network model	No	No	Running env., Needs metadata
Moose [14]	S	Pharo	Metadata	Queriable model	No	No	Learning curve, Need extra tools, Needs metadata
QAR [12]	S & D	-	Documentation, Source code	Class model, Component model	Yes	Yes	
Focus [13]	S	-	Source code, Arch. model	Class model, Component model, Sequence model	Yes	Yes	Needs Arch. model
Armin [36]	S	-	Documentation, Source code	Class model, Component model	Yes	Yes	Needs extra tools

Analysis types: S - Static, D - Dynamic.

are identified, then with dynamic analysis, the tool makes a service discovery identification to map services, understand their interactions and IP addresses enriching the model created during the static analysis. QUE-es Architecture Recovery (QAR) [12] also joins both approaches, using the static analysis for a similar purpose, creating a preliminary view of the architecture, but on the dynamic analysis tries to extract information about the call sequence of the participating classes and the sequence of operations performed in the application. In Table 3.1 there is only one tool that uses exclusively dynamic analysis, X-Trace. This tool tries to recreate the system's physical network, understanding which services are in the network, the flow of data among the services, and it goes one step further, detecting anomalies and failures in the network. This tool can live along with the services being analysed, providing a representation of the network architecture and live monitoring of the services.

It is worth noting that static analysis can be employed using clustering techniques like the ones mentioned at the beginning of this section. Even though they are challenging to use, they can be helpful for systems with a considerable degree of abstraction and legacy source codes [20, 30]. The primary purpose of clustering is to form groups of entities that are similar to one another and then relate those groups based on interactions [60]. The analysis of the clusters can then be combined with machine learning and searching algorithms like the Bunch tool, proposed by Mitchell et al. [35] to improve the final representation of the system. Despite its utility in legacy systems, looking into the comparison put together in [20] methods based on clustering show poor performances, with accuracies under 60%. Another downside of these algorithms is the artefacts used, which rely heavily on source code, which can be a problem for big-sized systems.

3.2.2 Object of the analysis

Analyzing a system to recover its architecture can be done solely with the information present on the project files or adding some extra information to specific files to help the tools understand the meaning of the information. This extra information, or metadata, is applied for different purposes and can help with static and dynamic analysis.

From all the tools on Table 3.1, Moose [14] is probably the one that uses metadata in the most peculiar way. Moose, more than a tool to recover architecture, is a tool to create a queryable environment of the system under analysis. At the end of the analysis, Moose creates a model that can be queried to get all kinds of information about the system using the Smalltalk language, such as getting all the methods that access a particular attribute or getting classes that have more than an arbitrary number of descendants. This powerful analysis is possible through specific metadata that needs to be applied to the source code. With such a powerful model of the system is possible to use other tools to create visual representations of the architecture, as some of the plugins and tools shown on the official moose website[1].

X-Trace [17] uses metadata to trace the information flowing in the network. For this to happen, all network protocols need to be modified to propagate the specific X-Trace metadata, which may require expert developers to do that. Also, adding this metadata to the communications can carry extra latency, a problem in some systems. Despite that, metadata usage brings much more knowledge to X-Trace, such as information about nodes, their parents, descendants and types.

The tools can be technology agnostic with the cost of polluting the source files with external entities by using metadata. Also, adding metadata can be a very complex task. To avoid changing the source files, the tools require a better knowledge of the system. As is often the case, one tool that analyzes systems built with technology A is not suited for technology B. To overcome this limitation, tools like QAR [12] and ARMIN [36] use other tools to perform a static analysis of the source code and then use that data as the starting point for the recovery. Without this trick, the tools usually are explicitly tailored to a set of technologies. They do not work with others, like MicroArt [21] which is described as an architecture recovery tool for Microservice-based systems, but the system must be containerized with Docker.

3.2.3 Level of automation

Software architecture recovery can be an entirely manual process, but that would be very tedious and time-consuming, especially for legacy systems with big code bases [28]. Automated or semi-automated tools can alleviate this task. However, there is a big concern when automating the software recovery architecture, the accuracy [28, 20]. This is due to the lack of ground truths and the difficulty in generating them. Thibaud Lutellier et al. in [28] conducted a comparative study on some architecture recovery techniques where the ground truth used took almost two years of collaboration between the application developers.

[1]Moose official website: `http://moosetechnology.org`

Semi-automated tools like QAR [12], Focus [13] or Armin [36] reduce the need for ground truths with manual operations that can happen in different stages of the recovery process. They are usually applied when the result of the automated step is very abstract or to generate inputs to conduct the operation of future steps. Focus, for example, uses as the starting point of the recovery a high-level model of the architecture, usually built by developers and experts of the application. This tool is highly iterative, starting with the high-level model, applying tools to analyse source code and files and then using that information to refine the first model. This approach gives some freedom to what should be recovered and to what degree since the automated steps can be applied to specific parts of the architecture.

Similarly, Armin has a starting point for its analysis, generated by automated tools but then manipulated manually so that errors are not propagated to the end models. Also, QAR combines manual steps to improve accuracy. It is a workflow with three activities, documentation analysis, static analysis and dynamic analysis. Each of the activities involves human intervention. For starters, documentation analysis is a purely manual activity to extract information. At the end of each analysis, static and dynamic, again, human intervention is used to extract the views, validate them with expert and developers information and finally generate the final model.

Looking at the fully automated tools on Table 3.1 it is possible to see that X-Trace [17] and Moose [14] have only achieved this level of automation using metadata, which acts as a co-pilot for all the analysis. MicroArt [21] despite not having metadata, is limited by the technology. In other words, this tool is fully automated because it was built specifically to analyse a specific type of project that follows some standards.

Achieving fully automated software architecture recovery is not an easy task, and most of the time, the accuracy is not high [27] but satisfactory enough to develop methodologies and tools, especially for extensive systems with thousands of lines of code.

3.2.4 Inputs and Outputs

The inputs and outputs of a software architecture recovery tool or methodology depend on the objective of said recovery. Also, the inputs and outputs will determine the tool or methodology to follow.

Analyzing the tools on Table 3.1 its possible to see that the source code files are the most common input used. After all, the application code is spread among these files. However, other than that, different information can be used and with different goals. QAR [12] uses documentation to extract a conceptual model of the system whereas Armin [36] uses documentation to refine the reconstructed views. MicroArt [21] analyzes docker files to understand the services involved in the application and extract information from log files of the communication among each service to track communications and enrich the generated model, with things like IP Addresses.

Metadata is a particular input type, consisting of blocks of information, keywords or even functions, spread across the files under analysis. It annotates the information to be easily identified or provides special features like improving the logged information while the application is running or including tokens in the system calls to facilitate the dynamic analysis. X-Trace [17] and Moose [14] are using metadata with these objectives.

In the same way, different tools will output their results in different forms and may use external tools to extend their final models. One such tool is Moose, which generates a queryable model of the system that can be used as information for several other tools to create visualizations, parsers, scripts, and many others. X-Trace generates a model of the network that allows maintenance activities since it represents malfunctioning nodes and communication problems, and MicroArt [21] generates a component model with proper notation. The other tools on the table, QAR, Focus and Armin, allow different models since some of the recovery steps are performed manually. This gives liberty to generate different models and to choose to follow standards in the representation, like UML[2] or the C4Model[3].

One interesting aspect of the outputs is its freedom to introduce changes, annotations or corrections that can then be used in future analysis. From the tools studied in Table 3.1 only the tools that produce its final model manually are classified with allowing changes, precisely by the fact that the final representation is manual. The other tools are just producing a final model. The inputs should change to generate new models, and a new analysis should be executed.

3.3 Tools for infrastructure visualization

In this section, we will explore some available tools to visualize the infrastructure of a system. We have focused this analysis on docker and terraform infrastructures. To properly evaluate docker projects, a fictitious e-commerce application[4] was used. The application uses a microservices architecture with different technologies like MongoDb, Redis, Spring Cloud and more. For terraform, the tools page provide enough examples to understand how the tool works and which outputs it provides.

We started this analysis with a tool called *docker-compose-viz*, available for free on github[5]. Running this tool on the evaluation project generates the diagram from Figure 3.2.

The tool correctly identified the services and their interactions in a few milliseconds. The diagram does not follow any standard, and as such, it is not easy to understand without going through the documentation. In this diagram, rectangles represent services or containers and circles the ports used to communicate with the service. Dotted arrows are used for dependencies and point to the service that declares the dependency. The representation directly translates the information in the docker-compose file, fetching no extra information. Also, this visualization cannot be changed or annotated to make it easier to read and enrich its content.

Docker-visualizer[6], is another tool that also tries to generate visualizations of the docker infrastructure, although with a very different notation like shown in Figure 3.3.

[2]More info about UML in `https://www.uml.org`
[3]More info about C4Model in `https://c4model.com`
[4]Evaluation project: `https://github.com/venkataravuri/e-commerce-microservices-sample`
[5]Project page of docker-compose-viz: `https://github.com/docker-compose-viz`
[6]Project page of Docker-visualizer: `https://www.npmjs.com/package/docker-visualizer`

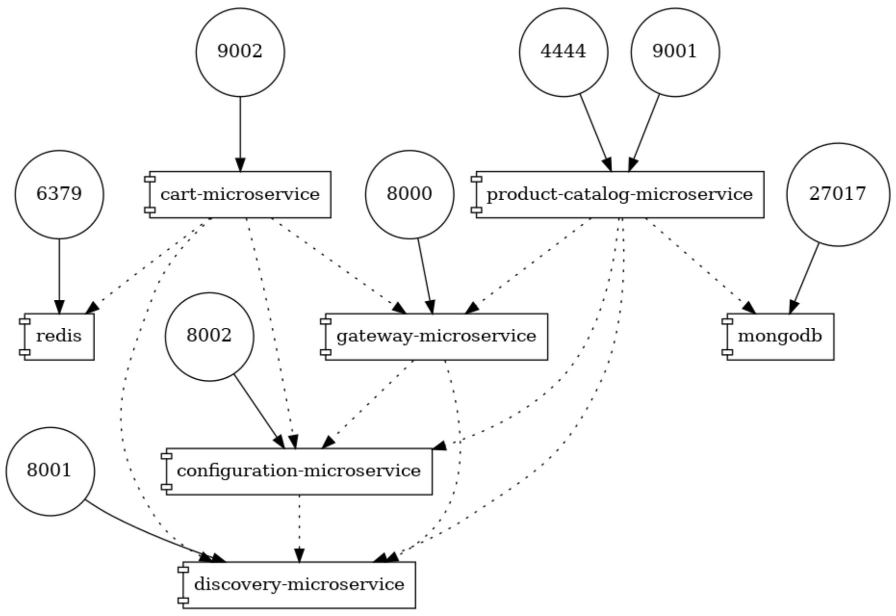


Figure 3.2: Docker visualization generated by docker-compose-viz.

The visualization provided by *docker-visualizer* is a bit more clumsy and straightforward. This visualization is just grabbing the information on the docker-compose file and creating boxes without precise representations of interactions and dependencies. It also always add the Root, Network and services boxes even if the docker configuration does not explicitly define them, like the case in the evaluation project, where the Network is not defined. Again there is no possibility of changes or additions to the generated diagram.

The previous tools are performing static analysis, and such, only the information available on the source files of the projects can potentially appear on the generated diagrams. *DockerViz*[7] is a tool that does a dynamic analysis of the docker infrastructure to try to get more information and help to understand what is happening in the system. Figure 3.4 shows the result of the analysis made by *dockerviz* in the sample project used. The tool correctly identified the services and marked the service that is not running in grey. Nevertheless, since the infrastructure was not all up, the tool could not represent how each service was interacting, generating a poor diagram.

However, it is interesting to note that if we combine static analysis tools with dynamic analysis, we get a richer model, with information on interactions, identification of the services in the network, and other pieces of information.

Another popular way to describe the infrastructure is with Terraform. Also, for Terraform is possible to find visualizers. For this comparison we have analyzed three terraform visualizers:

[7]Project page of DockerViz: https://github.com/justone/dockviz

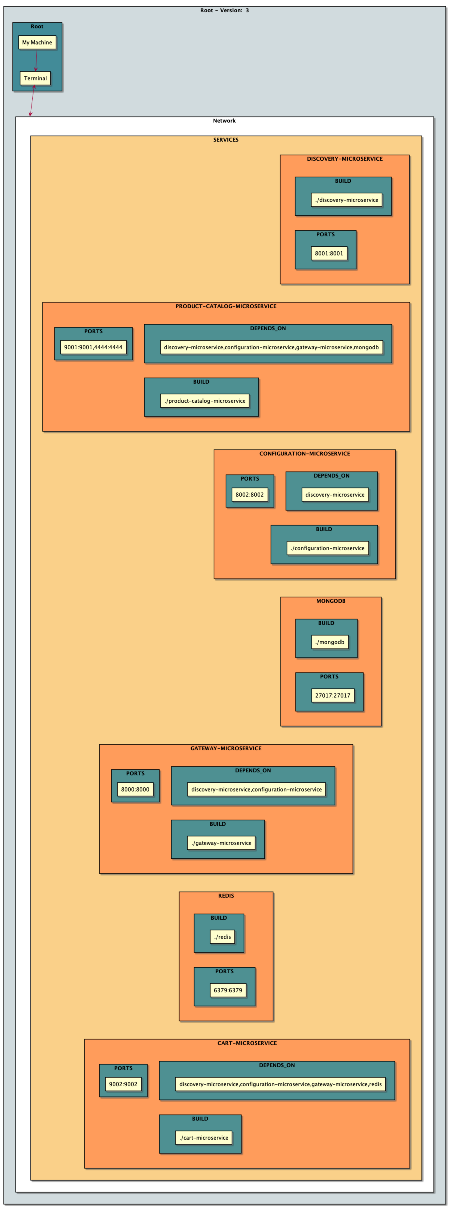


Figure 3.3: Docker visualization generated by docker-visualizer.

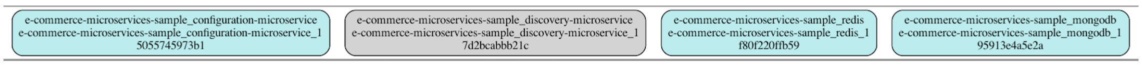

e-commerce-microservices-sample_configuration-microservice
e-commerce-microservices-sample_configuration-microservice_1
5055745973b1
e-commerce-microservices-sample_discovery-microservice
e-commerce-microservices-sample_discovery-microservice_1
7d2bcabbb21c
e-commerce-microservices-sample_redis
e-commerce-microservices-sample_redis_1
f80f220ffb59
e-commerce-microservices-sample_mongodb
e-commerce-microservices-sample_mongodb_1
95913e4a5e2a

Figure 3.4: Docker visualization generated by dockerviz.

Terraform Graph[8], *Inframap*[9] and *Terraform Visual*[10]. Despite being different tools, they work similarly, doing static analysis of the JSON terraform files and generating simple graphs, showing

[8]Project page of terraform Graph: `https://www.terraform.io/docs/cli/commands/graph.html`
[9]Project page of Inframap: `https://github.com/cycloidio/inframap`
[10]Project page of Terraform Visual: `https://github.com/hieven/terraform-visual`

the services in the network and arrows to represent their connections. One such diagram can be seen in Figure 3.5.

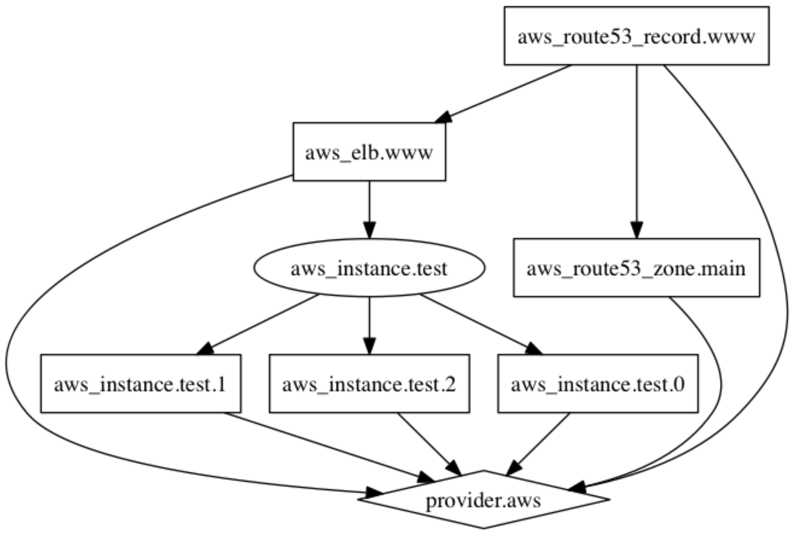


Figure 3.5: Graph representation of an infrastructure described with Terraform.

The diagram is elementary and has only basic information about the infrastructure, services, and connections. Also, the diagram does not follow any standard notation and does not allow any change or annotation. Once generated, the diagram is final.

Tools such as Argon [48] and Docker Composer [39, 40] also support a graphical representation of infrastructure specifications, but they do so with the goal of providing visual editors. As they are not concerned with helping to document an architecture, or an infrastructure, we have not covered in our analysis.

3.4 Discussion

We started this study with three questions that guided the rest of the analysis.

The first question (Q1) *What approaches are there to recover software architectures from the contents of their repositories?* To answer this we looked into Software architecture recovery methods, cf. Section 3.2, more specifically on the files used, the generated models and how they are used. For starters, the specificities of the project and the type of model required will guide the approach to follow. Table 3.1 is a summary of some of the methods studied and where it is possible to identify static and dynamic approaches with very different results. Also, the inputs used can vary profoundly, from raw source code to adding metadata to enrich the information extracted or using other more human-readable information like documentation or log files. The generated models can serve many purposes, documentation, planning future changes or even maintenance.

With question (Q2), we tried to find an answer for *What approaches are there for automatically documenting software systems architecture?* None of the analysed works was focused on documentation but rather on automating the recovery of architecture diagrams and models that could later be used in documentation. The level of automation is determined by the inputs and the level of detail needed in the generated model. Extracting information from files that follow a predefined structure is generally easy, and as such, the extraction process can be fully automated, like the approach followed by MicroArt [21] where the infrastructure information is extracted from docker files. Inputs with free structures can be challenging to automate, and in those cases, human intervention is needed to identify and describe some of the abstractions extracted by the automated processes, like in the Focus method [13]. Even though full automation can be accomplished for almost all projects, the effort needed may not be worthy, or the tool developed can be only used with one type of file and project, imposing challenges to put together a method that automatically generates documentation.

The last question was (Q3) *What approaches are there to providing a visual representation of the architecture based on orchestration artefacts?*. The approaches used in software architecture recovery gives us a good overview of what is needed to extract information from a project. Furthermore, we have observed that using structured inputs will ease the analysis and lead to higher levels of automation. In general, orchestration artefacts are structured files, the ones used to define, for example, docker, terraform, ansible, chef cookbooks or others. Tools to visualize the infrastructure are available, and some are free to use, cf. Section 3.3. Most of the tools perform static analysis since the orchestration files already have a good definition of the infrastructure. However, with dynamic analysis is possible to go even deeper and get a richer visualization in the end. Dynamic analysis, however, imposes a bigger setup effort, and if the setup is not properly executed, the analysis tends to be incomplete and sometimes even useless.

From all the frameworks and tools analyzed in this study, it is important to refer that the models and visualizations generated rarely followed any notation standard, limiting their direct usage in the documentation since custom notations need extra documentation to describe them. Also, the models and diagrams are final, meaning that no extra information can be added or changed after the extraction and generation. This aspect limits the usage of these methods and tools to constantly evolving projects since the whole process needs to run from scratch every time, and the extraction will always output the same abstractions. Furthermore, if the methodology or tool cannot extract part of the information, that means that the information will be lost between each run. So, the direct usage of the models and diagrams in documentation should be taken with care.

Chapter 4

Problem statement

The previous chapter presents methodologies and tools to recover architecture from project repositories. Most of the tools generate accurate representations of the parts of the system that were identified, following the expectations of their authors, like, for example, class diagrams or infrastructure models. Although all the models generated are final. They do not allow changes or annotations and cannot be used as feedback to improve the quality of future models. This chapter elaborates on how this research contributes to **a) an approach to automatically generate documentation that can be extended with user annotations and modifications, which are not lost between analysis** and **b) a tool to recover architectures following what was specified on a**). This tool, which we called Infragenie and is described in detail in Chapter 5, allows the users to annotate the final model and modify it to describe abstractions that were not captured, transforming it into a valuable documentation artefact. This chapter also describes, on a higher level, the methodology followed.

4.1 Scope

Software architecture models are descriptions of the software architecture. They can represent the whole or parts of the software, following proper or standard notation. They provide a way to quickly understand the architecture through abstractions and visual representations, helping to design new features and undergo maintenance activities.

Software architecture models like other artefacts that are manually updated will eventually get obsolete [32] or become a burden to keep it updated as the software evolves. Many reasons exist for this, from different software teams working on the same project, new developers joining the team, different backgrounds and habits or even lack of knowledge to do the necessary updates [1]. However, the primary reason to put aside documentation is that it is boring and a great source of frustration. A quote from Gerald M. Weinberg in Psychology of Computer Programming describes this perfectly:

> "Documentation is the castor oil of programming. Managers think it is good for programmers, and programmers hate it!" [63]

This inertia to keep documentation and models updated results in documentation missing, most likely the needed documentation. If the documentation exists most of the time, it cannot be trusted because it is misleading and obsolete.

Better ways to deal with the documentation exist, in the form of automated tools like the ones analysed in Chapter 3. However, most of the tools available work on a run once, produce once paradigm, meaning that the tool will analyse the project repository producing a final model, which cannot be easily changed and used as feedback for subsequent runs. Although this small detail seems of low importance in the presence of a tool that produces a tiresome artefact automatically, the truth is that if the tool is generating models with errors, those models cannot be directly integrated into the documentation. Every run will continue to generate the same errors which need to be solved every time. Also, they are not fitted for every project and sometimes are difficult to run. Some of the tools needed complex setups and preparation of the repository to run, making it impractical to use on every development phase. It can be an even more significant burden to the development team. Most likely, the tool will be abandoned.

4.2 Thesis statement

We know there are better ways to deal with documentation, specifically with architecture models, which allow us to stop regarding it as a chore, wrapped in frustration and a feeling of useless work. This is where the concept of living documentation coined by Cyrille Martraire in his book *Living Documentation: Continuous Knowledge Sharing by Design* [32] steps in and from where we combine some of the concepts to put together our hypothesis, for which we set ourselves to provide evidence that:

> "Recovering the software architecture from the code repository can reduce the burden of creating and updating documentation while bringing more knowledge and feedback about the changes introduced. Furthermore, having models living alongside the software and allowing to add manual changes and annotations will render the documentation more consistent and usable."

Consequently, creating and using documentation can become a consistent practice of the development process, indirectly helping and motivating architectural changes.

This hypothesis mentions models living alongside the software, which can be ambiguous terminology. Nevertheless, this is simply a way of talking about models that can be easily **updated** to reflect the changes introduced in the product, like referred to in the book Living documentation [32]. This means that software changes will promote changes in the architecture models if needed and preferably in an automated way, giving an immediate **feedback** of what has been changed. This practice will keep **consistency** between software and models.

4.3 Research questions

Software architecture recovery is usually applied to specific moments of a project lifetime, most of the time in legacy systems where the knowledge about the system has been lost. This research aims to contribute with a slightly different approach, applying architecture recovery whenever relevant during the development cycle. To enable our contribution, we identify the following research questions (RQ):

RQ1 "To what extent does the adoption of Infragenie lead to better and updated documentation?"

We aim to understand if using the tool developed in this work to generate models of the system architecture on each development iteration contributes to updated documentation and how valuable that model is for the overal documentation of the product. Have we reduced information redundancy between code and textual documentation? Have we create a systematic action to update documentation, like any other steps from a pipeline?

RQ2 "To what extent does the adoption of Infragenie improve the knowledge about the system architecture?"

Developers often add changes to the software without a full comprehension of its architecture. Having a model of the architecture can improve the confidence and planning better future changes? Can the model guide developers to make better decisions and develop faster?

RQ3 "What is the value and willingness to use Infragenie?"

Finally, understanding the acceptance of the tool is crucial. Using models like the Technology acceptance criteria [22] we will try to find out if the tool is adding real value to the project and if the developers are considering the tool for other projects. We will also try to get feedback on what works best and what should be improved with a special focus on the feature that distinguish our tool, the ability to preserve user changes between recoveries.

4.4 Methodology

To provide evidence on this thesis hypothesis, we have designed the methodology in Figure 4.1

We start with a **review of the existing state-of-the-art** approaches for recovering software architectures to understand their relevance and purpose. In this analysis, we focus on tools and methodologies and how they recovered models that can be used as documentation artefacts for the project. The revision compares some of the tools used to produce software models, highlighting advantages and problems so we can **formulate a hypothesis** and design an approach to prove it.

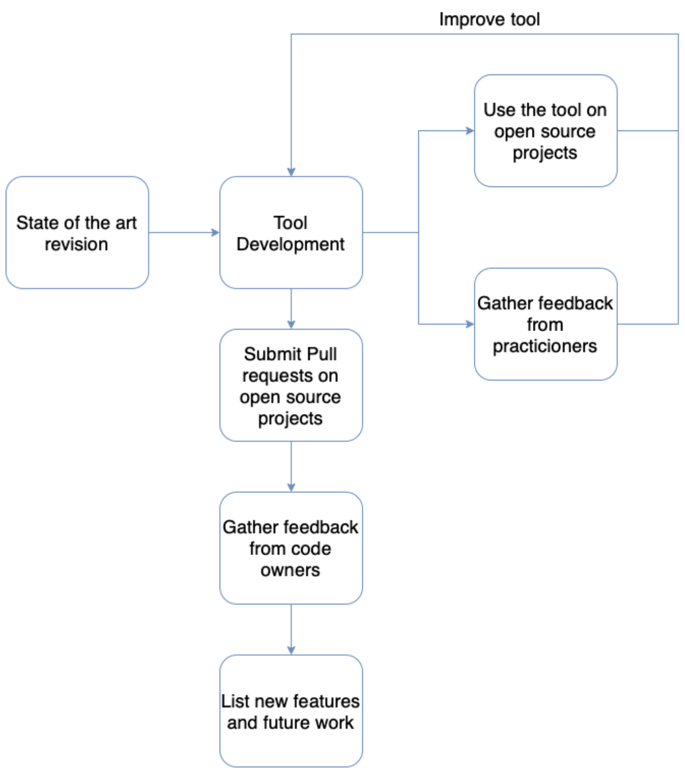

Improve tool
State of the art revision
Tool Development
Use the tool on open source projects
Gather feedback from practicioners
Submit Pull requests on open source projects
Gather feedback from code owners
List new features and future work

Figure 4.1: Steps of the methodology.

Following, we started our **case study**(cf. Chapter 6). The first step was to develop a **prototype** (cf. Chapter 5) with the knowledge acquired during the state-of-the-art revision. We then validated the tool with publicly available projects and started our first evaluation cycle. Were we **inquire** a group of practitioners to understand their willingness to use the tool, what features they value the most and which ones they would like to have. This step provided feedback to improve the tool and shed some light on finding an answer to the research questions, although still incomplete answers at this stage.

With the tool in a more mature state, we submitted pull requests on open-source projects and sent **structured questionaires** to the code owners. These questionnaires complemented the information obtained in the previous inquiry, providing answers to the research questions formulated. At the same time, we are trying to understand why the pull requests have been approved or disapproved. At this stage, the comments on the pull request were also used as feedback sources.

In the end, we had enough information to suggest new features to improve the tool and **validate the hypothesis**.

Chapter 5

Infrastructure recovery tool: Infragenie

As stated in the previous chapter, this thesis tries to address the problem of recovering software architectures using existent project artefacts. The recovery should be such that the architecture knowledge recovered can evolve with the development and provide the ability to add user changes. This last feature differentiates our approach from others already available, analysed in Chapter 3. The manual changes added should also be persistent so that if the architecture evolves, the manual additions are not lost. That means that the recoverer should understand which manual changes are relevant throughout the evolution of the project and decide if they should be kept or not. Another differentiator is the usage of the UML specification since using standards to describe the infrastructure is not common among the tools available for automatic recovery.

Our approach resulted in the development of a tool which we name Infragenie. This chapter will describe the tool, its architecture and how it addresses the problem. We describe some of the technologies used and why they were chosen. Ease of use was in mind through the entire process. In such a way, we end up developing an API that other tools can use. On top of this API, was developed a web application which allows a user to analyse projects and provides editing capabilities. All these elements are described in this chapter.

5.1 Approach

Our aim is to develop a prototype tool that can generate **architectural diagrams** that will **evolve** with the project and are **easier to generate**, **update**, and **annotate with extra information**. We believe such a tool will encourage project managers to keep their models updated and ensure their documentation is usable and a source of truth for the project.

On a high level, our approach to solving the problem under analysis is to generate the diagram only with one workflow, which could stop after generating the diagram or continue to provide editing capabilities. This workflow has the following steps:

1. **Finding and extracting infrastructure artefacts**: Using the fact that popular infrastructure frameworks use naming conventions (such as docker-compose), it is possible to search for specific files in the project and read their content. Also, this content obeys a structure, which defines services, ports, interactions and others.

2. **Apply manual changes**: This step is skipped without manual changes. If manual changes are found, the changes are applied in order. The recovery works like a version control system mixed with an event sourcing methodology where all the changes are applied until the latest.

3. **Provide a visual representation of the infrastructure**: With the infrastructure content, a set of PlantUML instructions is generated to provide the representation. Only the infrastructure in its latest state (with all the manual changes) is represented. Since we are using PlantUML, the UML specification is almost guaranteed out of the box.

4. **Persist manual changes**: If manual changes are introduced by a user and saved, the tool takes a snapshot of the current state of the diagram and saves it in the source code in a format compliant with Step 2.

This four-step workflow, represent the principals that we propose to develop the mentioned prototype. Also, this approach allowed a high level of abstraction, making it easy and fast to extend functionality and add new elements.

Following this workflow, we designed a tool that solves the requirements leveraged in Section 5.2 and is presented in more detail in the following sections.

5.2 Desiderata

At this point, it is essential to define the desired requirements for the tool under analysis. These requirements can be classified as primary and secondary. The primary requirements, the mandatory ones, are:

- *Finding infrastructure artefacts:* Given a project, finding the files where the infrastructure is defined should be possible. Usually, these files follow a naming and location convention, depending on the type of tools used.

- *Extract infrastructure information:* Ability to read infrastructure files and extract the pieces of information that define the infrastructure.

- *Provide a visual representation of the infrastructure:* Create graphical diagrams from extracted infrastructure information.

- *Allow manual editions:* The generated model should allow the user to include annotations and change and add elements.

- *Persist manual changes:* The changes included by users should have priority and be included in new versions of the generated model. For example, if a user manually changes the service name while that service exists on the infrastructure specification, the model will always show the said service with the name specified by the user.
- *Follow UML specifications:* The models should be generated and modified following the unified model language, specifically the deployment and component specification.

The secondary requirements, the ones that would bring an extra value to the final solution but can be removed without significant impacts, are:

- *Fast recovery:* Although fast is a very broad metric, since every person may have a different concept, the solution should provide a model and editing capabilities in a few seconds. Using this concept of fast is intentional since throughput and performance tests are not planned. Fast here only means having a usable solution.
- *Provide the model in different formats:* The ability to provide the graphical representation in different formats will bring more willingness to use, especially providing formats like svg, which can be used on various mediums without quality losses.
- *Automatically detect changes:* Once plugged into a project, the generated diagram can be automatically updated by detecting changes in the code.
- *Ability to represent using different standards:* Besides the UML specification, the user can choose different standards, like the C4Model, so that the diagram can be better integrated with the actual documentation of the project.

The following sections will elaborate on these requirements and how they have been integrated into the developed tool.

It is also important to refer that, the prototype was developed as a web API written in Python. On top of this API, we build a web application client that provides simple analysis features to generate an infrastructure diagram and editing capabilities in the form of a custom PlantUML editor.

We expect the workflow presented in the previous section and combination of tools to be easy to understand and appealing enough to be used and considered to improve the documentation. Moreover, we expect new clients built on this API to unlock new features and usages, as we suggest in the chapter on future work and improvements.

5.3 Architecture

Our prototype's conceptual architecture is depicted in Figure 5.1. The main components are the *Application*, where all the logic is implemented, the *Kafka* and *Zookeeper* services to orchestrate requests and allow scalability, the *PlantUML server* to render the images, the *GitHub API* used to gather project information and the *Web App* to facilitate the interaction. It is important to note that this architecture does not need any storage; everything happens in memory and is then saved on the project folder, making everything needed to generate and recover the diagrams always available on the source code.

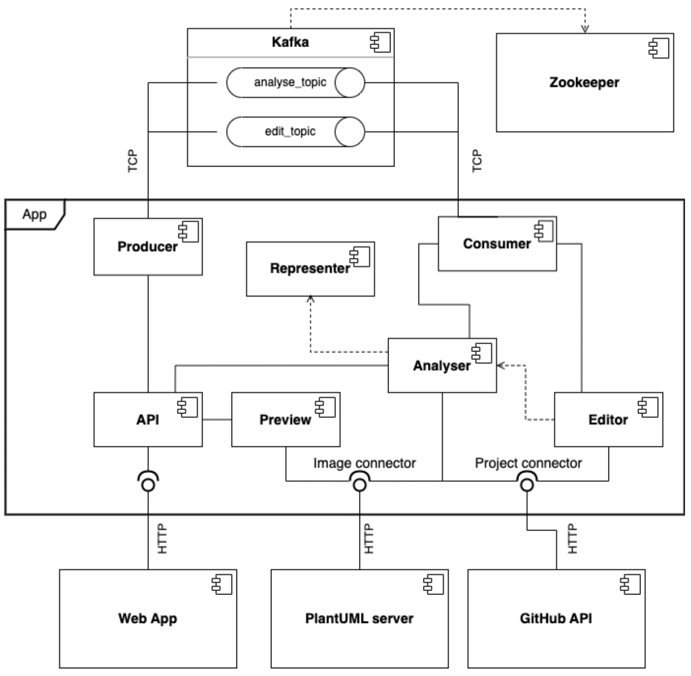


Figure 5.1: Prototype's conceptual architecture.

Starting by describing the components outside the application. *Kafka* is used to implement a queue of requests to analyse projects and edit diagrams. Using *Kafka* allows a high degree of concurrency since the application can deal with these requests asynchronously when it has free resources. *Kafka* depends on *Zookeeper* to get its configuration data. In this prototype, we are not taking full advantage of multiple brokers and partitions, which *Zookeeper* would also manage. The system only uses one broker. Nevertheless, this architecture is ready to scale if needed. The *GitHub API* helps to retrieve project files and their content to generate the diagrams. After analysing a project, *GitHub API* is again used to create requests on the projects with the diagram file. In the case of editions, the requests will be enriched with extra files to track the changes. The content of these files will be analysed later in this chapter. The component running a *PlantUML server* is used to generate graphical representations of the models. This component can receive requests with PlantUML text and responds with an image. We opted to host the server to avoid delays in the image generation that could occur if we used the public API of PlantUML. This decision improves the user experience, especially during the edition of the diagrams. To interact with the application, we have built a *Web App* client that uses the API provided by the application. This *Web App* will be reviewed more in-depth in Section 5.6. On a high level, the *Web App* allows creating diagrams, opening an editor, changing the diagram and saving the changes. It is also possible to use this client to preview and test changes directly on the editor screen without saving them.

Now for the components inside the application, the ones responsible for all the logic. The *Producer* and *Consumer* components serve as an interface between the application and *Kafka*. The *Producer* sends requests to the correct topic, and the *Consumer* retrieves those messages and routes them to the component that can understand them. The *Producer* immediately sends the messages when the API block receives a request, while the Consumer only gets new messages when resources are available to process them. This mechanism ensures that no message is lost.

Messages in the *analyse_topic* will be routed to the *Analyser* component. This component is responsible for gathering the source code's infrastructure files and reading their content, accomplished by using the *Project connector* that abstracts the calls to the *GitHub API*. The abstraction created by the *Project connector* allows new services to be added to expand the application compatibility. With the information retrieved, the *Analyser* creates a series of objects that form an internal diagram model. This model can then provide different representations formats of the infrastructure, like JSON used to answer API requests or plain text, PlantUML format in this case, to be used in the interaction with the *PlantUML server* to get an image visualisation. The *Representer* component manages these representations from which the *Analyser* depends. The *Analyser* also uses the *Image connector* to interact with the *PlantUML server* to get the images. The *Image connector* abstracts the interactions, allowing it to connect with other image engines.

The *Editor* component consumes messages from the *edit_topic*. The messages on this topic have information about the model elements that have been changed, created and deleted. Optionally, the edit message can have information on the latest state of the diagram. If that is the case, the *Editor* component, which depends on the *Analyser*, requests an image representation and then, using the *Project connector*, creates a new pull request on GitHub with the latest image and information on what was edited. In the cases where the diagram state is not sent on the edit message, the *Analyser* must construct the internal model and apply the changes to get the image of the infrastructure model. The content of the files added to the pull requests after an edit is described in the next section about design decisions.

To improve the interaction with the application, a preview functionality is provided to view the state of the infrastructure diagram in real-time. This feature is handy while editing the diagram to check the changes without submitting them. Previewing is accomplished with the *Preview* component. This component accepts a request with a textual representation of the infrastructure diagram and uses the *Image connector* to retrieve an image representation of the model.

Finally, the *API* component provides an interface to make requests and get applications' responses. This component can send messages to the *Producer* to start analysis and create edit requests. It interacts directly with the *Preview* component to get previews of the model and with the *Analyser* to receive different representations of the infrastructure, which can then be used to build clients to facilitate the interaction. The API calls will be described in Section 5.5.

5.4 Design decisions

The prototype uses the domain model represented in Figure 5.2 to describe the recovered infrastructure. This model uses an interface that creates a common contract for all the elements of the final diagram. The diagram components are then added to a *Graph* object responsible for managing the relations between each component and invoking their representation method to generate the final representation. With this model, we applied a set of known design patterns and strategies to develop the needed features. Those patterns and strategies will be reviewed in the following subsections.

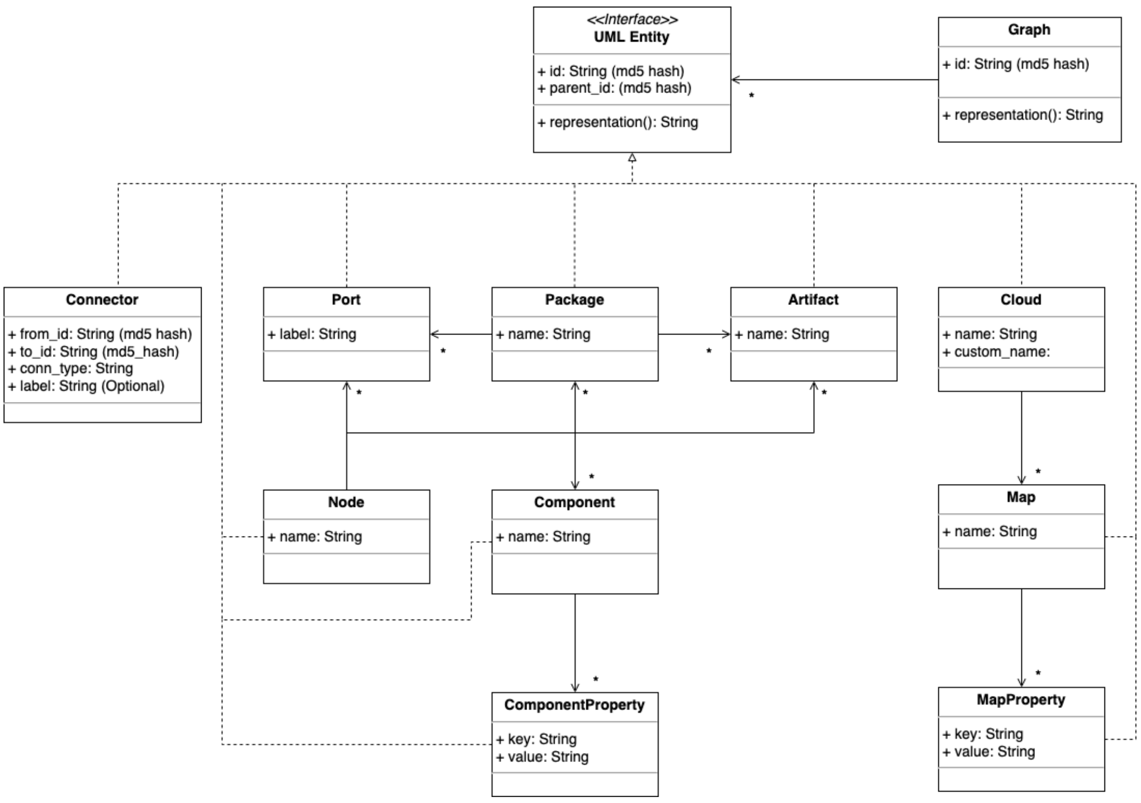


Figure 5.2: Class diagram of the Domain Model.

5.4.1 Creating different UML elements

Our application needs to support different types of UML elements while including the ability to add more elements if needed in the future. This needed behaviour was achieved with the design pattern **Factory Method**.

With our concrete example, and looking into the class diagram from Figure 5.2, all the elements, the products of our factory, implement a common interface, *UMLEntity*. The interface defines a unique identifier, the *id*, and a *parent_id* to identify if an element exists inside another. For example, all the UML elements will exist inside a Graph object, and a Node object can have

Ports and Packages inside. So in this simple example, the Node would have the Graph as a parent, and for the Ports and Packages, the Node will be the parent. The *UMLEntity* interface also defines the method *representation()* responsible for generating a representation of objects understandable by other application parts.

Besides the common interface, this strategy also relies on classes to help create each element, as shown in Figure 5.3. For the sake of simplicity, not all the creator classes are represented, but every *UMLEntity* class has its creator class.

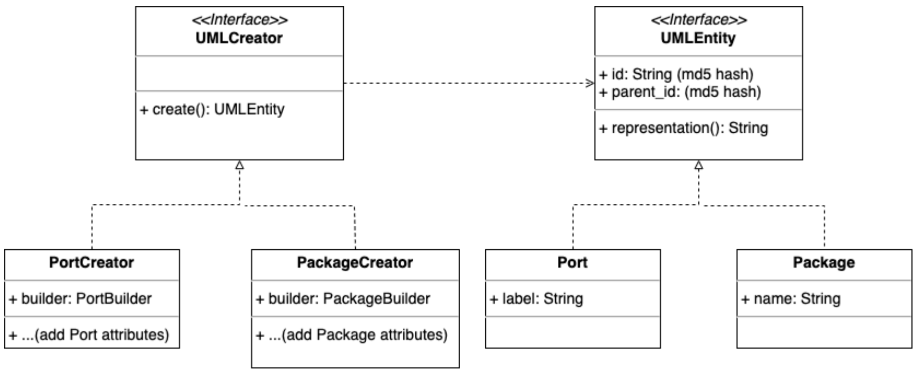


Figure 5.3: Class diagram for the Factory and Builder design patterns implemented.

5.4.2 Building UML elements step by step

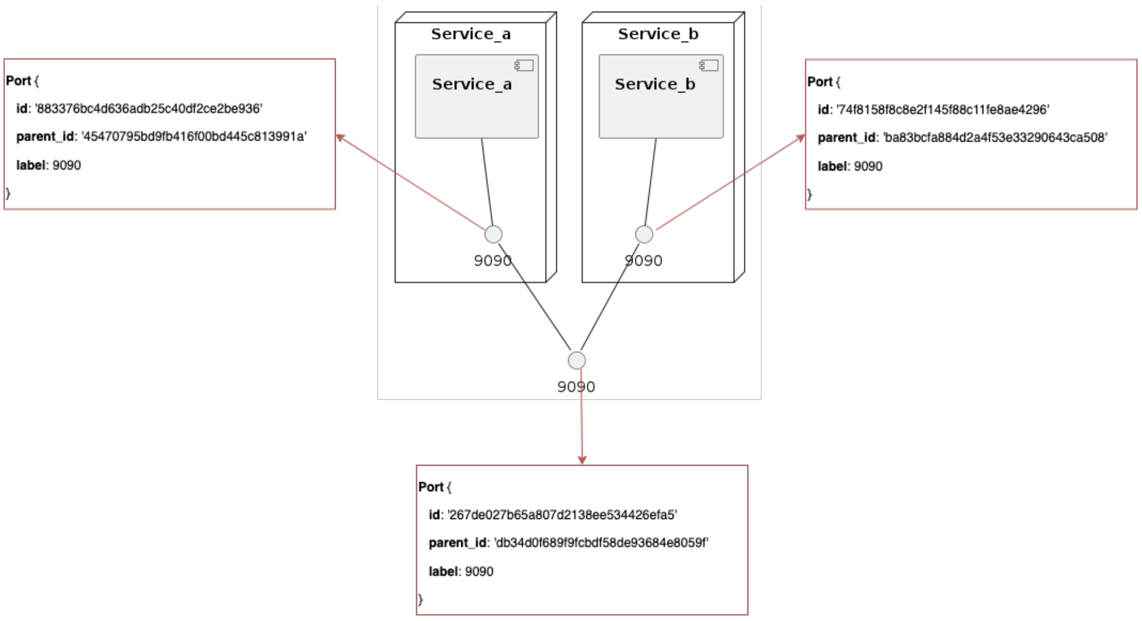
Since the information on each element of the diagram we are trying to build is scattered through the infrastructure files, the UML elements need to be built iteratively. Moreover, the same element may have different attributes. This problem can be solved using a **Builder** design pattern, where it is possible to build the elements step by step. Using this strategy is possible to create different elements using the same construction process.

The builder design pattern follows a structure where an interface defines the build methods, and then concrete builders are used to creating the final objects. In our implementation, since we already have factories for each UML element, we decided to apply a simpler version of this structure. Each factory has the necessary methods to add attributes to the objects, dropping the need for an interface. So, in our model, and taking the case of the *PortCreator* from Figure 5.3, the creator instance starts by instantiating an empty *PortBuilder* object and then this object is enriched with the information found in the infrastructure artefacts. In the end, the create method is called to return the final *Port* object.

5.4.3 Accepting manual changes and persisting them

A version control system was used to provide the ability to edit the diagram and save those changes. Based on the mechanisms used by Git [1], we have created a lightweight version control system powered by the unique ids of each element, as mentioned previously.

During the first analysis of the project, each element of the infrastructure is translated to an *UMLEntity* object with a unique ID, calculated using the md5 hashing algorithm. The hash is calculated using information like the name of the service, its properties and information being inferred during the analysis, like the parent element id. This combination is needed to avoid duplicated ids since we need a deterministic hashing mechanism. That is, the output is always the same for the same input. To better understand this, let us follow the example diagram generated by the application in Figure 5.4.



```
Port {
  id: '883376bc4d636adb25c40df2ce2be936'
  parent_id: '45470795bd9fb416f00bd445c813991a'
  label: 9090
}
```

```
Port {
  id: '74f8158f8c8e2f145f88c11fe8ae4296'
  parent_id: 'ba83bcfa884d2a4f53e33290643ca508'
  label: 9090
}
```

```
Port {
  id: '267de027b65a807d2138ee534426efa5'
  parent_id: 'db34d0f689f9fcbdf58de93684e8059f'
  label: 9090
}
```

Figure 5.4: Infrastructure diagram with generated ids.

In this example, all the services use port "9090" to map to an external port, also on the address "9090". Using a simple hashing by the address of the ports would result in three identical ports. Including context information in the hashing function, the application can generate unique ids for each port, as shown in the figure. The context information includes information like the service that the port belongs to and IN and OUT connections. With this information, and while the port exists in the infrastructure definition, it will always be associated with the same id and parent id. It is essential to uniquely identify the port even if we run the application multiple times. Using this approach, the objects of the diagram are always identifiable even if the version of the diagram changes by manual changes or changes in the infrastructure definition. This is the basis to keep manual changes and evolve the diagram as described in the following subsection.

[1] Official GIT website: https://git-scm.com/

5.4.4 Updating the diagram and keeping manual changes

Following the example from Figure 5.4, let us suppose we want to manually change the port on *service_a* to the address "9091". Being confident that this port will always have the same ID and combining this with **Event sourcing** [16] made it possible to create a memoryless system that can keep track of the changes and combines them with updates on the infrastructure. We say that the application is memoryless because nothing is saved on the application side. The repositories under analysis are used to capture all the changes. Event sourcing plays a critical role in keeping the manual changes between analyses. This is the mechanism used to understand what was changed by the user and to understand if the changes still make sense compared to the most recent developments of the project. By using event sourcing, the tool can apply a semantic diff to the changes, which means understanding the purpose of the change. Since every manual change originates a new state of the diagram and given that with event sourcing, the tool applies all the states every time an analysis occurs, it is possible to understand, for example, if an added port is still needed in the case that the infrastructure documents have been changed and the services associated with the port have been removed, or even if the port was added to the infrastructure docs and so, the manual change can be skipped.

Back to our example, the application starts by looking into the infrastructure artefacts and generates the first state of the diagram, the one in Figure 5.4. Nothing is saved on the project repository if the user does not apply any change, and only the diagram is provided. The repository analysis can always retrieve this first state, and all the objects will be identified in the same way. When the user edits the first version, this action is seen as an event that changes the diagram state. That event is saved on the repository by the addition of two files. One is responsible for keeping track of the order of the events. The other defines what happened in the event. If a new change is added, one new file with the changes is saved, and the order file is updated with a new reference. The file responsible for keeping track of events order is saved under the name generated by the hashing over the project name and has references for the files with the event changes that are saved with a hashing formed by the repository name and the date and time that the event happened. With this naming convention, the application only needs the project's name to find the edition files without needing pointers or references. In our example, the event file will have the structure in Table 5.1.

In this structure, the model can verify what was added, deleted or changed, and in the snapshots block, theres the last state of the relevant objects. Each edit event originates a file similar to this one allowing the application to capture all the changes as a sequence of events that must occur to reach the final state. In this structure, the *parent_id* is pivotal since it creates a hierarchy. Continuing with our example and assuming that *service_a* is removed from the infrastructure artefacts, the first analysis will generate a diagram only with *service_b*. When looking into the edition files, the application understands that this change is unnecessary and skips it.

Figure 5.5 shows a simplified flow chart of the operations described.

Table 5.1: Object saved in the event files after edit.

Changes object

```
{
  "add":[],
  "delete":[],
  "changes":[
      "883376bc4d636adb25c40df2ce2be936",
  ],
  "snapshots":{
      "883376bc4d636adb25c40df2ce2be936":{
          "id":"883376bc4d636adb25c40df2ce2be936",
          "parent_id":"45470795bd9fb416f00bd445c813991a",
          "label":"9091",
      }
  }
}
```

5.4.5 Providing different representations of the diagram

The ability to provide different representation formats was also a concern and a necessity while developing the application. The diagram should not be restricted to only one format to allow the evolution of the tool and, thus, the usage of different engines to generate the images. In addition to this, and to support the API, the diagram also needed to be represented in JSON format. A **decorator** design pattern was used to achieve this.

The decorator is combined with the Graph object, which wraps the diagram objects acting as an orchestrator for the operations on these objects. The class diagram for this pattern can be seen in Figure 5.6.

This pattern defines an interface for all the representation classes that will aggregate the *Graph* class so they can use its default representation and then apply transformations to achieve the desired representation.

5.5 API definition

The developed tool was built with evolution in mind, and we thought that having an API where different clients can be used is the best way to adapt the tool to more projects, i.e., developers can build clients on top of this API to use its recovery and editing capabilities and add extra functionalities inline with the project requirements. Although we do not consider this API to be one of our main contributions, it opens a door that we surely intent to explore in subsequent iterations of this work. Notwithstanding, this API is used by the client that was built and described in the next section, so it is essential to describe it briefly.

The API defines the endpoints in Table 5.2.

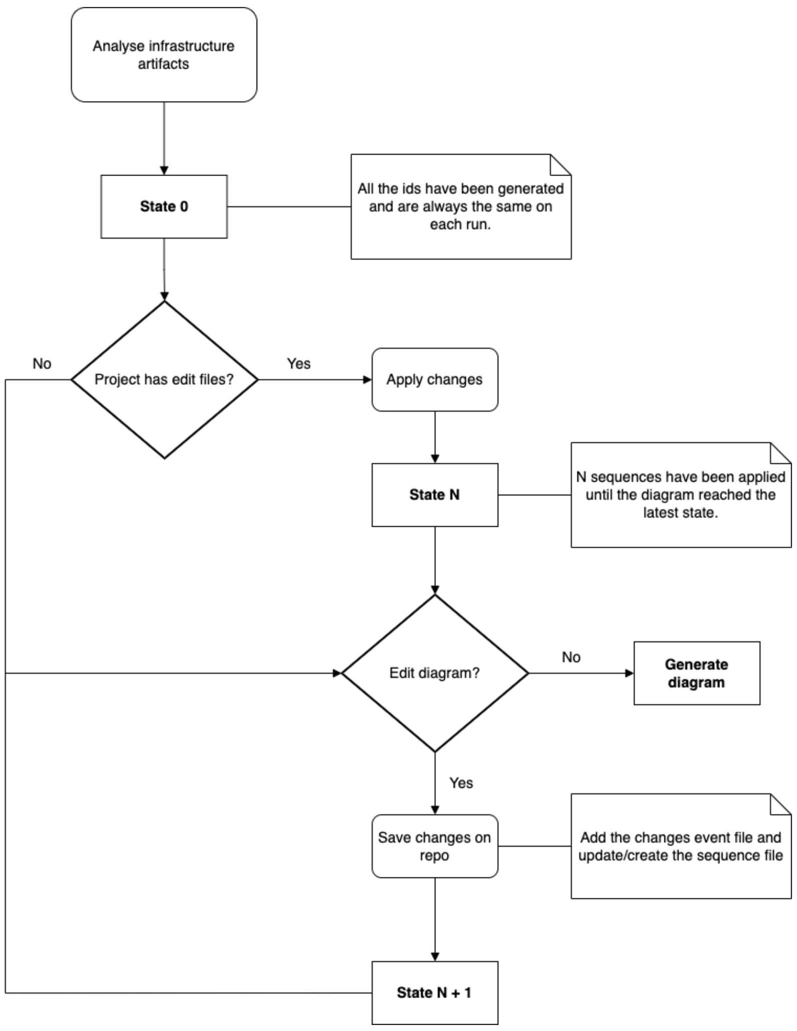

Analyse infrastructure artifacts
State 0
All the ids have been generated and are always the same on each run.
No
Project has edit files?
Yes
Apply changes
State N
N sequences have been applied until the diagram reached the latest state.
Edit diagram?
No
Generate diagram
Yes
Save changes on repo
Add the changes event file and update/create the sequence file
State N + 1

Figure 5.5: Infragenie simplified flow chart.

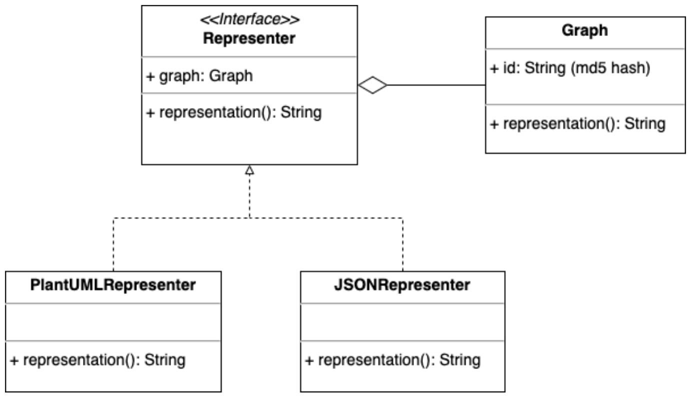

<<Interface>>
Representer
+ graph: Graph
+ representation(): String
Graph
+ id: String (md5 hash)
+ representation(): String
PlantUMLRepresenter
+ representation(): String
JSONRepresenter
+ representation(): String

Figure 5.6: Class diagram for the Decorator design pattern implemented.

The first endpoint starts the analysis of a project. The response of this endpoint is just a message stating that the analysis will start soon in the project. Since the analysis does not occur

Table 5.2: API endpoints.

Endpoints		
1	POST	`\analyse`
2	POST	`\edit`
3	POST	`\previewgraph`
4	GET	`\graph`

in real-time, it is scheduled with a Kafka message, as explained previously. The analysis will add a diagram of the infrastructure at the project's root. This endpoint expects the object in Table 5.3.

Table 5.3: Object used in the \analyse enpoint.

Analyse object

```
{
  "repository": "project location",
  "branch": "branch definition" (Optional),
  "add_readme": "TRUE or FALSE"  (Optional)
}
```

The object receives the repository location. A branch can be provided if the project works with Git, so the analysis occurs on that branch. The *add_readme* parameter is used to include the diagram in the readme file. If a readme file does not exist on the project's root and this parameter is TRUE, a new readme file will be created with the needed diagram information. Regarding the branch, if it is not provided, the analysis occurs on the project files that are returned by the location provided, usually the main or master branches.

The *\edit* endpoint also interacts directly with the project code by updating the diagram with the changes that resulted from the edit operation and adding new files to recover the edit operation in the future, as explained in the previous section. This endpoint expects the object in Table 5.4.

The first three parameters are the same used in the *\analyse* endpoint. In addition, this endpoint receives an object that describes what was changed, deleted and added during the edit event. It also receives a representation of the diagram after the edition. The representation of the diagram is used solely to make this operation faster. With this information, the analysis operation is unnecessary, and the application can invoke the image generation for the diagram during the edit operation. Given that the diagram information should be available at the end of an edit operation, this optional parameter can be provided without resourcing to other sources. If this parameter is not provided, the application needs to run an analysis on the project and apply the changes to update the diagram.

Table 5.4: Object used in the \edit enpoint.

Edit object
`{`
`"repository":"project location",`
`"branch": "branch definition" (Optional),`
`"add_readme": "TRUE or FALSE"  (Optional),`
`"changes": "similar to the object in Table 5.1",`
`"graph_text": "representation of the diagram" (Optional)`
`}`

Using the *preview* endpoint is possible to obtain an image of the diagram in base64 format. The endpoint expects an object like the one in Table 5.5.

Table 5.5: Object used in the \preview enpoint.

Preview object
`{`
`"graph_text": "representation of the diagram"`
`}`

The *graph_text* parameter expects a PlantUML representation since it is the only image engine used, but in future versions, this endpoint should be more general and maybe work together with another endpoint that provides representations in different formats.

The last endpoint, *graph*, can be used to get a JSON representation of the diagram. This enpoint expects information about the project location and the branch to be analysed in a call similar to the following:

```
...\graph?repo=repo-location&branch=branch-name
```

When this endpoint is used, an analysis is started immediatly and the response is a JSON representation of the *Graph* object representated in the domain model from Figure 5.2. A sample of this JSON object can be seen on Table 5.6.

When this endpoint is used, an analysis is started immediately, and the response is a JSON representation of the Graph object represented in the domain model from Figure 5.2. A sample of the response can be seen in Table 5.6.

The *graph* endpoint can be used to get a complete representation of the diagram, with the ids generated as well as the relations between each object.

Table 5.6: Graph object used in the response of \graph endpoint.

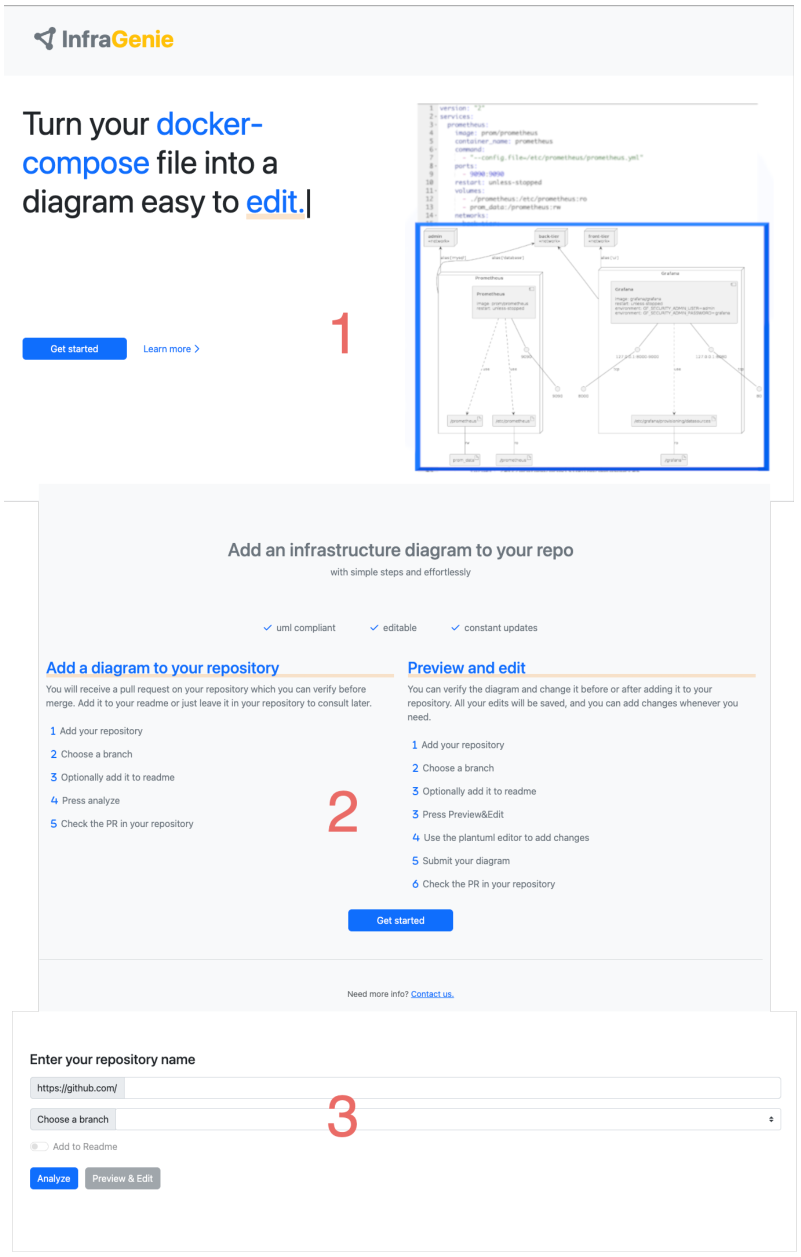
Graph object in JSON format

```
{
  "object": "Graph"
  "id": "73047605a30e73c1116eafd242d3262b",
  "nodes": [
      {
          "object": "Node",
          "id": "ba9f11ecc3497d9993b933fdc2bd61e5",
          "parent_id: "73047605a30e73c1116eafd242d3262b",
          "name": "Git",
          "components": [
              {
                  "object": "Component",
                  "id": "2ccd832ffb54029d6c5715cbd9928c6e",
                  "parent_id": "ba9f11ecc3497d9993b933fdc2bd61e5",
                  "name": "git",
                  "properties": [...]
              }
          ],
      },
      {...},
  ],
  "connectors": [
      {
          "object": "Connector",
          "id": "7b548ffbb150d9301ecc0a699cabda5b",
          "parent_id": "73047605a30e73c1116eafd242d3262b",
          "from_id": "ae7a460a21e18a1bb1312f616b1487ac",
          "to_id": "01711216126be07766ef27ce7e0393dc",
          "connector_type": "..d..>",
          "label": "use",
      },
      {...},
  ],
  ...,
}
```

5.6 User interface and experience (UI/UX)

As mentioned throughout this chapter, we developed a web application to provide a more friendly user experience with the tool. In this section, we will review the screens and functionalities of this web application. It is important to note that this web application was also used to showcase the tool and its functionalities. So, the web application is more than just a client. It is also an onboarding instrument.

The web application is available at the address https:\www.infragenie.eu. Some screenshots of the homepage can be seen in Figure 5.7.


InfraGenie

Turn your docker-compose file into a diagram easy to edit.|

```
version: "2"
services:
  prometheus:
    image: prom/prometheus
    container_name: prometheus
    command:
      - "--config.file=/etc/prometheus/prometheus.yml"
    ports:
      - 9090:9090
    restart: unless-stopped
    volumes:
      - ./prometheus:/etc/prometheus:ro
      - prom_data:/prometheus:rw
    networks:
```

Prometheus
image: prom/prometheus
restart: unless-stopped

Grafana
image: grafana/grafana
restart: unless-stopped
environment: GF_SECURITY_ADMIN_USER=admin
environment: GF_SECURITY_ADMIN_PASSWORD=grafana

Add an infrastructure diagram to your repo

with simple steps and effortlessly

✓ uml compliant ✓ editable ✓ constant updates

Add a diagram to your repository

You will receive a pull request on your repository which you can verify before merge. Add it to your readme or just leave it in your repository to consult later.

1 Add your repository
2 Choose a branch
3 Optionally add it to readme
4 Press analyze
5 Check the PR in your repository

Preview and edit

You can verify the diagram and change it before or after adding it to your repository. All your edits will be saved, and you can add changes whenever you need.

1 Add your repository
2 Choose a branch
3 Optionally add it to readme
3 Press Preview&Edit
4 Use the plantuml editor to add changes
5 Submit your diagram
6 Check the PR in your repository

Need more info? Contact us.

Enter your repository name

https://github.com/

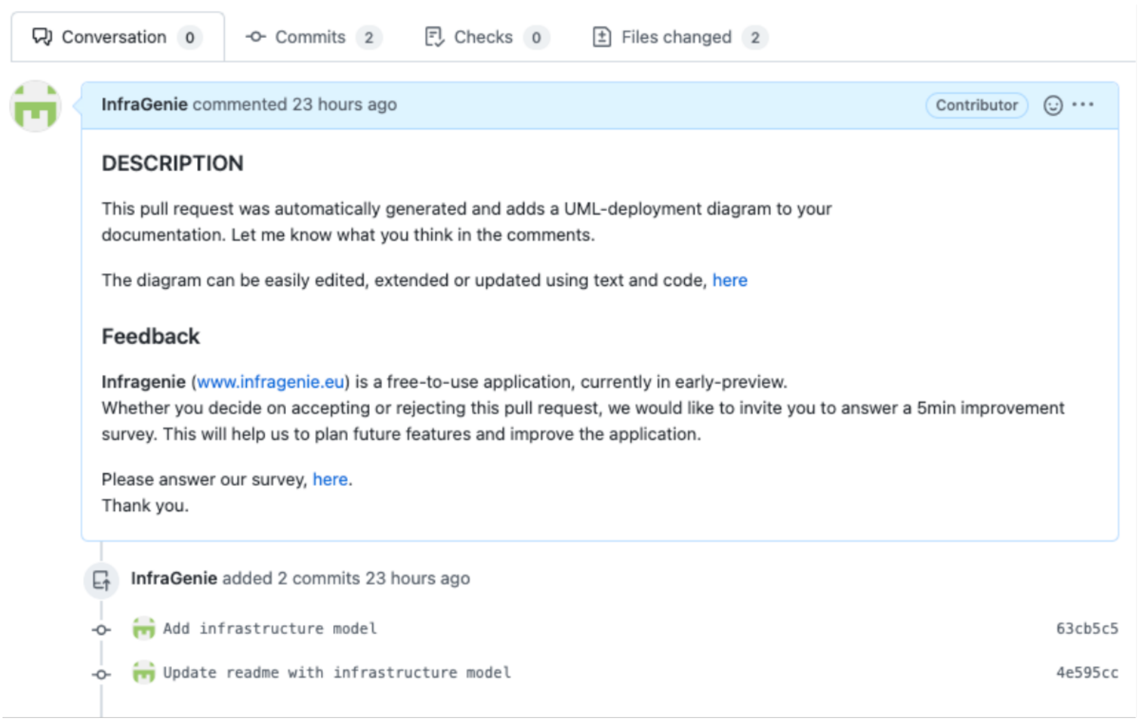
Choose a branch

Add to Readme

Figure 5.7: Infragenie: Homepage.

The homepage is comprised of three main areas, marked with numbers 1 to 3 in the Figure. The first one is a banner promoting the product and a call to action to analyse projects. The second area shows a small guide on generating diagrams and editing them if needed. Finally, the third area, the most relevant, is the one that allows the tool's usage. Here, the user needs to add the name of the GitHub repository to be analysed. The application will pre-populate a dropdown menu with all the branches available on the project. After choosing the branch, the user can select if he wants to add the diagram to the readme file or not. This information is needed by both the `\analyse` and `\edit` endpoints, which are used here. With all the information, the user can follow two different paths on the application by pressing either of the buttons, *Analyse* or *Preview & edit*.

Pressing *Analyse* creates a request to analyse the project, which will end with creating a pull request in the user repository. The pull request adds a diagram image and changes the Readme file if that was selected. Besides that, the pull request is accompanied by a description of the project, the project's URL, a direct URL to start the editor and a request to fill in the survey used in the study presented in the next chapter. An example of one pull request can be seen in Figure 5.8, where it is possible to check that the application adds two commits to the project, one to include the diagram and another to update the readme file.



InfraGenie commented 23 hours ago

DESCRIPTION

This pull request was automatically generated and adds a UML-deployment diagram to your documentation. Let me know what you think in the comments.

The diagram can be easily edited, extended or updated using text and code, here

Feedback

Infragenie (www.infragenie.eu) is a free-to-use application, currently in early-preview.
Whether you decide on accepting or rejecting this pull request, we would like to invite you to answer a 5min improvement survey. This will help us to plan future features and improve the application.

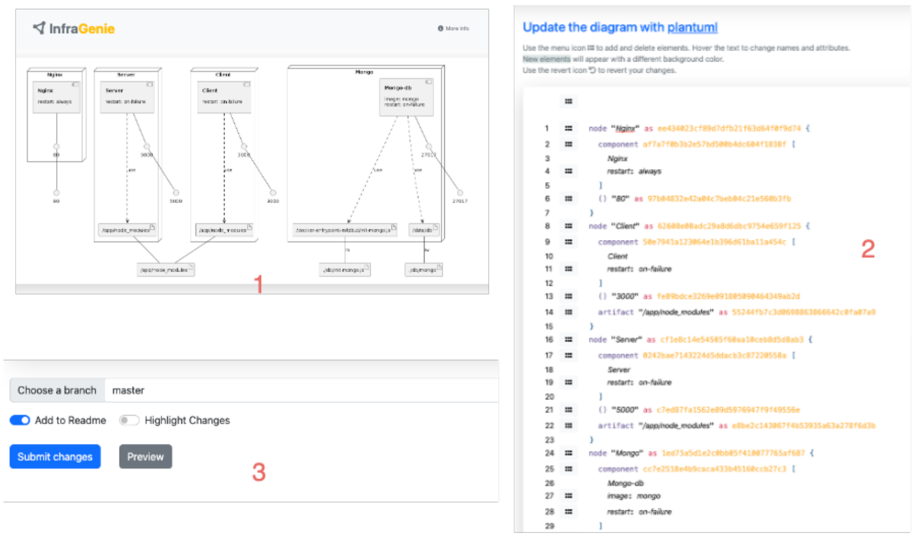
Please answer our survey, here.
Thank you.

InfraGenie added 2 commits 23 hours ago

```
Add infrastructure model                     63cb5c5
Update readme with infrastructure model      4e595cc
```

Figure 5.8: Infragenie: Pull request.

Using the button *Preview & edit*, the editor page opens, and the user can check the diagram and edit it before sending it to the project. The editor page has three areas, the previewer with a preview of the current state of the diagram, the editor where it is possible to interact with the PlantUML code that generates the diagram and the third one with a form similar to the one on the home page to save what was edited and update the preview image. This page uses the `\edit`, `\preview` and `\graph` endpoints, and the three main areas can be seen in Figure 5.9.



Update the diagram with plantuml

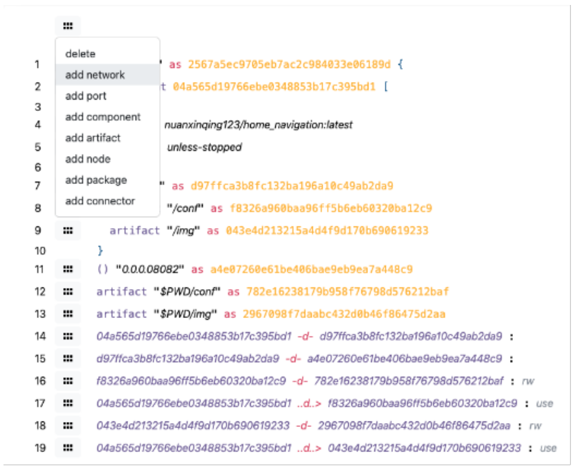
Use the menu icon to add and delete elements. Hover the text to change names and attributes.
New elements will appear with a different background color.
Use the revert icon to revert your changes.

```
node "Nginx" as ee434023cf89d7dfb21f63d64f0f9d74 {
  component af7a7f0b3b2e57bd500b4dc604f1838f [
    Nginx
    restart: always
  ]
  () "80" as 97b04832e42a04c7beb04c21e560b3fb
}
node "Client" as 62608e08adc29a8d6dbc9754e659f125 {
  component 50e7941a123064e1b396d61ba11a454c [
    Client
    restart: on-failure
  ]
  () "3000" as fe89bdce3269e0918050904643\49ab2d
  artifact "/app/node_modules" as 55244fb7c3d0698863866642c0fa07a9
}
node "Server" as cf1e8c14e54505f60aa10ceb8d5d8ab3 {
  component 0242bae7143224d5ddacb3c87220558a [
    Server
    restart: on-failure
  ]
  () "5000" as c7ed87fa1562e89d5976947f9f49556e
  artifact "/app/node_modules" as e8be2c143067f4b53935a63a278f6d3b
}
node "Mongo" as 1ed75a5d1e2c0bb05f410077765af687 {
  component cc7e2518e4b9caca433b45160ccb27c3 [
    Mongo-db
    image: mongo
    restart: on-failure
  ]
```

Figure 5.9: Infragenie: Edit page.

Although based on PlantUML, the editor does not allow inserting PlantUML text freely. Instead, each line has a context menu where the allowed changes for that zone of the diagram can be selected. This decision was made to guide the users to insert valid elements and enforce them to maintain the specifications of the component and deployment UML diagrams. Also, at this stage not all the specification is supported, has can be seen in the domain model from Figure 5.2.

One of the context menus can be seen in Figure 5.10, showing that it is possible to delete or add new elements to the diagram. Changing names, labels, or descriptions is as simple as double-clicking and changing the text.



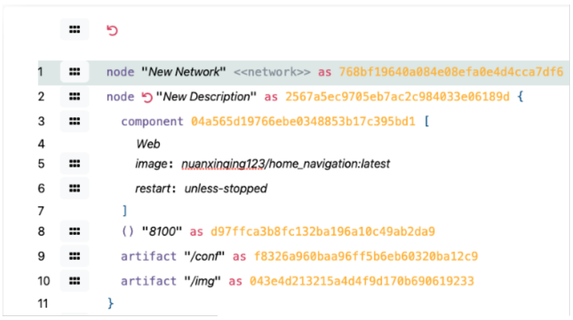
```
' as 2567a5ec9705eb7ac2c984033e06189d {
t 04a565d19766ebe0348853b17c395bd1 [

  nuanxinqing123/home_navigation:latest
  unless-stopped

" as d97ffca3b8fc132ba196a10c49ab2da9
  "/conf" as f8326a960baa96ff5b6eb60320ba12c9
    artifact "/img" as 043e4d213215a4d4f9d170b690619233
  }
  () "0.0.0.08082" as a4e07260e61be406bae9eb9ea7a448c9
  artifact "$PWD/conf" as 782e16238179b958f76798d576212baf
  artifact "$PWD/img" as 2967098f7daabc432d0b46f86475d2aa
  04a565d19766ebe0348853b17c395bd1 -d- d97ffca3b8fc132ba196a10c49ab2da9 :
  d97ffca3b8fc132ba196a10c49ab2da9 -d- a4e07260e61be406bae9eb9ea7a448c9 :
  f8326a960baa96ff5b6eb60320ba12c9 -d- 782e16238179b958f76798d576212baf : rw
  04a565d19766ebe0348853b17c395bd1 ..d..> f8326a960baa96ff5b6eb60320ba12c9 : use
  043e4d213215a4d4f9d170b690619233 -d- 2967098f7daabc432d0b46f86475d2aa : rw
  04a565d19766ebe0348853b17c395bd1 ..d..> 043e4d213215a4d4f9d170b690619233 : use
```

Figure 5.10: Infragenie: Context menus to edit the diagram.

Every change on the diagram will reveal a revert icon to facilitate the rollback of the changes. Also, to make it visually more understandable, all the new additions to the editor appear with a different background. These aspects can be seen in Figure 5.11.



```
node "New Network" <<network>> as 768bf19640a084e08efa0e4d4cca7df6
node "New Description" as 2567a5ec9705eb7ac2c984033e06189d {
  component 04a565d19766ebe0348853b17c395bd1 [
    Web
    image: nuanxinqing123/home_navigation:latest
    restart: unless-stopped
  ]
  () "8100" as d97ffca3b8fc132ba196a10c49ab2da9
  artifact "/conf" as f8326a960baa96ff5b6eb60320ba12c9
  artifact "/img" as 043e4d213215a4d4f9d170b690619233
}
```

Figure 5.11: Infragenie: Editor with changes.

To facilitate the edition and help the user, the form on the third zone of the edit page has a *Preview* button that updates the diagram on the first zone. These changes can be highlighted by selecting the *Highlight Changes* radio button. An updated diagram with highlighted changes can be seen in Figure 5.12. This example reflects the changes from Figure 5.11, where a new network was added and the service name changed.

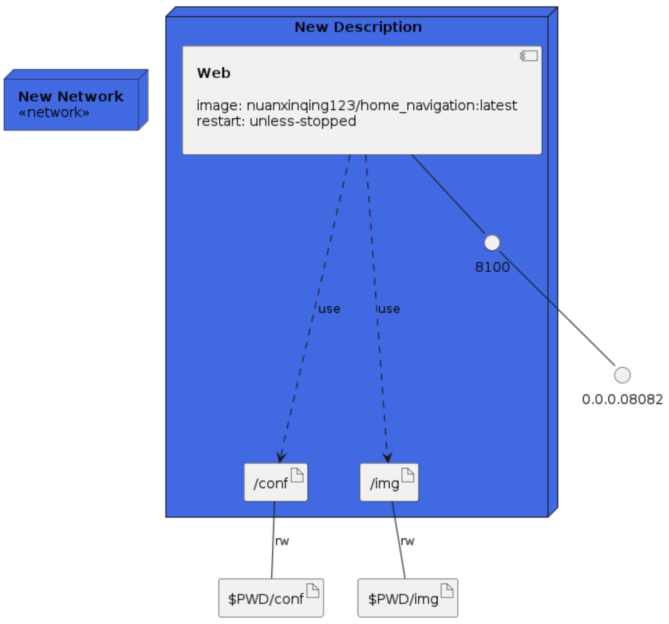


Figure 5.12: Infragenie: Diagram with changes highlighted.

In combination with the preview feature, the editor allows not only corrections to the diagram but also experimenting with changes to the infrastructure and plan and understanding of the impact of future changes on the architecture.

When the user is done with the changes, the form on the third zone of the page can be used to submit the changes, which will generate a new pull request on the project with the edit event details. The result of the flow presented in the previous sections.

5.7 Limitations

Some concessions have been made to constrain the application's scope and allow the development of a minimum viable product in the available time frame.

The first one was to limit the infrastructure analysis to docker-compose. Docker is one of the most popular containerization technologies, with 83% of the market [23]. As such, many applications use docker-compose to orchestrate multiple containers. Our study will focus on open source projects, and a simple search on Github returns more than 20k projects using docker-compose. This simplification will keep the scope small and not compromise the evaluation study.

Another simplification was the restriction for projects hosted on Github. Compared to other tools, there is a broader knowledge of the GitHub API, leading to a faster development process. The API provided by GitHub also has the advantage of allowing searching and reading content of files without downloading them, reducing the storage needs during the deployment of the application. This has the advantage of reducing not only deployment costs but also complexity.

The graphical engine behind the application uses PlantUML[2]. PlantUML is easy to integrate and already complies with almost all the UML standards, which does not happen with other tools, especially the ones that are more graphical-oriented. Although, it has the disadvantage of requiring a learning curve since it is a textual modelling language. On the other side, text-based is the prefered way to generate models among the software development community [38], which is our target for the evaluation.

5.8 Deployment

Infragenie was deployed using the Google cloud platform, containerized with Docker containers managed by docker-compose. All the containers have then been deployed to Google cloud virtual machines. Since the application does not keep states and memory, there is no need for databases, and the disk requirements are also minimal, reducing the deployment costs. A general scheme of the technologies being used and how each container interacts is depicted in Figure 5.13.

Infragenie backend and the web client have been built using the Django framework, deployed on its own container. This container also hosts the Gunicorn server, a lightweight HTTP server fully compatible with the Django framework. To increase the fault tolerance and security, thus making the deployment more "production ready", we decided to combine Nginx with Gunicorn.

Nginx works as a proxy between external requests and the application and as a load balancer for better traffic handling. Nginx is also used to manage and apply the certificates generated by

[2]Official PlantUML webpage and documentation: https://plantuml.com/

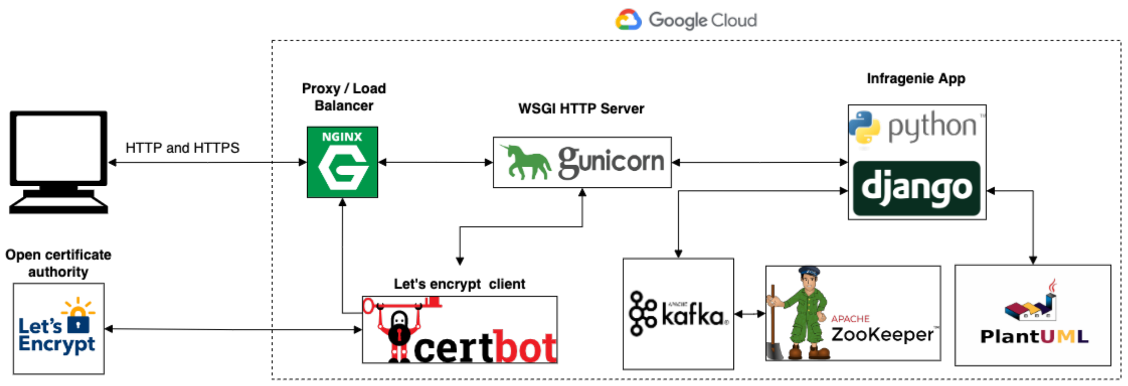


Figure 5.13: Technologies used in the deployment of the application.

Certbot, which acts as a client for Let's Encrypt. This way, the application is served over a secure and certified connection.

The way Kafka, Zookeeper and PlantUML server are being used have already been described in previous sections.

In Figure 5.14, there's a representation of the infrastructure generated by Infragenie. The image shows all the services described so far and also reveals an extra service, the Cron. This service is used to run a daily cron job to renew the certificates and reload the Nginx configuration. It is also easy to identify how the services interact and which ports are being exposed.

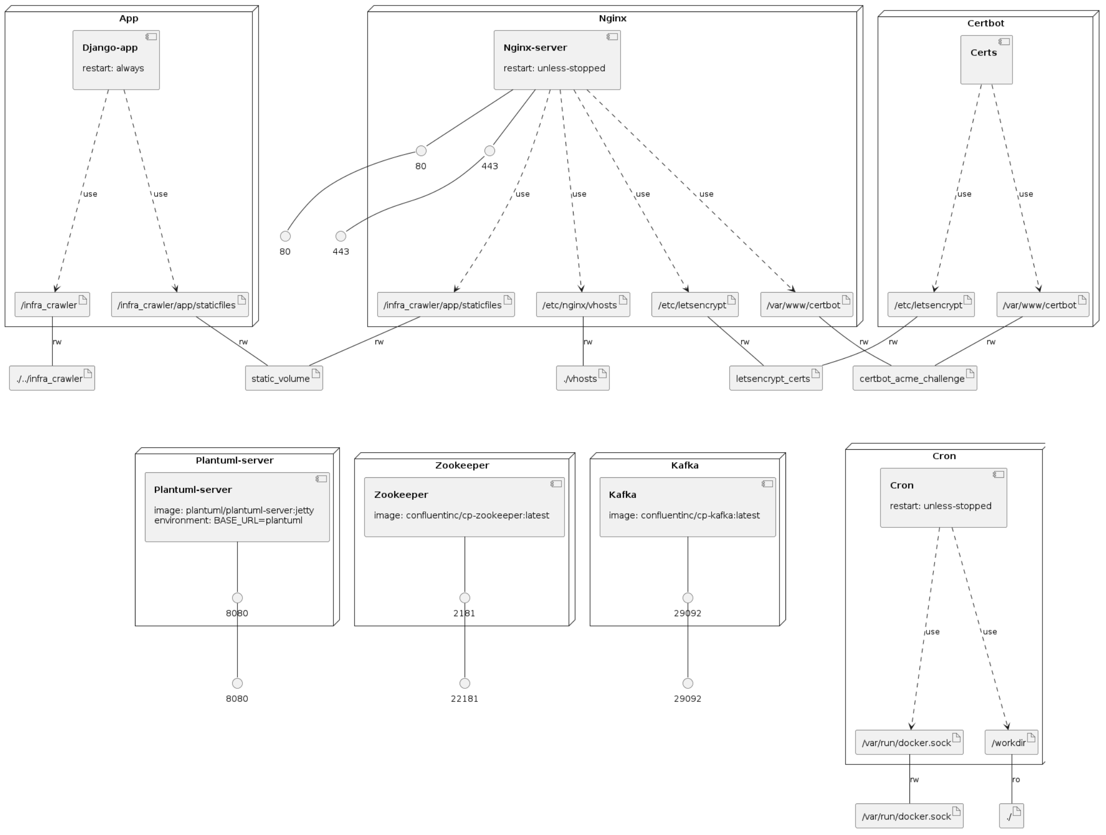


Figure 5.14: Infragenie: Deployed infrastructure.

Chapter 6

Empirical Study

This chapter describes the study carried out to answer the research question formulated in Chapter 4. The study can be framed as **Engineering Research** (i.e. Design Science) [42] that leans on the development of Infragenie (cf. Chapter 5) and uses a Questionnaire Survey [42] to evaluate our prototype. Given our research questions (cf. Section 4), the survey was built around the **Technology Acceptance Model (TAM)** [22]. The research questions try to prove evidence that adopting a tool with similar features to the ones of Infragenie (cf. Section 5.2) would lead to better documentation and overall willingness to use it to produce and keep documentation updated. Following the guidelines of TAM, the answers to the survey can provide evidence of responders' intention to use Infragenie, how they perceive its usefulness and how easy it is to use the tool. The survey incorporates questions using the **Likert scale** [57] and open questions to allow open feedback from the respondents. The chapter starts with a review of the study's goals, following a definition of the method used to select the participants and collect data. It continues with a brief description of how the study was designed and which instruments were used. Then, with the analysis of the collected data, with which we provide answers to the research questions and highlight evidence to prove this thesis hypothesis (cf. Section 4.2). To conclude the chapter, there is a discussion about the threats to the study's validity and a discussion summarising the main findings.

All the documents, resources and other artefacts mentioned in this chapter are available in a **replication package**, hosted in a public repository on GitHub[1]. This package can be used to recreate this study. More details about the replication package and its content can be found in Section B.1

6.1 Study Goals

This study aims to evaluate how fitted is the developed tool to solve the problem of automatically recovering project infrastructures, making it available as an up-to-date artefact that evolves with

[1]Replication package repository: https://github.com/ricardojaferreira/infragenie-replication-package

the project and can be used to improve documentation. Furthermore, we seek evidence to prove this thesis hypothesis, as stated in Section 4.2, where we hypothesise that having infrastructure models that evolve with the source code will promote the creation, updating and usage of the documentation and motivate architectural changes. All of these aspects can be leveraged by a tool like Infragenie. To produce insights into the hypothesis, this study answers the research questions from Chapter 4, transcribed here for the reader's commodity:

RQ1 "To what extent does the adoption of Infragenie lead to better and updated documentation?"

We aim to understand if using the tool developed in this work to generate models of the system architecture on each development iteration contributes to updated documentation and how valuable that model is for the overall documentation of the product. Have we reduced information redundancy between code and textual documentation? Have we create a systematic action to update documentation, like any other steps from a pipeline?

RQ2 "To what extent does the adoption of Infragenie improve the knowledge about the system architecture?"

Developers often add changes to the software without a full comprehension of its architecture. Having a model of the architecture can improve the confidence and planning better future changes? Can the model guide developers to make better decisions and develop faster?

RQ3 "What is the value and willingness to use Infragenie?"

Finally, understanding the acceptance of the tool is crucial. Using models like the Technology acceptance criteria [22] we will try to find out if the tool is adding real value to the project and if the developers are considering the tool for other projects. We will also try to get feedback on what works best and what should be improved with a special focus on the feature that distinguish our tool, the ability to preserve user changes between recoveries.

6.2 Study design

We have designed this study as a benchmarking mechanism to evaluate the developed tool and provide insights into the research questions. For that, an online survey questionnaire [2], comprised of mostly close-ended questions to evaluate the tool, was used to conduct an **Engineering research** as defined by the empirical standards [3]:

[2] ACM page for Questionnaire surveys: https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=QuestionnaireSurveys
[3] ACM page for Engineering Reasearch: https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=EngineeringResearch

> This standard applies to manuscripts that propose and evaluate technological artifacts, including algorithms, models, languages, methods, systems, tools, and other computer-based technologies.

We have chosen this approach for our study because, as the definition says, it is a suitable methodology to evaluate tools and computer-based technologies, a category where we can include Infragenie. Moreover, part of our research focus is to understand how easy to use and adopt the tool is, so a benchmarking study with reduced interference in the process of knowing and using the tool was essential.

The methodology followed in conducting this study and the sections that describe the various steps are summarized in Figure 6.1. The following sections cover in more detail some aspects of the study.

To provide insights about the tool and allow users to give more informed answers, we have prepared a series of instruments to onboard and help use the tool. Those instruments are described in Section 6.2.2.

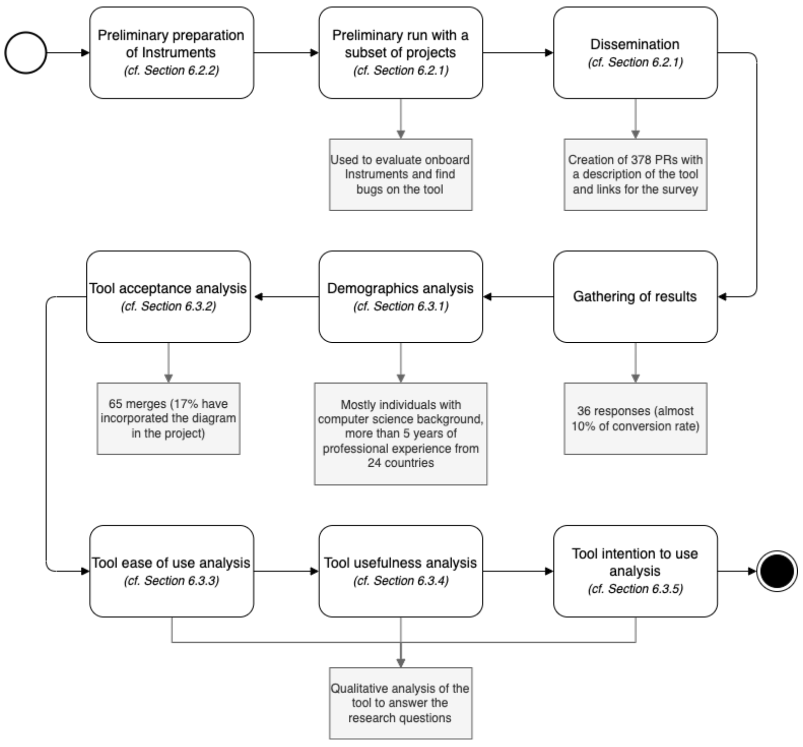

Preliminary preparation of Instruments (cf. Section 6.2.2)
Preliminary run with a subset of projects (cf. Section 6.2.1)
Dissemination (cf. Section 6.2.1)
Used to evaluate onboard Instruments and find bugs on the tool
Creation of 378 PRs with a description of the tool and links for the survey
Tool acceptance analysis (cf. Section 6.3.2)
Demographics analysis (cf. Section 6.3.1)
Gathering of results
65 merges (17% have incorporated the diagram in the project)
Mostly individuals with computer science background, more than 5 years of professional experience from 24 countries
36 responses (almost 10% of conversion rate)
Tool ease of use analysis (cf. Section 6.3.3)
Tool usefulness analysis (cf. Section 6.3.4)
Tool intention to use analysis (cf. Section 6.3.5)
Qualitative analysis of the tool to answer the research questions

Figure 6.1: Study methodology steps and results.

6.2.1 Sampling

GitHub has a large community of contributors and open source projects, so we chose this platform to introduce the tool and invite contributors to respond to the online survey. This was achieved by creating pull requests contributing with infrastructure diagrams generated by the developed tool.

These pull requests were accompanied by a description of the project and a link to the survey. To improve the changes of review of the pull requests and to guarantee a good fit between the project and the developed tool, we used the advanced search functionality of GitHub with the following parameters:

- **Created after 2018-01-01**: To filter out older projects that could be using deprecated docker compose specifications. Version 1.6.0 of docker compose [4] introduces a new format for the docker-compose.yaml file, the one supported by Infragenie. This new version was introduced in 2016 and we are selecting projects that have been created two years later to increase the changes that they are using the supported specification.
- **With more than 200 stars**: The number of stars shows how popular a project is and is also an indicator that it is an overall good project with good acceptance by the community. From the various filters combinations we tried, the number 200 for the stars, was the one that returned better results. Lower numbers were returning projects with low activity and bigger numbers were returning too many educational and reference projects, like exercise books, courses and templates to produce docker-compose files.
- **With code changes after 2021-11-01**: This parameter was used to try to select only active projects. We are using a relatively large window of time not to restrict the search, since most recent dates returned very few results. We must remember that the results are a combination of all the filters, for example, the number of stars also has an influence on active projects returned.
- **Including the filename docker-compose**: Here, we are restricting the search to projects that use the supported infrastructure framework, docker-compose.
- **With the file extension yml or yaml**: The main docker-compose file is written in YAML, which usually has one of these two extensions.

The values for each parameter were found by experimenting, and manual verification of the results returned. The combination of these filters in the search query produced 22447 results. From these, we randomly chose 378 projects for our study. With this number we tried to sample a relevant portion of our population size, because it was impractical to disseminate the survey for all the 22447 projects. To find this sample of projects we have calculated the sample size selecting a margin of error of 5% so that the results can, mostly, reflect the views of the entiry population and a confidence level of 95% so that we can be certain that a big percentage of the population would give similar answers. Following this we used a popular sample size calculator[5] to get the needed sample size.

The first 100 projects were used to test the tool. With this first run, we were able to find and fix some processing errors and apply improvements. The most notable changes were made in the

[4]Docker compose release notes: https://docs.docker.com/compose/release-notes/

[5]Sample size calculator tool: https://www.surveymonkey.com/mp/sample-size-calculator/

way the tool searches for the infrastructure files, increased compatibility with different versions and styles of the docker-compose file and better fault tolerance. The first run was also essential to validate our study design and instruments which are described in the following section. At this stage, we corrected commit messages and typos in the pull request description and reached out to some users to understand if the available resources were helpful.

For the rest of the sample, we monitored the interactions with the pull requests, answering the reviewers' questions and trying to persuade their participation in the survey.

This sample of projects provided a good variety of diagrams, from complete ones to others not so complete, where edits were needed. This variation is excellent to provide insights for RQ2 and a feeling about the perception of users about the diagrams' completeness to help answer RQ1.

6.2.2 Instruments design

The questionnaire was designed and made available as an online form. The questions were designed to keep the response time small. They were written to be as precise and targeted as possible, combining close-ended and open-ended questions. However, the use of open-ended questions was limited to only capturing respondents' comments that were not anticipated by the other questions. Open questions were used to let users provide more information about their experience with the tool, which helped complement the answers to all three research questions. All the open-ended questions were optional. The close-ended questions have been created as a five-point Likert scale [57, 26] or single choice, and were mandatory.

The questions are organized around five main groups. The last four are directly aligned with the *technology acceptance model* [22] used to validate the developed tool: *a) participants characterization*, *b) Infragenie ease of use*, *c) Infragenie usefulness*, *d) intention to use Infragenie* and *e) improvements and open comments*.

To provide insights about the developed tool and to allow the respondents to give more informed answers to the questionnaire, the survey starts with a 2min video highlighting the purpose, functionalities and how to use Infragenie. The questionnaire is available in the study replication package[6] and Appendix B. The video can be found in the replication package and also on Youtube [7].

The potential respondents were invited to answer the survey in the description of the pull requests mentioned in the previous section. The description of the pull requests was thought to describe the purpose of the pull request, to introduce Infragenie, highlighting its edition capabilities with a direct link to edit the generated diagram and a direct link to the survey.

The web client developed (cf. Section 5.6), which is part of Infragenie and is used to request project analyses and edit diagrams was also designed as an instrument to help with the study by onboarding users in using the tool and providing valuable guidelines. As described in Section 5.6, the web client pages have different spaces and banners with the tool functionalities and descriptions of how to use them.

[6]Study replication package: https://github.com/ricardojaferreira/infragenie-replication-package

[7]Infragenie presentation video: https://youtu.be/XF5n-kkLiAw

6.2.3 Research variables

The demographic questions were used to characterize the respondents and help interpret the responses. Our study used *academic background*, *years of professional experience*, *UML knowledge*, and *experience with plain text or code to generate diagrams* as **independent variables** to segment results and gain insights into how the background and experience vary the perception of the developed tool. We have also used the *country* information as an independent variable to measure the diversity of respondents. These variables help us better interpret how the developed tool improved the comprehension of the architecture and the value given to it by users, which is essential to respond to RQ2 and RQ3. We can even relate how users' experience affected their perceived ease of use, usefulness and intention to use, in line with the technology acceptance model (TAM).

As for the **dependent variables**, we have used metrics like *ratings of the diagram generated*. Where we asked users to rate the diagram in terms of its completeness, precision and understandability to understand better the value given by users to the diagrams generated and with that answer RQ3, this variable also helped to get a better insight on how easy it was to interpret the diagrams provided by Infragenie, which shed some insights to answer RQ2. Another dependent variable used was *editor's usefulness*. Here we tried to capture the editor value and ease of use as an instrument to add value to the generated diagram and experiment or plan new changes in the architecture, which gave insights to answer RQ2 and RQ3. To better respond to RQ1, we have also added metrics to measure *how users compare other diagrams with the ones generated by the tool*, the *number of projects that are using the diagrams produced by the tool* and the *willingness of users to continue to use Infragenie*. The willingness to use measures not only the usage in new projects to generate diagrams but also the continued usage to keep those diagrams updated. This also provides useful information for RQ3. The number of projects using Infragenie was not a direct question on the survey but rather an analysis of the outcome of the Pull requests generated.

6.2.4 Data analysis design

To create a narrative around the data collected and to understand it we have created different visualizations of the data, which can be seen on Section 6.3.

Participants background and diversity of the respondents is analysed with the help of distribution graphs and pie charts, showing occurrence percentages for each group. We have also used some aggregations in the form of stacked bar graphs to understand the experience with UML and related tools of the participant population.

Stacked bar graphs, are useful to quickly interpret the information provided by Likert scale questions, cleary showing the distribution of the five point scale used (*Strongly disagree*, *Disagree*, *Neutral*, *Agree* and *Strongly agree*). These type of visualizations have been used in all Likert scale questions, like for example to understand how *useful* was Infragenie.

An also important classification was the correlation between experience and perceived usefulness, to infer if the participants experience and background increases their willingness to use tools that can improve and keep the documentation updated. This verification was accomplished with

Table 6.1: Engineering research standard: Essential attributes followed.

Attribute	Where it is applied
describes the proposed artefact in adequate detail	cf. Sections 5.3, 5.4, 5.5, 5.6
justifies the need for, usefulness of, or relevance of the proposed artefact	cf. Sections 4.1, 4.2
conceptually evaluates the proposed artefact; discusses its strengths, weaknesses and limitations	cf. Sections 5.3, 5.6, 5.7
empirically evaluates the proposed artifact using a method for which a clear and convincing rationale is provided	cf. Sections 6.2, 6.3
discusses state-of-art alternatives	cf. Sections 3.3, 3.4
empirically compares the artefact to one or more state of the art alternatives	cf. Section 6.5

the usage of annotated heatmaps, were it is possible to see how the variables relate and the trend of the relation.

6.2.5 Framing in the ACM SIGSOFT Empirical Standards

This study was designed to provide insights about the developed tool, Infragenie, as a means to promote evolving architecture diagrams that are automatically generated to be used in the documentation. This objective falls into the description of **Engineering research methods** as described in the Empirical standards [42], which documents the expectations for empirical research in software engineering. Engineering research is characterized as *"Research that invents and evaluates technological artefacts"*. A characterization that can be easily applied to this work since we are proposing a tool or a technological artefact, and we wish to evaluate it to understand if it solves the problem statement of this thesis, defined in Chapter 4. The evaluation method used for this engineering research is a Questionnaire survey[8] used with mostly closed-ended questions (cf. Section 6.2.2) to analyze participants answers which were sampled using the strategy depicted in Section 6.2.1.

Engineering research must show a set of essential attributes that are captured in Table 6.1, together with the place where this work applies them.

There is also a set of desirable attributes to follow by such a study. The desirable attributes followed by this work are depicted in Table 6.2

[8]ACM page for Questionnaire surveys:
https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=QuestionnaireSurveys

Table 6.2: Engeneering research standard: Desirable attributes followed.

Attribute	Where it is applied
provides supplementary materials, including source code and datasets	cf. Appendix B
justifies any items missing from replication package	NA*
discusses the theoretical basis of the artefact	cf. Section 5.1, 5.4
includes one or more running examples to elucidate the artefact	cf. Appendix A
evaluates the artefact in an industry-relevant context (e.g. widely used open-source projects)	cf. Section 6.2.1

*All the necessary items to replicate the study are on the replication (cf. Section B.1).

In the list of desirable attributes, we have not included the attribute *justifies any items missing from replication package based on practical or ethical grounds* because our universe of answers is not big, so there was no necessity to apply complex data analytical strategies.

6.3 Data Analysis

From our sample of 378 pull requests (cf. Section 6.2.1) we have obtained 36 responses to the questionnaire. Which represents 9,5% of conversion rate. Considering that online surveys have on average 30% of conversion rate [64], the number achieved is relatively small compared to that standard. Although, several factors contribute to the reduction of conversion rates, among them the delivery method of the survey, size of the population and number of days the survey was left open could lead to a variation of more than 60%. Examining our example, the delivery method used was not ideal, despite being the needed method to validate and introduce the tool to an unbiased audience. And it was not ideal because, we observed that some pull requests did not had any interaction, most possible because the project was not active (we have used projects with code changes after 2021-11-01 cf. Section 6.2.1) or the contributors did not have the necessary time to look into it. Also, some GitHub users received the contribution as an advertising to a commercial tool and decided to close the pull request. To add to that, some projects use automated tools to run validation steps on the pull requests and require SLA agreements to accept contributions, whithout these validations the pull requests are marked as invalid and did not receive any review until they are valid. We could not account for all these variables. Having this in mind the attained conversion rate can be explained and even considered good.

Looking only quantitatively to the number of 36 responses we believe that they can provide the necessary information to respond to the research queries. In the follwing sections we will analyze these 36 responses, exploring different ways of grouping the data.

6.3.1 Participants characterization

One of the concerns of this study was to reach a broad and diverse audience from the developer community, to get the viewpoint from different contexts and different levels of knowledge. To evaluate this heterogeneity, we have included in the questionnaire questions to evaluate the respondents' profiles.

The answers revealed that we had participations from 24 different countries, almost equally distributed among them, with China, Germany and USA with a few more representations, as revealed in Figure 6.2.

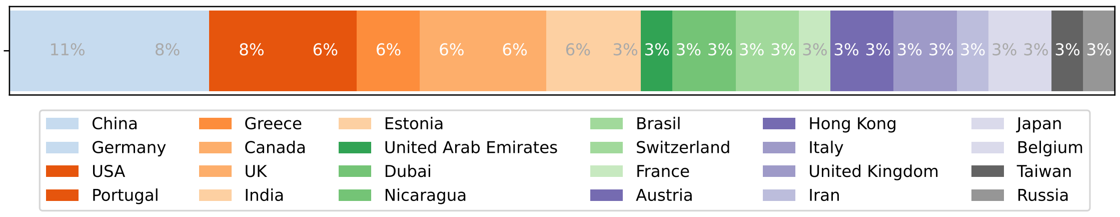


Figure 6.2: Distribution of respondents by country.

Regarding academic background and professional experience, almost every participant had a university degree (95%) and three quarters of the respondants have a background in computer science, as can be seen in Figures 6.3 and 6.4. Furthermore, a high percentage has more than 5 years of professional experience (78%), which reveals a very proficient group, most likely with a formed idea on the importance of documentation. The professional experience data is represented in Figure 6.5.

Regarding academic background and professional experience, almost every participant had a university degree (95%), and three-quarters of the respondents have a computer science background, as seen in Figures 6.3 and 6.4. Furthermore, a high percentage has more than five years of professional experience (78%), which reveals a very proficient group, most likely with a formed idea on the importance of documentation. The professional experience details are represented in Figure 6.5.

These demographic details show that we have reached an audience of computer science related individuals with different backgrounds, showing that we have chosen the distribution platform correctly.

Besides the demographic data, we also tried understanding the knowledge level around UML diagrams and textual tools to generate diagrams. Those are important metrics to understand how comfortable the respondents are with the diagrams produced by the tool and the editor. To evaluate

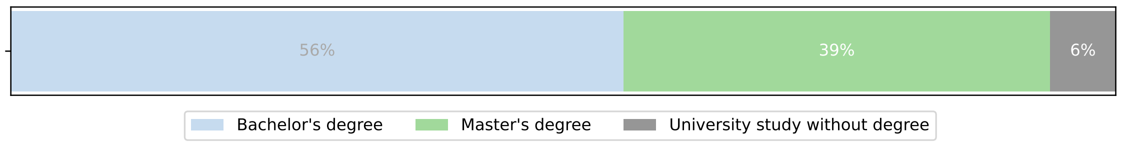


Figure 6.3: Respondents educational attainment.

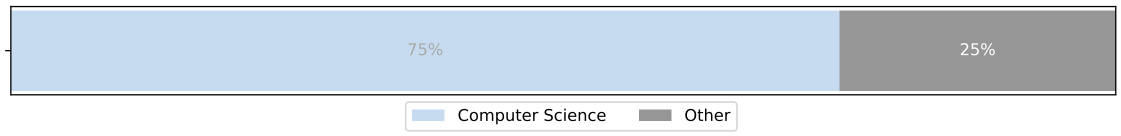


Figure 6.4: Respondents area of study.

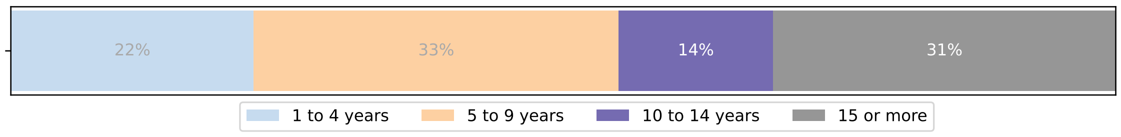


Figure 6.5: Respondents professional experience in years.

this knowledge, we have asked four questions using a Likert scale with five points. The questions asked were:

Q1 "I have experience with tools that allow the creation of diagrams from plain text or code (e.g. PlantUML, Mermaid.JS, others)"

Q2 "I am familiar with the Unified Modeling Language (UML)."

Q3 "I am familiar with UML component diagrams."

Q4 "I am familiar with UML deployment diagrams."

The answers to these four questions are in Figure 6.6.

Overall, the respondents are comfortable with UML and creating diagrams with text-based tools. The lower point in our Likert scale is *Strongly disagree*, which no respondent answered in any of the questions, showing that even those less comfortable with these topics have some knowledge about them.

6.3.2 Acceptance

To measure acceptance, we based our analysis on the pull requests' state. In Figure 6.7, it is possible to see the percentage for each state among the 378 pull requests created.

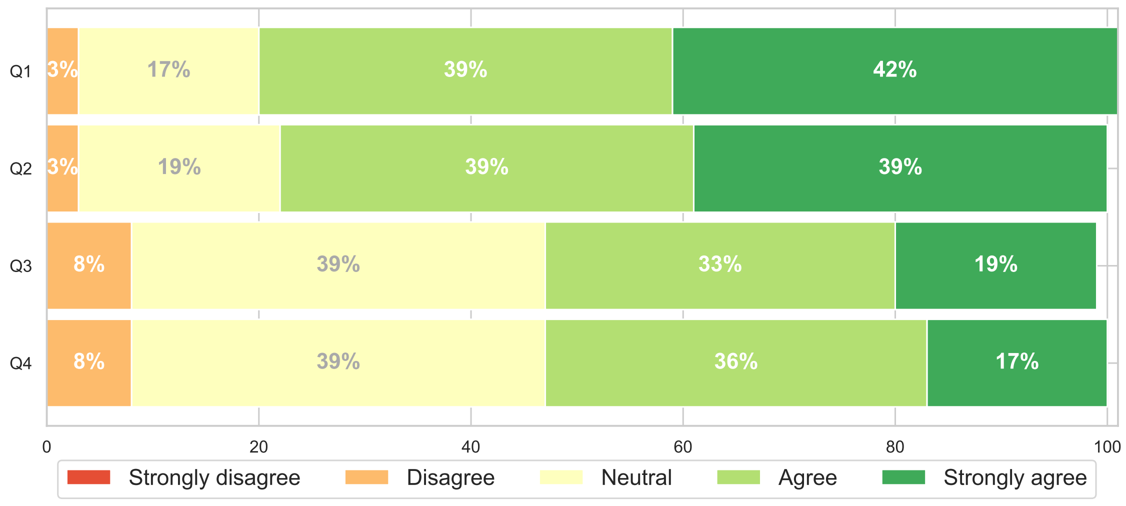


Figure 6.6: Respondents knowledge level.

Due to decimal rounding some bars are not reaching 100% or are exceeding it.

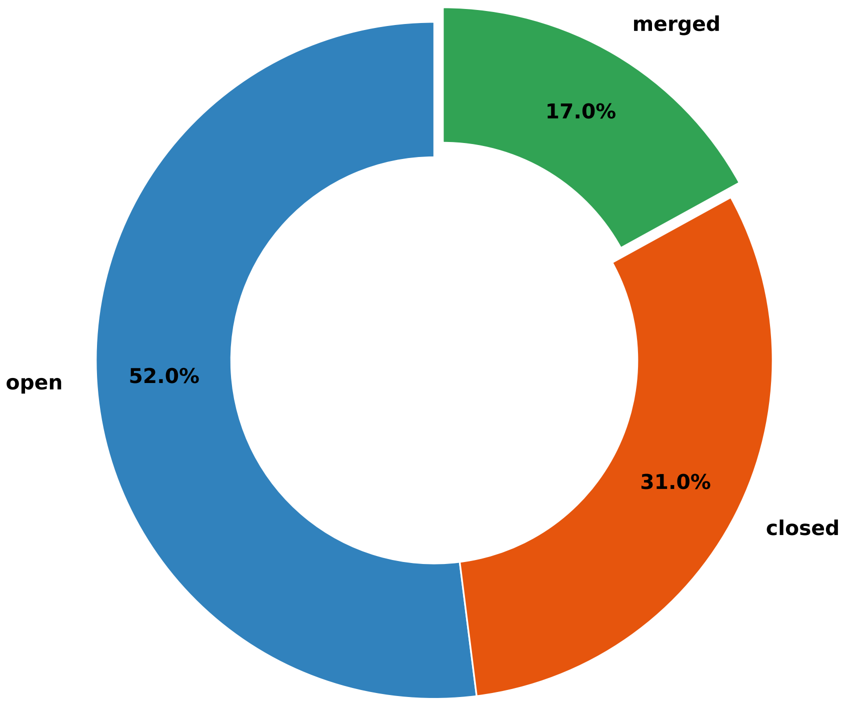


Figure 6.7: Distribution of pull requests states at the end of the study.

52% of the pull requests remained in the open state, which helps explain the low conversion rate of the questionnaire, as discussed at the beginning of this chapter. Nevertheless, this alone does not imply that the tool had low acceptance. Several reasons exist to explain why the pull re-

Table 6.3: Users reasons to close pull-requests.

Although this is a cute little diagram, it doesn't provide any value to the project and so I won't be adding it. This would be a great addition to a larger project.
Thank you for this PR but at the moment we don't feel we need such a document, as the architecture is quite simple.
I took a look at your picture. I think your idea is great, but the current generated graphs are not beautiful enough. I wish you can improve it better.
Sorry, seems like an ad to me.
Thank you for the contribution. However we don't need this at the moment.
very nice app, it's displayed very clearly, but I think maybe this pic not Suitable for my repo. So I will close this pr. thx!
We are not giving free advertising.
This repo serves as a tutorial for other projects. We have several docker-compose files and theres a diagram just for one.

quests remained in the open state at the end of the study. Some projects run verification pipelines on the pull requests, which require a specific structure in the pull request. Others require CLA licence agreements. Our tool was not complying with any of these aspects, so the pull requests were not reviewed. Unfortunately, the GitHub search feature does not allow filtering projects with these characteristics. Another reason may be related to the fact that the project has few contributors, and so, being a project with high activity may slow down the process of reviewing pull requests. Filter projects by the number of contributors could be done, but that would significantly impact the results returned, so we decided not to apply that filter parameter.

Looking at projects in the closed state, they represent 31%, which is a relatively high percentage. We have contacted the project owners to understand better the reasons for closing the requests. Some of the answers are aggregated in Table 6.3. The answers reveal that some users mistakenly took the contribution as an advertisement for commercial software and did not give a change to the tool. Due to having small infrastructures, others found the diagram not very helpful or were hoping for a neater representation. The answers also revealed that some projects were tutorials or courses about Docker, meaning that a tool like Infranie does not provide valuable information. Tutorials or similar projects could not be filtered off from the search.

For all the reasons discussed, we regard 17% of merged pull as very positive. Many users found the diagrams helpful and decided to add them to their documentation.

This analysis of the state of the pull requests shows evidence of what we are trying to answer on RQ3. We can say there is a willingness to use Infragenie, although the tool needs some improvements to increase its acceptance rate, as noted by some of the comments in Table 6.3. Some of those improvements include better quality diagrams. Also, such a tool may not be suited for all projects or not be given enough importance, especially for projects with simple and small architectures, which is expected. From the number of users that merged the pull requests, we can conclude that those users agreed that the diagrams produced led to better and updated documentation on the project, which gives a positive answer to RQ1.

6.3.3 Ease of use

In the section for evaluating ease of use in the questionnaire, we have asked the following six questions:

I think that with Infragenie, it is easy to...
Q1 "create architectural diagrams."

Q2 "keep architectural diagrams updated."

Q3 "keep architectural diagrams consistent with the source-code."

Q4 "add new information (not inferred from the source-code) to the diagrams."

Q5 "change the generated architectural diagrams."

Q6 "verify how changes may impact the architecture."

The answers to these questions are aggregated in Figure 6.8.

Looking at the graph, the positive side, represented by the green portion of the bars, is more prominent, being more than 50% in all six questions. Creating diagrams with the developed tool was appointed as the easiest thing to do in the application, while verifying impacts by changes in the architecture was the most challenging but not significantly difficult. To get a better insight into the data, we have looked at the individual answers to understand if the users' knowledge of UML and text-code editors, like plantUML, evaluated in the previous section, could influence the ease of use perception. All the relations can be seen in Section C.1. The ease of creating and changing diagrams were the most significant correlations, and can be seen in Figures 6.9 and 6.10, respectively.

Creating diagrams is not affected by user knowledge. Even users with lower knowledge of UML or text-based tools have agreed on how easy it is to create diagrams. On the other hand, having more knowledge makes changing diagrams easier. This reveals that the editor may need some improvements, like for example being more graphical oriented to not depend on the plantUML knowledge. Overall, users were comfortable using the tool, revealed by the bigger saturation in the top left corner of the heat maps analysed, which can be a good indicator of willingness to use helping to provide evidence for RQ3.

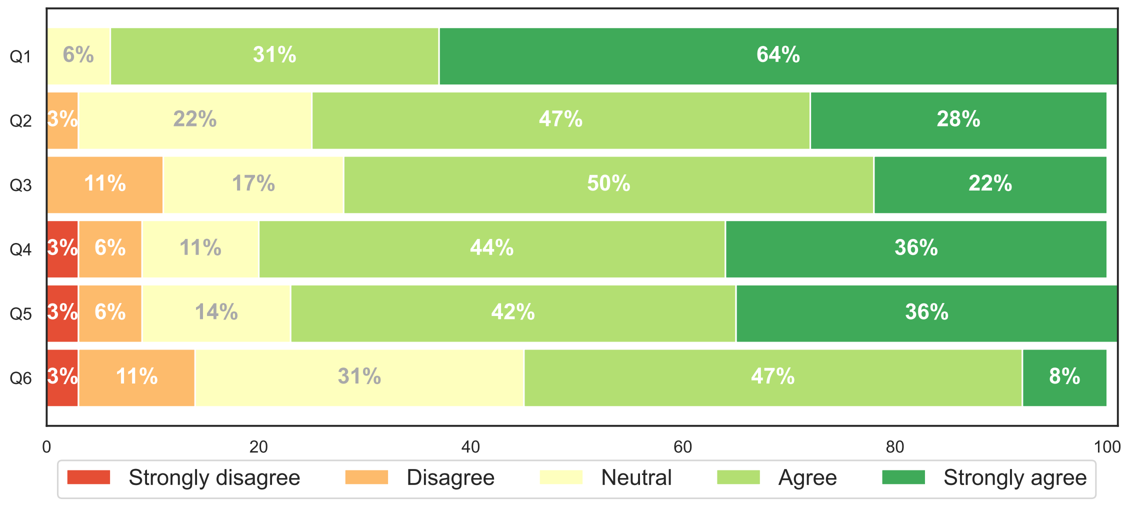


Figure 6.8: Infragenie ease of use evaluation.

Due to decimal rounding some bars are not reaching 100% or are exceeding it.

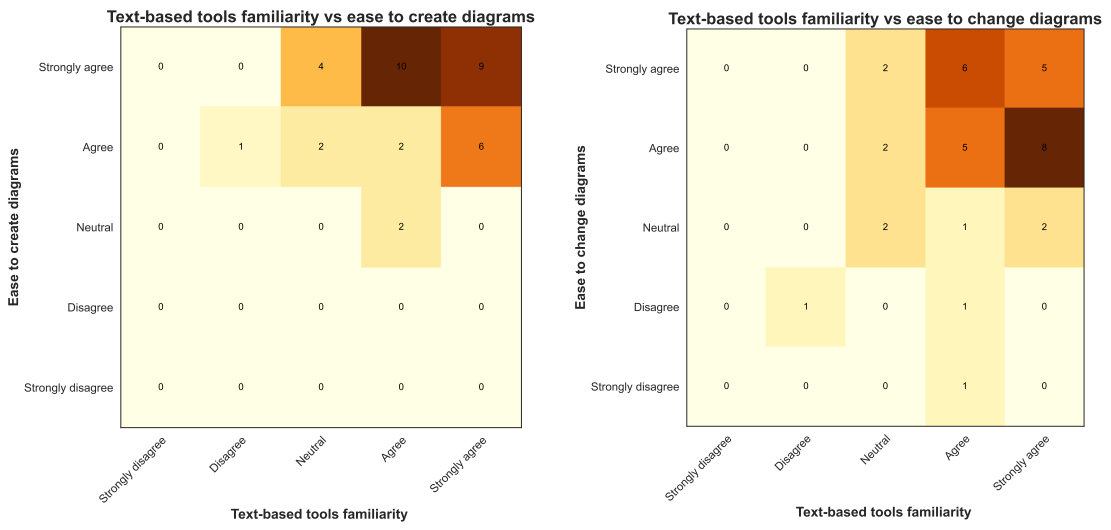


Figure 6.9: Relation between text-based tools (like plantUML) knowledge and creating and changing diagrams with Infragenie.

6.3.4 Usefulness

Four groups of questions measured usefulness. The first one was targetting the usefulness of the documentation. The second was the usefulness of using the tool to plan changes in the architecture, the third group the usefulness of the diagrams produced and the fourth group targetting the edition capabilities of Infragenie.

In the first group, we asked the following: *I think that Infragenie is useful to help ensure that architectural documentation is...*

Q1 "complete."

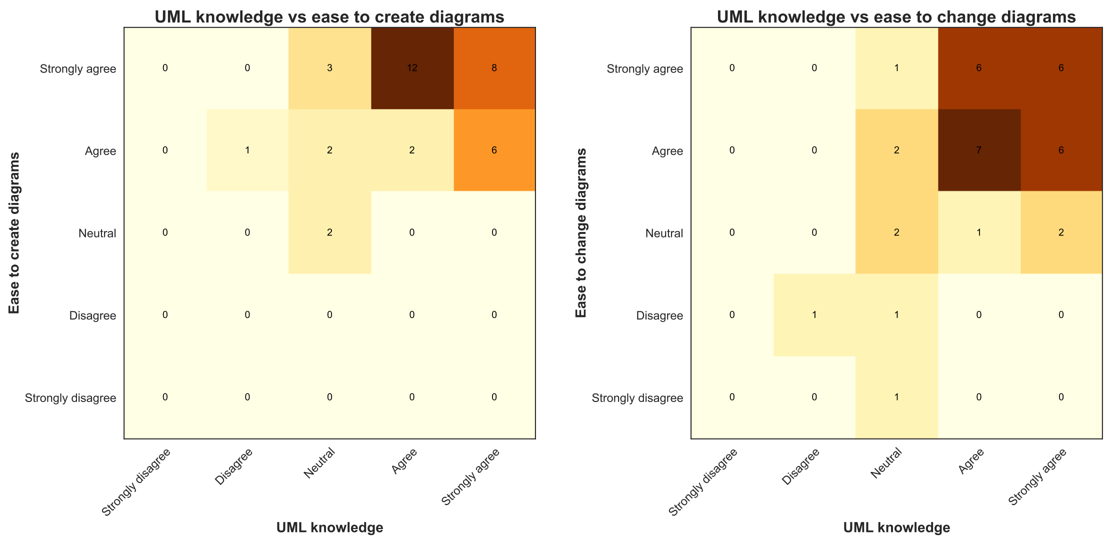


Figure 6.10: Relation between UML knowledge and creating and changing diagrams with Infragenie.

Q2 “consistent.”

Q3 “precise.”

Q4 “easier to consult.”

Q5 “easier to understand.”

Q6 “easier to maintain.”

Q7 “easier to update.”

The results for this group of questions are in Figure 6.11.

In the second group, the question asked was: *I think that Infragenie is useful to help to...*

Q1 “plan future changes on the architecture.”

Q2 “understand the impact of changes in the architecture.”

Q3 “be more confident while introducing architectural changes.”

Q4 “implement architectural changes faster.”

Figure 6.12 shows the answers from the second group.

To check on the usefulness of the diagram, the question was: *I think that the diagram provided by Infragenie shows...*

Q1 “a complete view of the architecture”

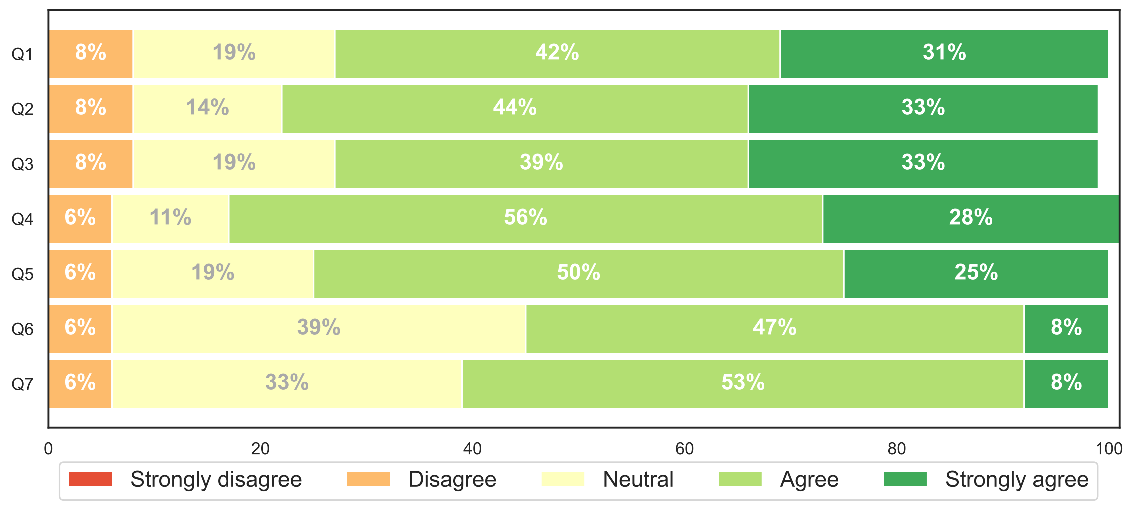


Figure 6.11: Infragenie usefulness for project documentation.

Due to decimal rounding some bars are not reaching 100% or are exceeding it.

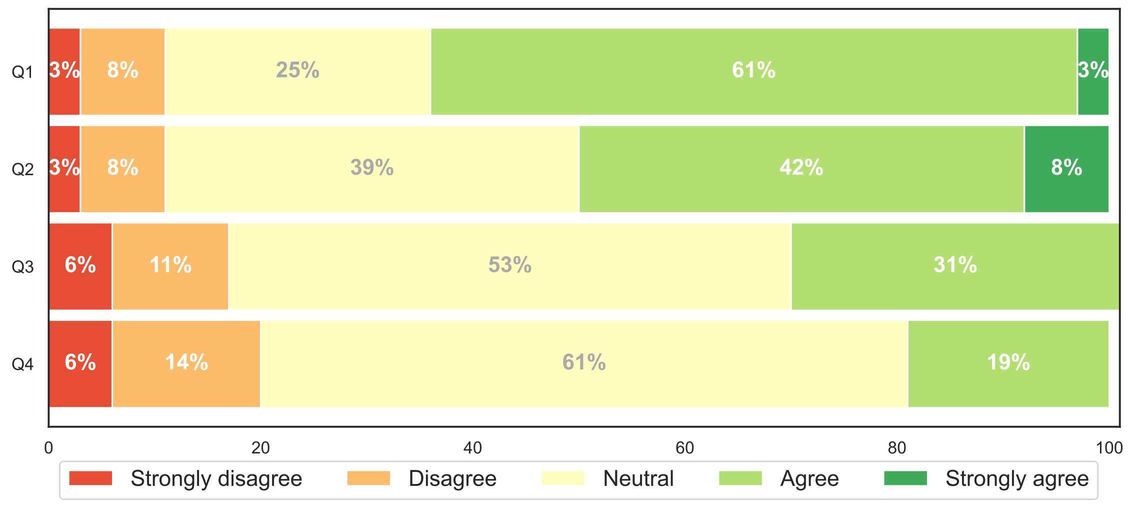


Figure 6.12: Infragenie usefulness for verifying future changes.

Due to decimal rounding some bars are not reaching 100% or are exceeding it.

Q2 "the most relevant elements of the architecture."

The results for the third group are in Figure 6.13.

To get more insights on the edit capabilities, we used the question: *The edit capabilities of Infragenie allow to...*

Q1 "increase the diagram value."

Q2 "make the diagram more consistent."

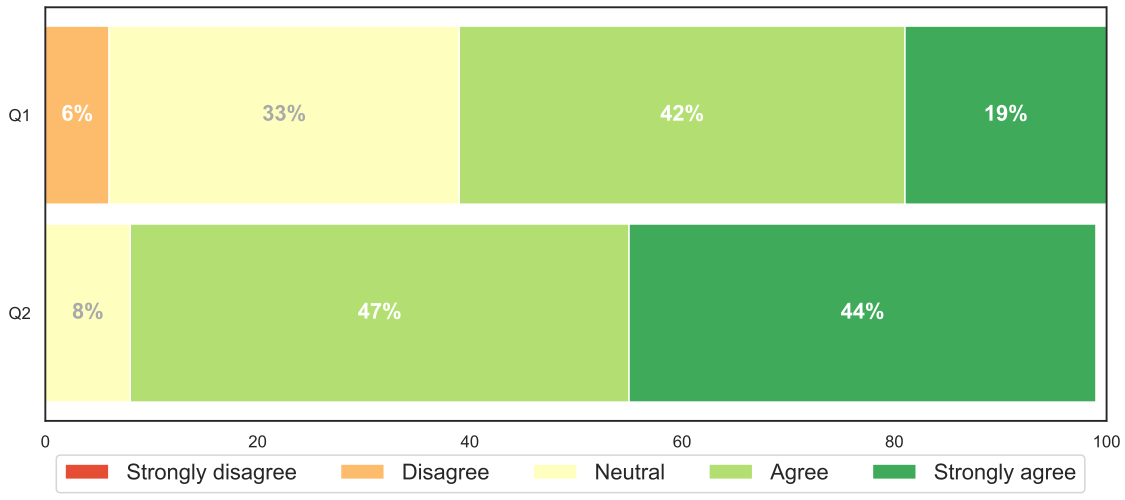


Figure 6.13: Infragenie usefulness of the diagram produced.

Due to decimal rounding some bars are not reaching 100% or are exceeding it.

Q3 "representing details that were not automatically captured."

The obtained answers are grouped in Figure 6.14

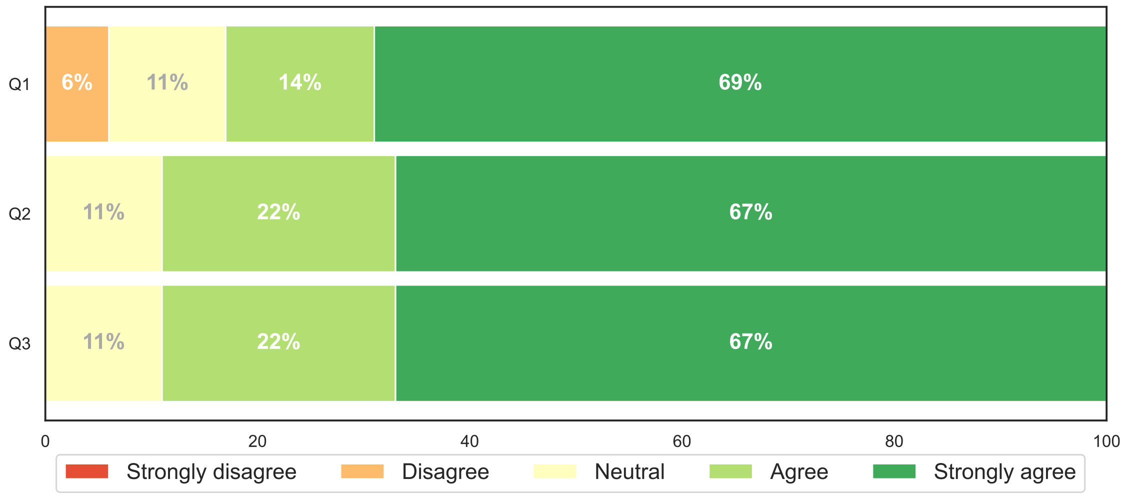


Figure 6.14: Infragenie usefulness of the edit capabilities.

Due to decimal rounding some bars are not reaching 100% or are exceeding it.

The usefulness of the developed tool for documentation (cf. Figure 6.13) was very positive, without any strong disagreement and with most answers between agreeing and strongly agreeing. This is especially useful to corroborate that a tool like Infragenie can lead to better and updated documentation, answering in that way to RQ1. The usefulness to understanding the impact and plan for future changes on the architecture (cf. Figure 6.12) was not consensual, but was mostly

Table 6.4: Users answers to what was not being represented by Infragenie.

Other parts of the architecture are not being managed by docker compose and it would be nice to have a view of those pieces also.
It should be possible to choose the docker-compose file to analyze. For example dev and prod docker files generate different diagrams.
It lacks the analysis of other infrastructure files, rendering the diagram very incomplete.
Some connections between services are missing.
Overall the information was there.
All the essential was there.

positive. Despite this, they did not see the same usefulness of being faster and more confident while introducing changes in the architecture. This analysis can be further complemented with the open-question about what was not being represented by infragenie. The answers to this question are grouped in Table 6.4.

These answers revealed that a tool like Infragenie should look into more artefacts than just the infrastructure ones. This information helps us answer RQ2 with the following statement, having a diagram can improve the knowledge and help to plan future changes, but for implementing these changes, the infrastructure diagram alone is not enough, since the architecture can be defined in many more places. Furthermore, the developed tool probably needs other mechanisms like generating and updating the infrastructure artefacts.

The responses on the diagrams' usefulness provide even more evidence of the diagram value, showing that the vast majority of the users found the diagrams complete and showed the most relevant elements of the architecture.

Regarding the edition capabilities (cf. Figure 6.14), more than 60% of the users strongly agreed that this feature is useful and increases the diagram's value. This shows that adding the possibility to add changes in automatic recovery tools is a much-appreciated feature, to allow representing not captured details or making the diagram more consistent.

As in the previous section, we have crossed the results from the usefulness questions with the user's knowledge. All the relations can be seen in Section C.2. From the first group of questions, we can highlight the relation between knowledge and how useful it is a tool for maintaining documentation, as depicted in Figure 6.15.

The relation shows that knowing UML and related tools can help but is not mandatory. Some

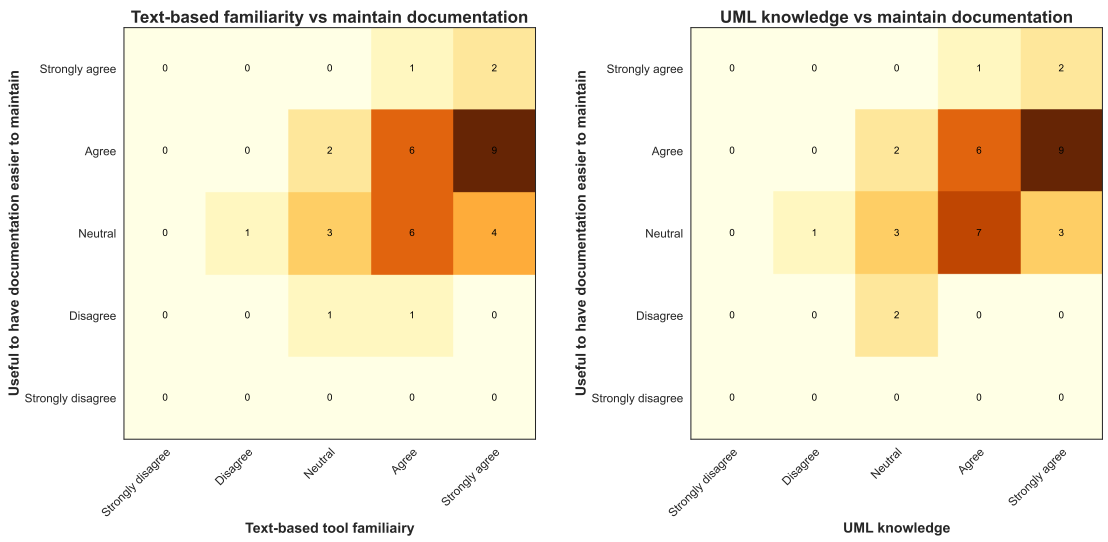


Figure 6.15: Relation between user knowledge and maintaining documentation.

users, despite having lower knowledge still find the tool usefull enough for maintaining the documentation.

As for the usefulness of Infragenie for planning, understanding and implementing changes, depicted in Figure 6.16, we have found the relation between user knowledge and planning future changes more scattered. In this relation, it is easy to see that users with knowledge find the tool usefull to plan changes, while, for users with less knowledge its difficult to make assumptions, since there isn't a noticible tendency.

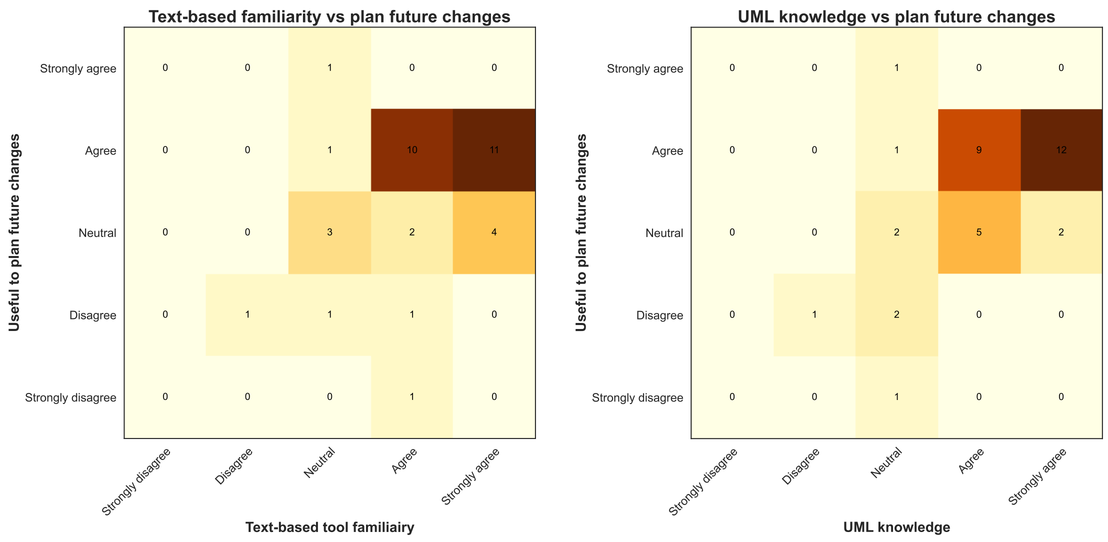


Figure 6.16: Relation between user knowledge and planning future changes.

One last relation worth noticing is the relation between user knowledge and the added value of the diagram brought by the edit capabilities. This relation is depicted in Figure 6.17 and shows that the users tend to agree that the edit capabilities increase the diagram value, despite the knowledge.

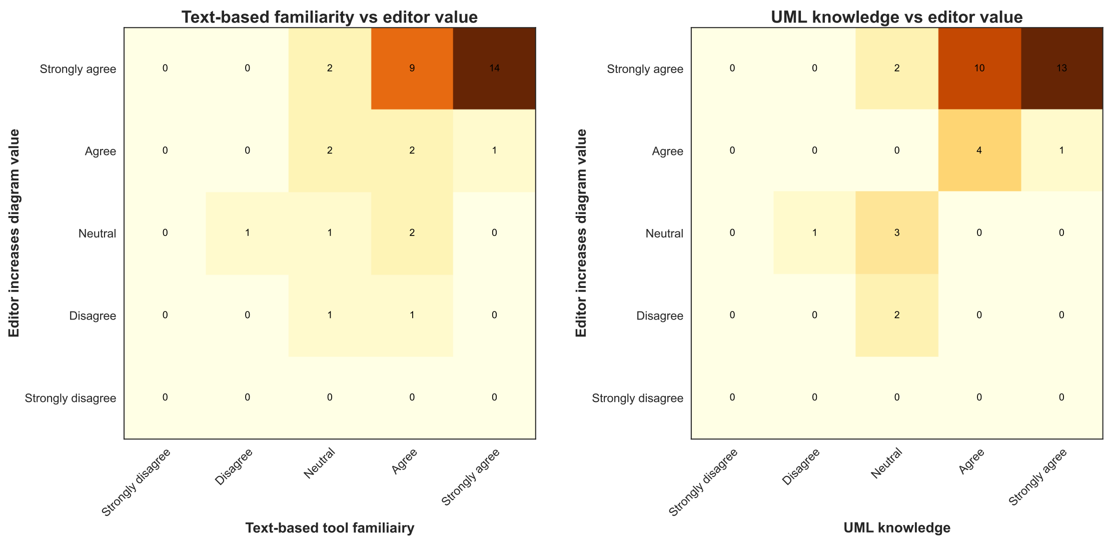


Figure 6.17: Relation between user knowledge and edit capabilities allowing to increase diagram value.

We can now answer partially to RQ3 and RQ2 by saying that the tool was perceived has having a good value, especially its diagrams, increased by the fact that it is possible to edit them, which can be used to get more knowledge on the architecture and plan future architectural changes. This perception of the diagram value can also indirectly lead to conclude on the usefulness of Infragenie for the documentation, providing insights of what we are trying to evaluate on RQ1. Furthermore, the editor plays a pivotal role in the application, leading to increased adoption, especially if built with the low assumption of the user's knowledge and good usability. The editor allows more consistency and completeness of the diagrams render them more suitable to be a source of truth and reference.

6.3.5 Intention to use

To evaluate intention to use, we started to try to understand if the evaluated projects had an architectural diagram before and, if so, how that diagram compares with the one generated by the developed tool. The answers to this question can be seen in Figure 6.18.

This part of the questionnaire was also accompanied by a text box for users to leave their comments. The most relevant ones are in Table 6.5.

In the universe of projects with older diagrams, Infragenie generated a better diagram in more than 75% of the cases, showing more details, being UML compliant and even detecting errors in the infrastructure artefacts. In the other cases, Infragenie was not adding anything new (14% neutral) or unable to detect everything (7% worst). As detected during the analysis of the acceptance of the tool, some projects have simple infrastructures, and a tool like Infragenie is not valued in such contexts.

Still to evaluate the intention to use we have asked: *I am considering to...*

Q1 "add diagrams generated by Infragenie to my projects."

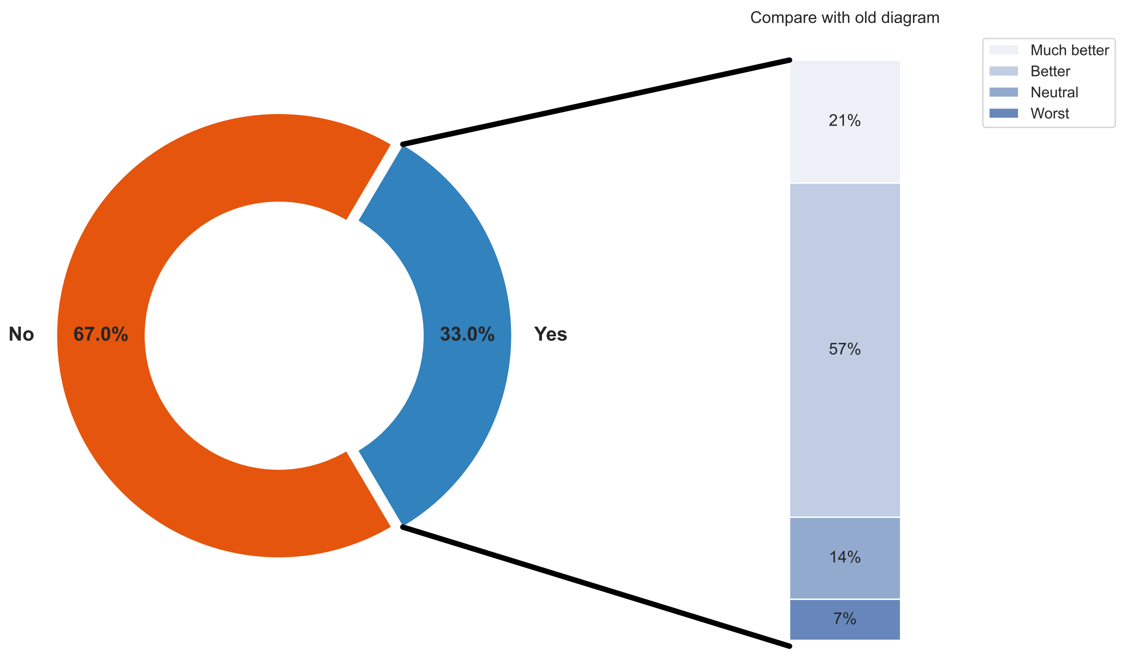


Figure 6.18: Percentage of projects that have architectural diagrams before Infragenie's analysis (YES) and how they compare.

Q2 "use infragenie to visualize and plan changes in the architecture."

Q3 "use infragenie in new projects."

The answers can be seen in Figure 6.19

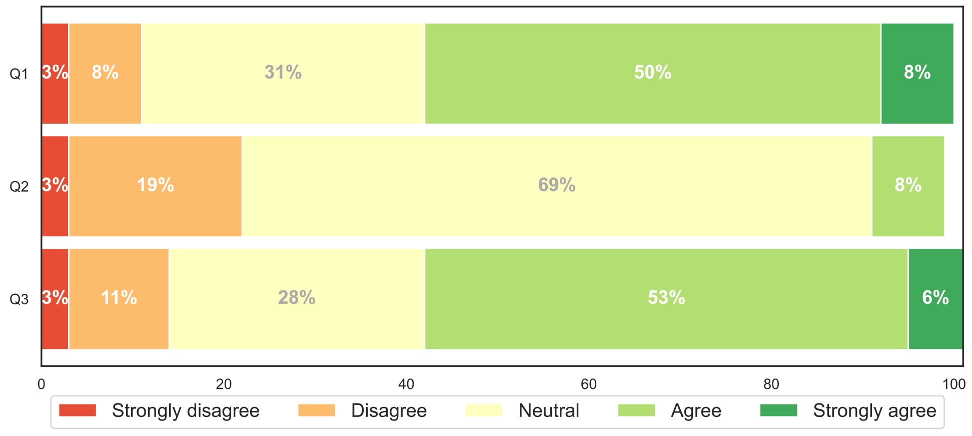


Figure 6.19: Intention to use Infragenie.

There is a high intention to use Infragenie in current and new projects, but not so high an intention to use Infragenie to plan future changes. These results prove what we have been observing so far: The tool is helpful, and the users are giving value to the diagrams generated. The edit capabilities, despite needing some improvements, for example, to move away from PlantUML syntax and

Table 6.5: Users comments while comparing old diagrams with the one generated by Infragenie.

The actual documentation is not comprehensive and is old.
It shows some more details like properties of services and the ports are easier to follow.
Not better, not worst, but different. It has the advantage of being auto generated.
Some connections were missing.
It's more formal.
The previous diagram was really simple and not UML compliant.
The infrastructure was very small, no new addition was made.
It detected a typo on one of the components.

to be more helpful at planning changes, are good enough to increase the diagram completeness, which was the main objective of the feature.

6.3.6 Improvements and comments

To conclude our analysis, we tried to understand if the users were valuing the fact that the diagrams produced were UML compliant and if automatic detection of architectural changes was something that could increase the adoption. For that, we had one last question that goes like this: *Infragenie would be better if ...*

Q1 "more UML elements were available to use."

Q2 "it wasn't restricted to UML elements."

Q3 'it automatically detected architectural changes (keeping the diagram always updated without manual intervention)."

In the end, the users were also invited to leave comments so that we could get a better vision of what was on user's minds about the tool. Figure 6.20 shows the results for the question, and Table 6.6 aggregates the most useful comments.

A significant percentage of users agree that having the diagrams compliant with UML is better, hoping to see more UML elements available and discouraging the usage of non-UML compliant elements. Also, having automatic detection features is seen as a good improvement for the tool.

Table 6.6: General users comments about Infragenie.

Some sort of pipeline plugin to suggest changes. Also, opening a pull request is a bit invasive, although understandable why it works like that. Having the tool integrated into CI tools would be better.
It seems a cool application but needs some improvements.
Nice tool. Hope to see compatibility with more technologies (ansible, chef, others)
It needs to look into all the architecture files on the project and not only to docker-compose. Other than that the tool is nice and can be very useful.
Having to go to the webpage is not very practical. Auto-generating documentation should be more straightforward.
The application is still immature, the way it interacts with the code by creating pull requests is not ideal, and the editor needs some improvements.
The editor needs a massive improvement. Other than that the tool is nice.
The editor should allow more freedom.

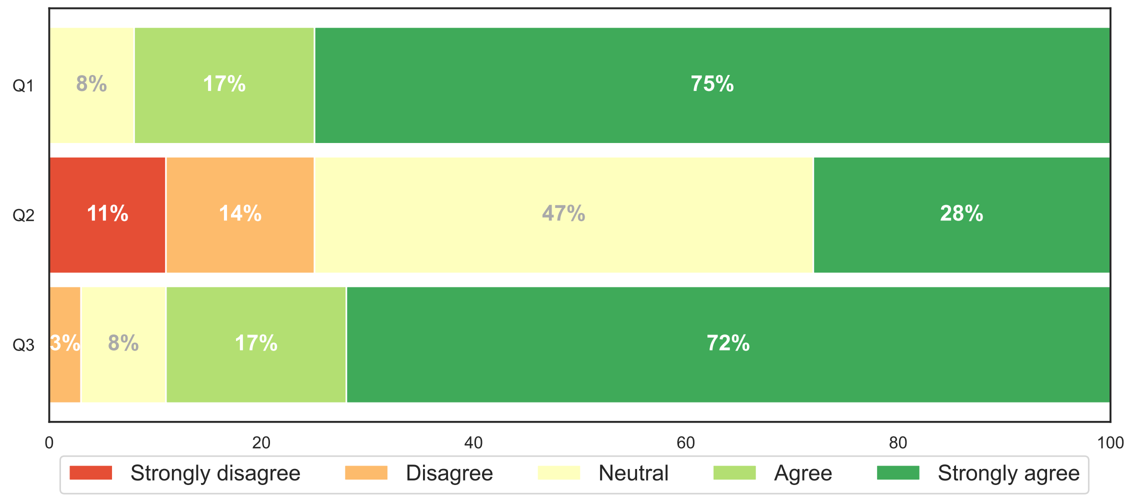


Figure 6.20: UML compliance and automatic detection valorization.

The comments strengthen the conclusions and observations made in previous sections. We have users unpleasant with the fact that the application opens a pull request, which was observed during the acceptance analysis as one of the reasons to not give a change to the tool. The users also expect a better editor, a more straightforward flow to generate the diagrams and a broader analysis of the architecture artefacts, not only docker-compose.

6.4 Threats to validity

To design this research, we have also thought about conditions that could cause deviations in the results and compromise the study's validity. Knowing those conditions helped us to trying to mitigate them and provided important context to our interpretation of the results. The remainder of this section lists the threats to the validity of the conclusions in this chapter.

Participant's lack of knowledge: The developed tool addresses problems related to the generation of software architecture diagrams. While software architecture is a broader concept that a bigger audience can know, the different types of notations to create the diagrams are not so generalised. Since our tool was creating UML diagrams, we were hoping to find a group of individuals familiar with the notation to be able to understand the generated diagrams and, so, be able to evaluate them. Also, how a user evaluates the tool, especially the editor, could be significantly influenced by its knowledge of UML and its familiarity with PlantUML, making it harder to understand how to use the tool. We were looking for a group with different levels of experience to get different views of the usability and value of the editor. To identify possible biased answers due to lack of knowledge, we tried to assess the respondent's level of familiarity with these topics. To reduce bias changes, we also tried to find a group of individuals with a computer science background, targeting projects on

GitHub. We had 75% of the respondents from an area of study related to computer science and professional experiences between 1 and more than 15 years. Even though the respondents claimed to have high knowledge of UML and plantUML, the universe of respondents was diverse enough to validate the study's conclusions.

Sample size: The sample size for this study was relatively small. A bigger group of respondents would give more strength to the conclusions. Allowing grouping of the results by respondent profile to verify trends in those groups, for example. That said, this treat cannot be disregarded but we believe that the insights provided are useful and can be usefull to plan future iterations of this study.

Oversampling: The diversity of the participants can also have a significant impact on the results. As mentioned in previous sections, having a big group of individuals without knowledge of topics like UML can negatively impact how the tool is evaluated and vice versa. To mitigate this threat, we have used open source projects on Github, which resulted in a very diverse group of respondents, as evidenced in Section 6.3.1. With this heterogeneity among the respondents, we are not expecting to see a significant difference in the results analyzed if a more extensive set of participants responded to the questionnaire.

Duration of the study: One of the characteristics of a tool like Infragenie is its ability to follow the project evolution, i.e., updating the architectural diagram every time changes are introduced to keep the diagram updated. Given that architecture and infrastructure changes do not happen very often, evaluating this evolution properly was not possible during the duration of the study. Also, since the study depends on the availability and willingness of the participants to provide feedback, it would be challenging to guarantee responses after a certain trial period. To overcome this, we have prepared a set of instruments to highlight the tool's features in a short time and allow informed answers.

Suitability of the analysed project: One concern for this study's validity was finding projects with relevant architectures that could be analysed by the developed tool. Using the tool in projects where the analysis would result in irrelevant diagrams or, even worst, where the application would not be able to generate a diagram could influence the results very negatively. For this reason, we tried to filter those projects using the advanced search features from GitHub. Given that the filter is not very accurate, we found some respondents questioning the validity of the diagrams for their projects, as we could confirm by the comments they have left. Nonetheless, this threat was reduced with the instruments prepared, allowing even participants that received less relevant diagrams to understand the tool. We believe their experience did not influence the answers.

Lack of guidance and interference: The study was designed to happen without manual intervention by researchers. The potential respondents received a diagram generated by Infragenie and were asked to answer a survey about it. This approach relies a lot on the tool's simplicity and the effectiveness of the materials we make available to participants to explain what it does and how it works. This could lead to participants responding to the survey without a good understanding of the tool and what problems it tries to solve. To reduce this threat, we sought feedback from participants by sending them messages to better understand some decisions and tried to answer all the comments left on the pull requests. However, looking at the overall positive evaluation of the tool, we can assume that a good portion of the respondents tried the tool and analysed the resources created to onboard the tool, reducing the effects of this possible threat.

6.5 Discussion

During this chapter, we presented the conducted study used to prove evidence of this thesis hypothesis by answering the research questions formulated in Chapter 4. The study, which follows the principles of an *Engineering research*, was designed to evaluate the developed tool, Infragenie, following the *Technology Acceptance Model* by accessing: 1) **acceptance**, 2) **ease of use**, 3) **usefulness** and 4) **intention to use**.

The study was realized using an online questionnaire shared in the scope of repository contribution made by the application. The contribution happened as a pull request in selected projects from GitHub. GitHub was used to reach a diverse set of people working with software development. From the demographic and background analysis, it was possible to verify that the respondents indeed showed diverse backgrounds, coming from 24 different countries, with professional experiences between 1 and more than 15 years of experience and coming from different areas of study and with different academic degrees.

We have also evaluated the UML knowledge of the participants as well as their experience with tools like plantUML to detect biased answers. Given the fact that our tool follows the UML standard, it is important to be evaluated by people with good knowledge of UML. Our evaluation showed that most of the users were highly familiar with the standard, given informed answers especially when evaluating the validity and understandability of the diagrams. The analysis of the experience in text-based tools was helpful to show evidence that a high proficiency in those tools was not necessary to use Infragenie.

Having assessed the participant's knowledge and collected information on the mentioned metrics, we can try to answer this dissertation's research questions:

RQ1 *"To what extent does the adoption of the developed tool lead to better and updated documentation?"* - To find an answer to this question, we have measured how users evaluate the diagrams provided by the tool and how these compare with other diagrams. More specifically, we asked questions to assess how users evaluated the completeness and ease of understanding the diagrams to get insights on whether the diagrams provided better documentation. We have also asked how easy it was to maintain and update the diagrams with Infragenie to conclude if the tool could lead to updated documentation. The results from these questions can be seen in Figure 6.11 and show that more than 60% of the respondents agreed or strongly agreed that Infragenie produced complete and easier to understand diagrams, which were easier to maintain and update. The question about ease of use, which answers are represented in Figure 6.8, also revealed how easy it is to generate diagrams and keep them updated and consistent with the source code. In the open question about comparing the diagrams generated by Infragenie and other diagrams (cf. Table 6.5) there were users clearly saying that Infragenie's diagram was better, showing more details, being UML compliant, and even detecting problems in the infrastructure definition. Furthermore, almost 80% of the total respondents that had diagrams before using Infragenie found that Infragenie's diagram was better or much better (cf. Figure 6.18). The comments about Infragenie at the end of the questionnaire (cf. Table 6.6) were also valuable in revealing interest in a systematic action to update the documentation. In sum, we have observed that a tool like Infragenie can recover Infrastructure diagrams with good consistency and precision without much effort, which are UML compliant, rendering these diagrams useful and most likely better than others created manually. Also, adopting Infragenie can help keep the diagrams updated, and this action can become a common practice during project development.

RQ2 *"To what extend does the adoption of Infragenie improve the knowledge about the system architecture?"* - Looking at the results aggregated in Figure 6.13, which represent the answers to the usefulness of the diagram, it is possible to see that this figure is mostly green, showing that overall the users of Infragenie consider that the diagram generated shows a complete view of the architecture and the most relevant elements. Furthermore, possible inconsistencies or missing information could be added to the diagram using the edit capabilities, a process found easy to accomplish by more than 70% of the respondents as depicted by question Q4 and Q5 from Figure 6.8 and in Figure 6.14 where it is accessed the usefullness of the edit capabilities. Having such a diagram, we can say that it is a valuable reference to describe the architecture of a project, consequently providing good knowledge about it.

RQ3 *"What is the value and willingness to use Infragenie?"* - For this question, we have looked into the acceptance of the tool by quantifying how many projects were effectively using the diagrams. The percentage of projects using the tool was 17% which seems low, but we cannot disregard the fact that 52% of the contributions introduced by Infragenie remained unreviewed. Several reasons could have happened for that, but one that we were able to verify was due to restrictions, like licence agreements, imposed by certain projects, to which Infragenie did not comply. We believe that the number of accepted contributions could be higher if those restrictions were followed, although they were out of the tool's scope. Looking at the 31% of rejected contributions, we should also analyze the reasons for that. As seen in Section 6.3.2, some projects were very simple, and participants found that a diagram such as this was unnecessary. This shows that a tool like Infragenie has more value if used in projects with rather complex architectures, where the automatic recovery can indeed save developers time and errors. Also, some project owners found the pull-requests created by Infragenie invasive and not the best way to showcase a new tool and did not even try to understand the tool. Neverthless, Figure 6.19 reveals that about 60% of the users intend to add the generated diagrams to their projects and even to future projects, which shows a positive acceptance of the tool. So, we can conclude that there is a willingness to use an application with the attributes of Infragenie.

Although all the questions had positive answers, we should not disregard the fact that many users pointed out that the tool needs improvements. This was expected because Infragenie was built as a minimum viable product to evaluate the hypothesis that auto-generated documentation could help motivate its creation and usage, as stated in Section 4.2. Even though it was possible to verify this hypothesis throughout the analysis. We were able to observe that there is a willingness to use Infragenie to generate diagrams, both in old and new projects. The diagrams generated produce valuable information about the project, increasing the project knowledge and fomenting the usage of the application to keep the diagrams updated.

As a final note, we can say that future iterations of this work could reveal a higher adoption if the editor and the flow of generating diagrams are improved.

Chapter 7

Conclusions and Future Work

This chapter summarizes what was discussed and found in this thesis and reflects on its contributions. In the end we, explore possible changes to the tool and study to propose future iterations of this work.

7.1 Summary

This work started with a literature review (Chapter 3) conducted to answer three questions:

1. *Q1: What approaches are there to generate a model of a software system from the contents of its repository?*

2. *Q2: What approaches are there for automatically documenting the architecture of a software system?*

3. *Q3: What tools are there to provide a visual representation of the architecture based on orchestration artefacts?*

This review overviews common problems and challenges of software architecture documentation. Analyses software and techniques for architecture recovery and evaluates available tools to visualise infrastructures. As a result, we were able to identify problems in the analysed methodologies to propose then a tool that could overcome those weaknesses and led to the formulation of a hypothesis (Chapter 4):

> “Having architecture models living alongside the software can reduce the documentation burden while bringing more feedback about the changes introduced. Creating and using documentation can become a consistent practice of the development process, indirectly helping and motivating architectural changes.”

Manual documentation creation is often seen as a burden for the development teams and is often disregarded, becoming outdated and obsolete. Better ways to deal with documentation exist in the form of automated tools. However, the existing tools are not fitted for most projects and, most of the time, are challenging to use.

This motivates us to propose a tool (Chapter 5), focused on the generation of architectural diagrams, named Infragenie, that puts together a set of features that we found essential to increase its adoption and lead to living documentation. Besides ease of use and compatibility with a large set of projects, the most differentiator features are the possibility of adding manual changes and annotations that are not lost in between analysis. The tool is also UML compliant, something that is not easy to find in the available tools since they usually propose a proper notation.

Our tool was developed to target all the projects using docker-compose as the main infrastructure framework. Using a selection of open source projects defining their infrastructure with docker-compose, we conducted a study (Chapter 6) following the **Engineering research** methodology to evaluate the proposed tool. We have analysed the users' perception and adoption of the tool from four different perspectives: 1) **acceptance**, 2) **ease of use**, 3) **usefulness** and 4) **intention to use**. This analysis allowed us to answer the research questions formulated in Section 4.3.

RQ1 "To what extent does the adoption of the developed tool lead to better and updated documentation?" - Having auto-generated diagrams with complete information about the system increases its adoption, consequently driving its introduction in the project's documentation. The users also point out that in most of the projects, the developed tool was easy to use to generate and maintain updated diagrams.

RQ2 "To what extent does the adoption of the developed tool improve the knowledge about the system architecture?" - Most users agreed that the diagrams had an overall complete and precise representation of the infrastructure, showing the most relevant elements. Such a diagram was easy to consult and thus contributed to a better knowledge of the architecture.

RQ3 "What is the value and willing to use the application developed?" - Despite showing a good acceptance of the tool and its diagrams, the study also revealed that tools that auto-generate documentation are expected to have the minimum human intervention possible. The analysis showed some needed improvements to increase the adoption of the proposed tool. Applying those improvements, we believe that the willingness to use would significantly increase.

7.2 Main contributions

The main contributions of this thesis are:

- **State-of-the-art review**: We explored and evaluated techniques and tools for auto-generating documentation artefacts, highlighting their strengths and weaknesses.

- **Approach** - We proposed a set of features that can increase the adoption of tools to auto-generate documentation, thus leading to better and updated documentation. We have also proposed a methodology to develop a tool with those characteristics and how to evaluate it.

- **Infrastructure recovery tool: Infragenie** - The prototype developed, following the proposed approach, to solve the problems of integrating tools to generate documentation in the development practice.

- **Engineering research** - A study conducted with a diverse set of individuals, without any human intervention, to avoid biasing the results and to evaluate the developed tool and its utility. This study helped confirm that the right set of features, precision and low effort for using would increase the willingness to adopt tools to generate documentation artefacts and thus keep the documentation updated.

7.3 Future Work

To expand this work with the objective to find a tool that can be largely accepted and help on the adoption of documentation practices, some aspects that can be reviewed and improved regarding the tool and the empirical study are proposed next.

Tool improvements

- **Increased compatibility** - A tool that generates documentation artefacts in an automated way is expected to run with minimal manual intervention while producing complete, precise and consistent elements. In the concrete case of this work, the tool is expected to provide the best possible visualization of the infrastructure. As stated throughout this document, the infrastructure of a project can be defined by multiple artefacts included in the source code. Furthermore, those artefacts can be associated with different frameworks or templates. The developed tool was only looking at docker-compose artefacts, although we believe that having a tool with higher compatibility would allow the generation of richer diagrams, showing more details and reducing the need for manual changes. Leading to a more straightforward adoption of the tool.

- **Auto detect changes** - The improvement section of the questionnaire used in the study showed that the users agreed on the benefits of having a tool that can automatically understand the changes and update the diagrams. Furthermore, this functionality would greatly expand on the concept of having a tool that assists in the creation of updated documentation, leading to projects with living documentation. Also, achieving this functionality does not represent a considerable effort since the developed tool already exposes a public API, as described in Section 5.5. So, detecting changes can be as easy as adding some more endpoints to integrate with CI/CD tools like Jenkins, CircleCI or even GitHub automations, to name a few. This functionality would allow a more thorough investigation of the benefits of having auto-generated documentation artefacts that can evolve with the project. This addition could shed some light on the question: How willing are users to keep their documentation updated if the process to generate it is just an initial configuration of the project?

- **Improved editor** - The editor of Infragenie was somehow limited, built as a proof of the concepts leveraged to address the problems stated in Chapter 4. The editor was too tied with the elements defined in a docker-compose specification. Also, the editor was built around plantUML, and despite not being necessary experience with plantUML, some knowledge of this text-based tool would facilitate its usage. A tool that can be used in various projects, independent of the developers' experience or technologies used, would allow a more diverse study, potentially bringing more knowledge on the users' expectations around documentation. Can a tool assist in creating documentation even if used by users without experience? Could we improve the adoption of the tool to plan future changes and experiment with the architecture?

- **Visual editor** - Following on the previous point, a visual editor could provide more freedom to users to edit and expand the representation while also facilitating its usage. Most likely, the editor's acceptance would increase, although a visual editor imposes some challenges if the UML standard continues to be a desired feature. In a free visual editor, it is not easy to drive and ensure users follow the specification.

Empirical study

- **Improved selection of projects** - The selection of the projects to run the study was one of the factors that influenced the results. Both quantitatively and qualitatively. For example, the number of respondents could be higher if we excluded projects expecting specific procedures that Infragenie was not following. Also, in very simple projects is not easy to understand the benefits of a tool like Infragenie, which can negatively impact the assessment of the tool's value. In future replications of this study, we cannot rely only on the GitHub filters but rather apply external filters.

- **Increased duration** - To further investigate the benefits of the proposed tool, the study should occur across several development cycles. Allowing to obverse how the tool is keeping track of the infrastructure changes. This observance would provide reinforced evidence around the concept of living documentation. Also, a more extended period would increase the users' familiarity with the tool, leading to more informed answers and such, more fundamented conclusions.

- **Different diffusion methodology** - Creating contributions in the form of pull requests without any prior contact with the project owners, and sharing a questionnaire inside that pull request, was not received well by some users. For that reason, those users did not even try the tool and completely disregarded the questionnaire. In that way, having those contributions in the collected data generates noise in the results, especially while analysing the tool's acceptance. To increase the quality of the data to be analysed, the next iteration of this study should find a different diffusion method, for example, using software development forums to invite users to try the tool.

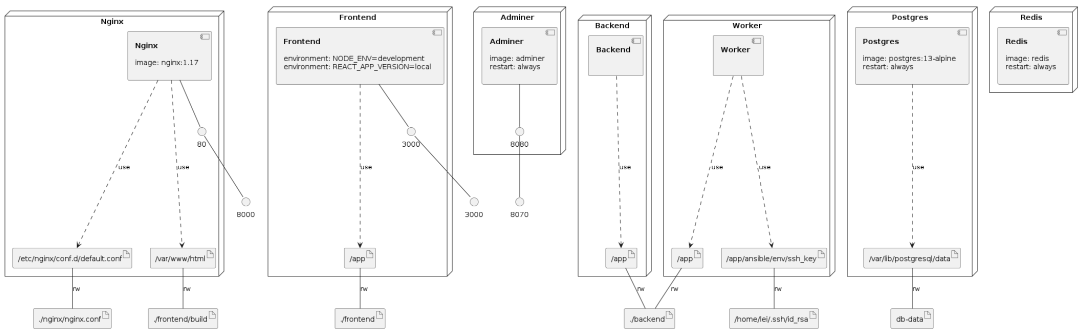
- **Controlled experiment with human guidance** - Our study was designed to target the most diverse set of developers and let them experiment with the tool on their own without any human intervention. It could be interesting to run a controlled experience with guidance and proper onboarding of the tool. Users with more information would give more value to the tool?

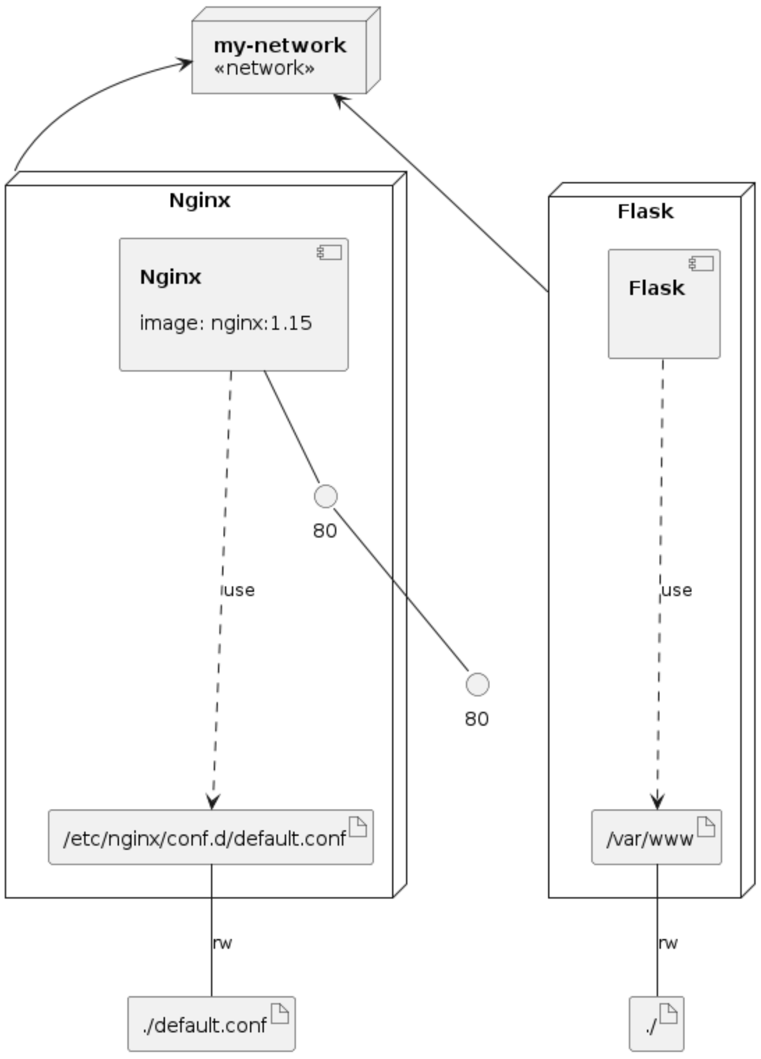
Appendix A

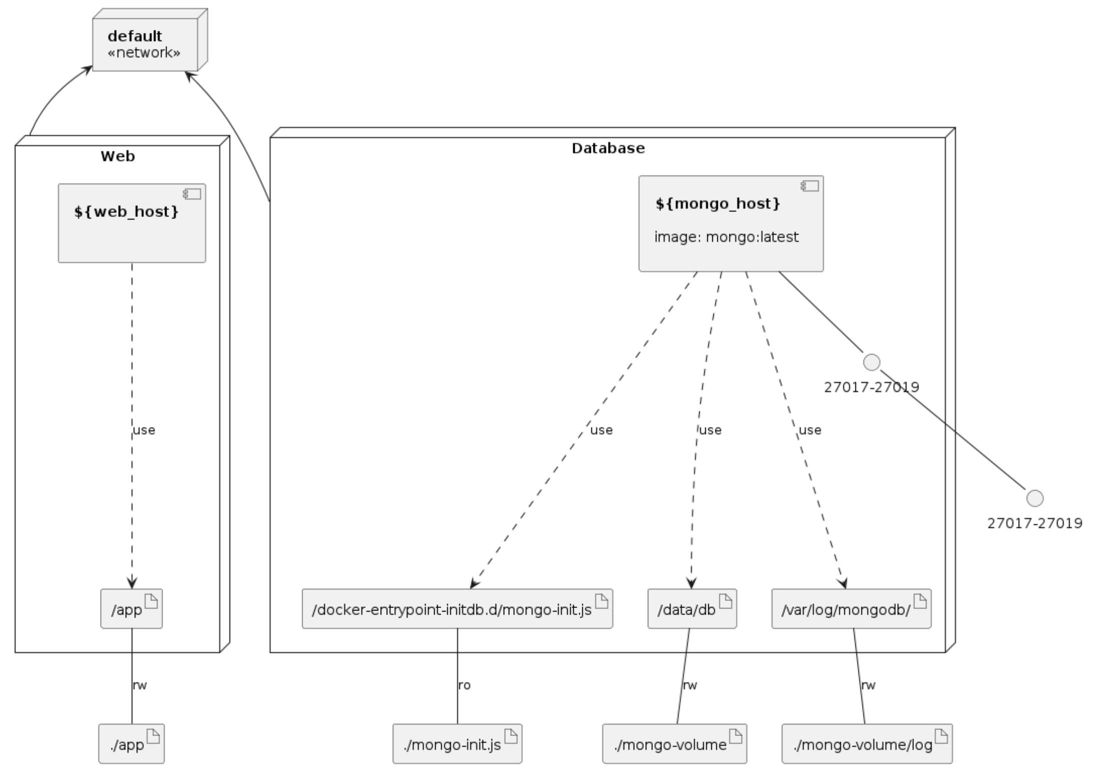
File samples generated by Infragenie

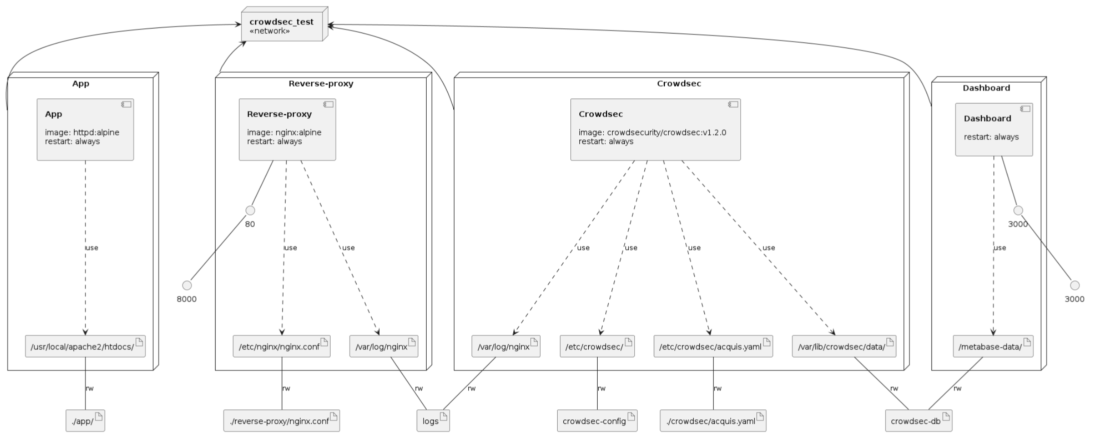
This appendix shows some examples of the diagrams produced by Infragenie.

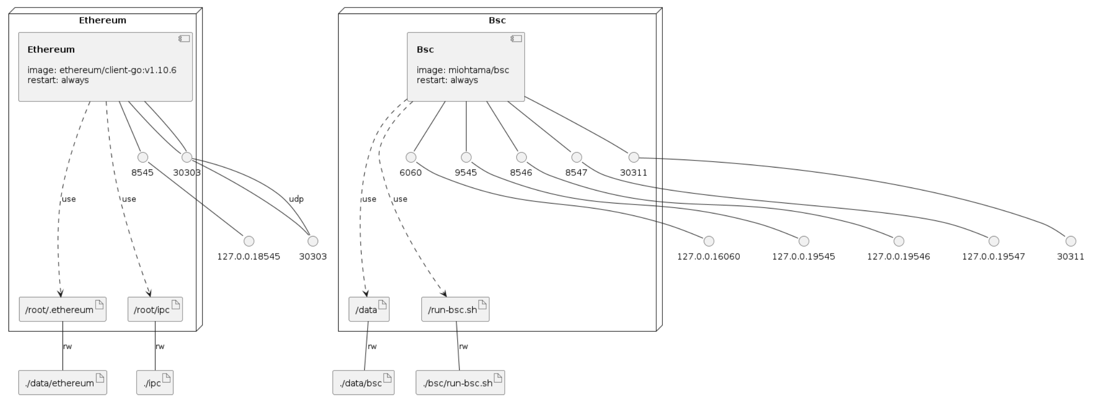
A.1 Infragenie diagrams examples

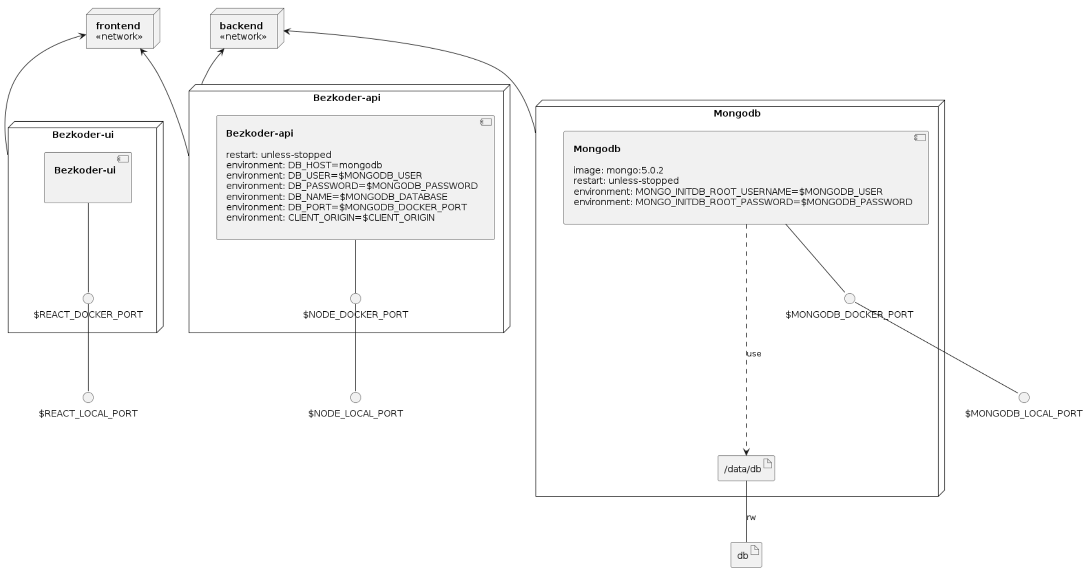



my-network
«network»
Nginx
Nginx
image: nginx:1.15
80
80
use
/etc/nginx/conf.d/default.conf
rw
./default.conf
Flask
Flask
use
/var/www
rw
./


default
«network»
Web
${web_host}
use
/app
rw
./app
Database
${mongo_host}
image: mongo:latest
use
use
use
27017-27019
27017-27019
/docker-entrypoint-initdb.d/mongo-init.js
/data/db
/var/log/mongodb/
ro
rw
rw
./mongo-init.js
./mongo-volume
./mongo-volume/log


crowdsec_test
«network»
App
App
image: httpd:alpine
restart: always
use
/usr/local/apache2/htdocs/
rw
./app/
Reverse-proxy
Reverse-proxy
image: nginx:alpine
restart: always
80
8000
use
use
/etc/nginx/nginx.conf
/var/log/nginx
rw
rw
./reverse-proxy/nginx.conf
logs
Crowdsec
Crowdsec
image: crowdsecurity/crowdsec:v1.2.0
restart: always
use
use
use
use
/var/log/nginx
/etc/crowdsec/
/etc/crowdsec/acquis.yaml
/var/lib/crowdsec/data/
rw
rw
rw
rw
crowdsec-config
./crowdsec/acquis.yaml
crowdsec-db
Dashboard
Dashboard
restart: always
3000
3000
use
/metabase-data/
rw


Ethereum
Ethereum
image: ethereum/client-go:v1.10.6
restart: always
8545
30303
use
use
udp
127.0.0.18545
30303
/root/.ethereum
/root/ipc
rw
rw
./data/ethereum
./ipc
Bsc
Bsc
image: miohtama/bsc
restart: always
6060
9545
8546
8547
30311
use
use
127.0.0.16060
127.0.0.19545
127.0.0.19546
127.0.0.19547
30311
/data
/run-bsc.sh
rw
rw
./data/bsc
./bsc/run-bsc.sh



```
restart: unless-stopped
environment: DB_HOST=mongodb
environment: DB_USER=$MONGODB_USER
environment: DB_PASSWORD=$MONGODB_PASSWORD
environment: DB_NAME=$MONGODB_DATABASE
environment: DB_PORT=$MONGODB_DOCKER_PORT
environment: CLIENT_ORIGIN=$CLIENT_ORIGIN
```

```
image: mongo:5.0.2
restart: unless-stopped
environment: MONGO_INITDB_ROOT_USERNAME=$MONGODB_USER
environment: MONGO_INITDB_ROOT_PASSWORD=$MONGODB_PASSWORD
```

Appendix B

Experimental Guide

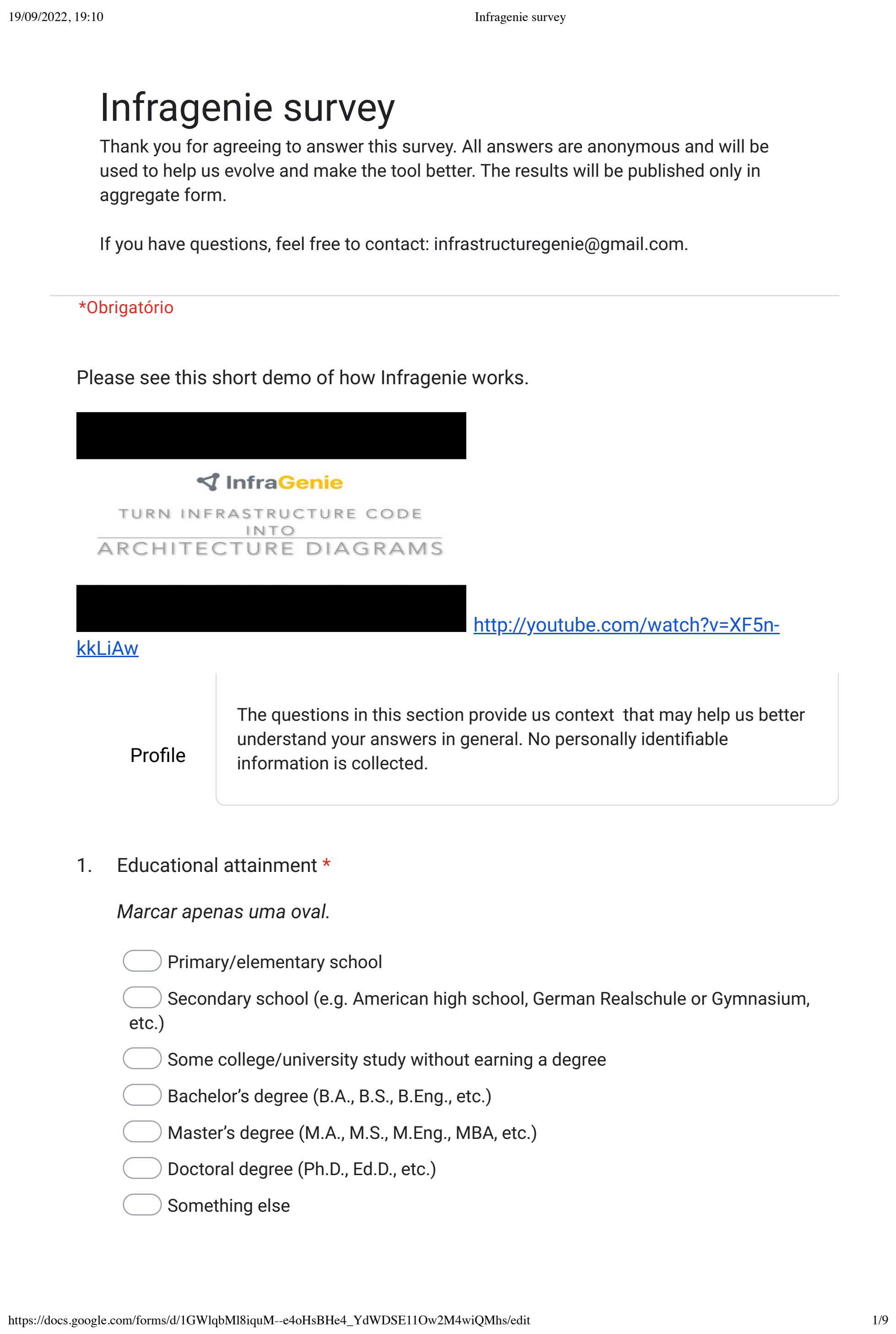
This appendix presents the survey used to conduct the study in Chapter 6 as well as some instructions about the replication package that can be used to replicate the study.

B.1 Replication package

The replication package that allows the replication of the study can be found in following repository: https://github.com/ricardojaferreira/infragenie-replication-package. The content of the repository has the following structure:

- **/data_analysis**: This folder contains jupyter notebooks to analyse the collected data. Namely, *participants_characterization.ipynb* used to analyse demographics and background of the respondents. *pull_requests_analysis.ipnb* to check the final state of the requests cerated during the study. *tool_analysis.ipnb* used to analyse the data collected to evaluate the tool
- **/collected_data**: This folder contains the raw data collected in csv format (raw_results.csv) and the list of pull requests created (pull_requests.csv), also in csv format.
- In the root of the project can be found a pdf version of the questionnaire (infragenie_survey.pdf) and the presentation video used to onboard the tool (infragenie-onboarding.mov).

B.2 Questionaire



19/09/2022, 19:10 Infragenie survey

Infragenie survey

Thank you for agreeing to answer this survey. All answers are anonymous and will be used to help us evolve and make the tool better. The results will be published only in aggregate form.

If you have questions, feel free to contact: infrastructuregenie@gmail.com.

*Obrigatório

Please see this short demo of how Infragenie works.

http://youtube.com/watch?v=XF5n-kkLiAw

Profile	The questions in this section provide us context that may help us better understand your answers in general. No personally identifiable information is collected.

1. Educational attainment *

Marcar apenas uma oval.

◯ Primary/elementary school

◯ Secondary school (e.g. American high school, German Realschule or Gymnasium, etc.)

◯ Some college/university study without earning a degree

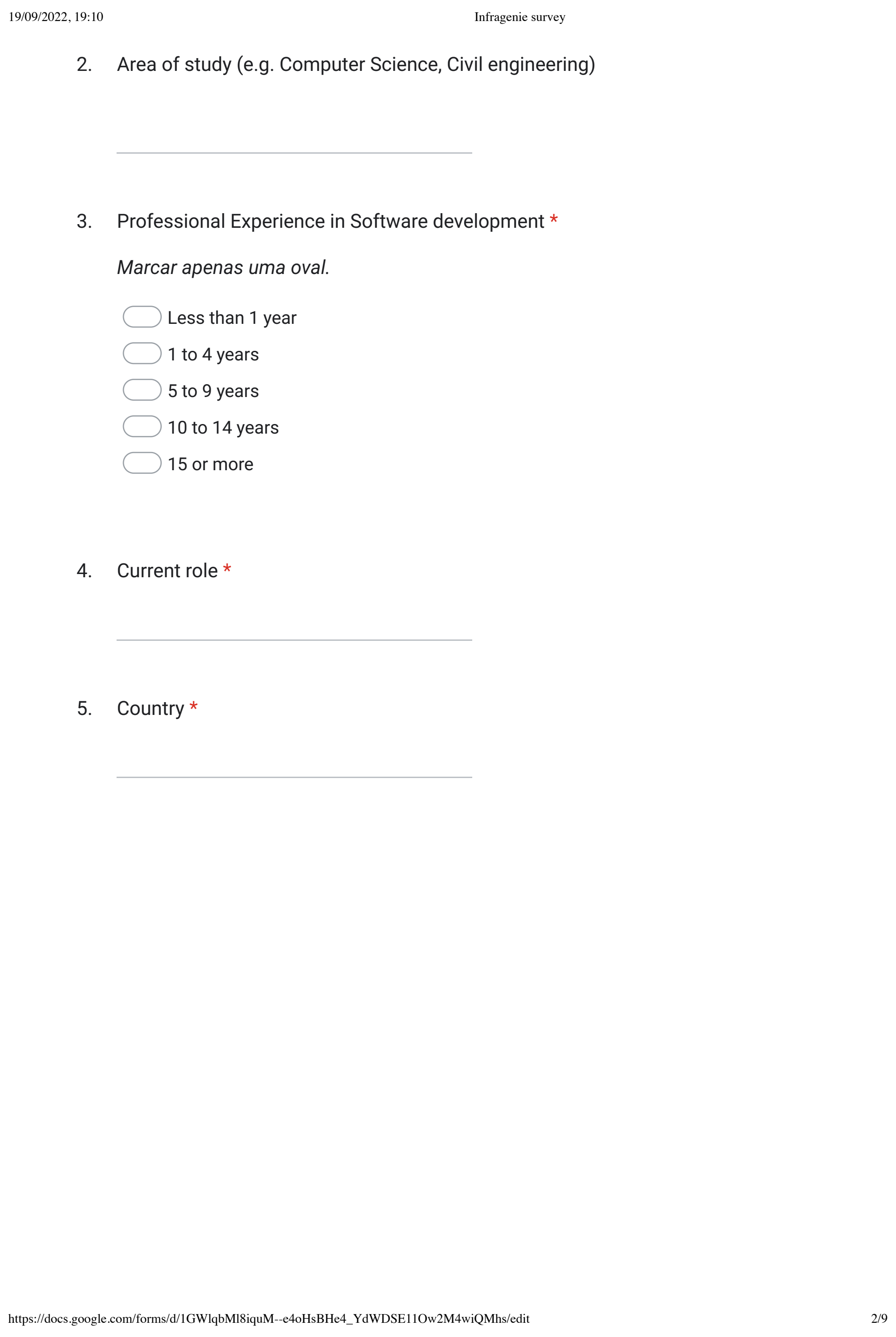
◯ Bachelor’s degree (B.A., B.S., B.Eng., etc.)

◯ Master’s degree (M.A., M.S., M.Eng., MBA, etc.)

◯ Doctoral degree (Ph.D., Ed.D., etc.)

◯ Something else

https://docs.google.com/forms/d/1GWlqbMl8iquM--e4oHsBHe4_YdWDSE11Ow2M4wiQMhs/edit 1/9



19/09/2022, 19:10 Infragenie survey

2. Area of study (e.g. Computer Science, Civil engineering)

____

3. Professional Experience in Software development *

Marcar apenas uma oval.

- ◯ Less than 1 year
- ◯ 1 to 4 years
- ◯ 5 to 9 years
- ◯ 10 to 14 years
- ◯ 15 or more

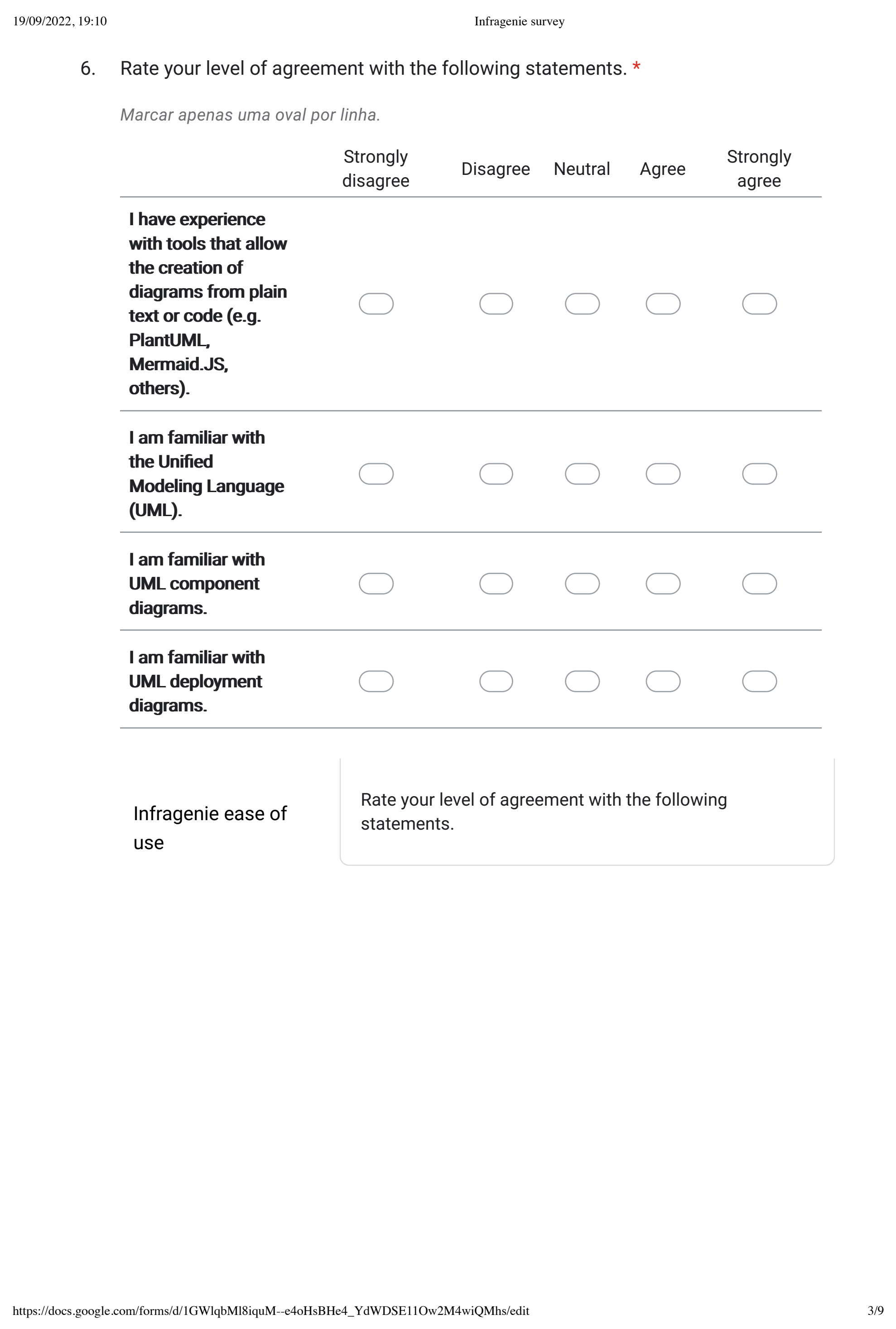
4. Current role *

____

5. Country *

____

https://docs.google.com/forms/d/1GWlqbMl8iquM--e4oHsBHe4_YdWDSE11Ow2M4wiQMhs/edit 2/9



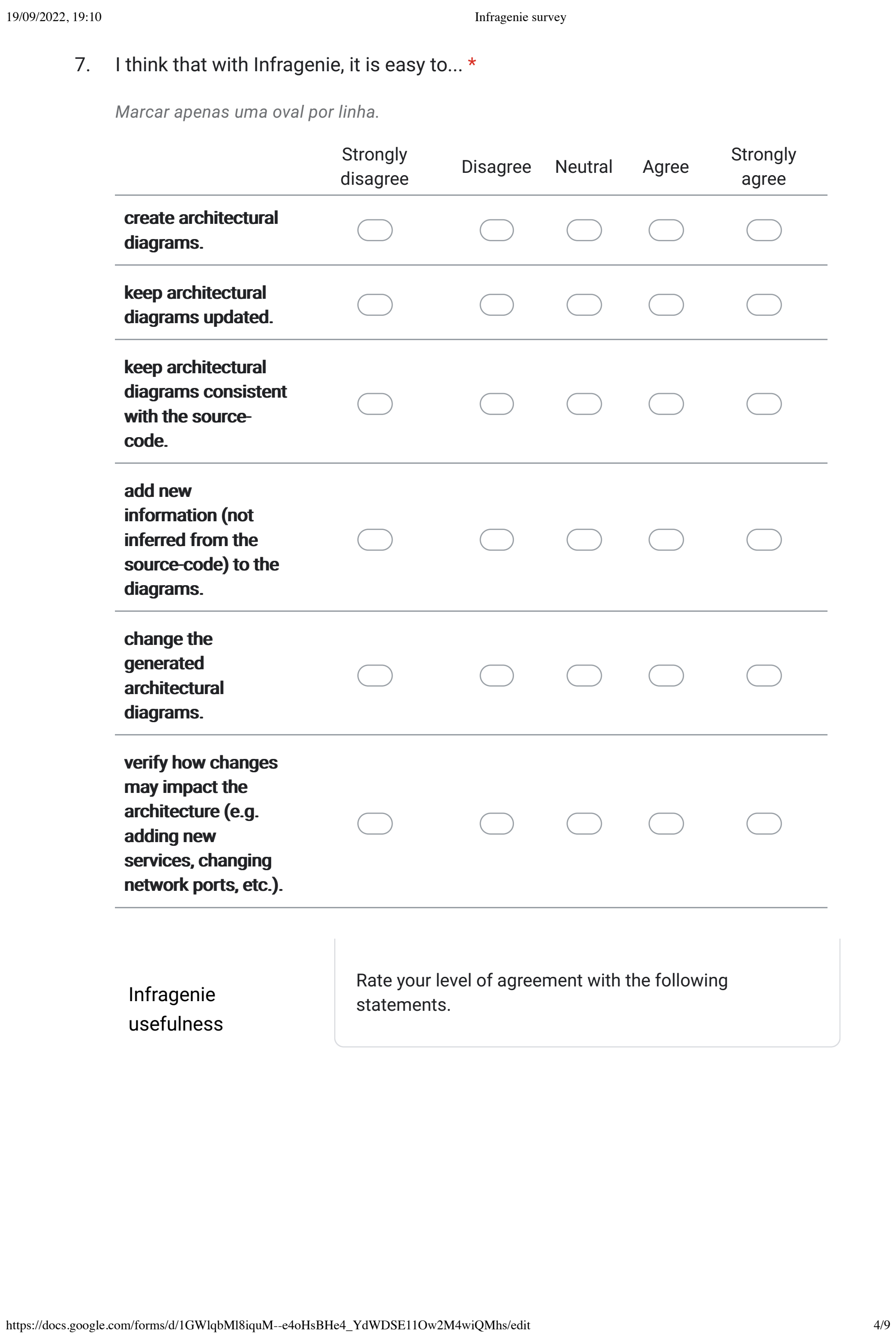
6. Rate your level of agreement with the following statements. *

Marcar apenas uma oval por linha.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
I have experience with tools that allow the creation of diagrams from plain text or code (e.g. PlantUML, Mermaid.JS, others).	◯	◯	◯	◯	◯
I am familiar with the Unified Modeling Language (UML).	◯	◯	◯	◯	◯
I am familiar with UML component diagrams.	◯	◯	◯	◯	◯
I am familiar with UML deployment diagrams.	◯	◯	◯	◯	◯

Infragenie ease of use

Rate your level of agreement with the following statements.



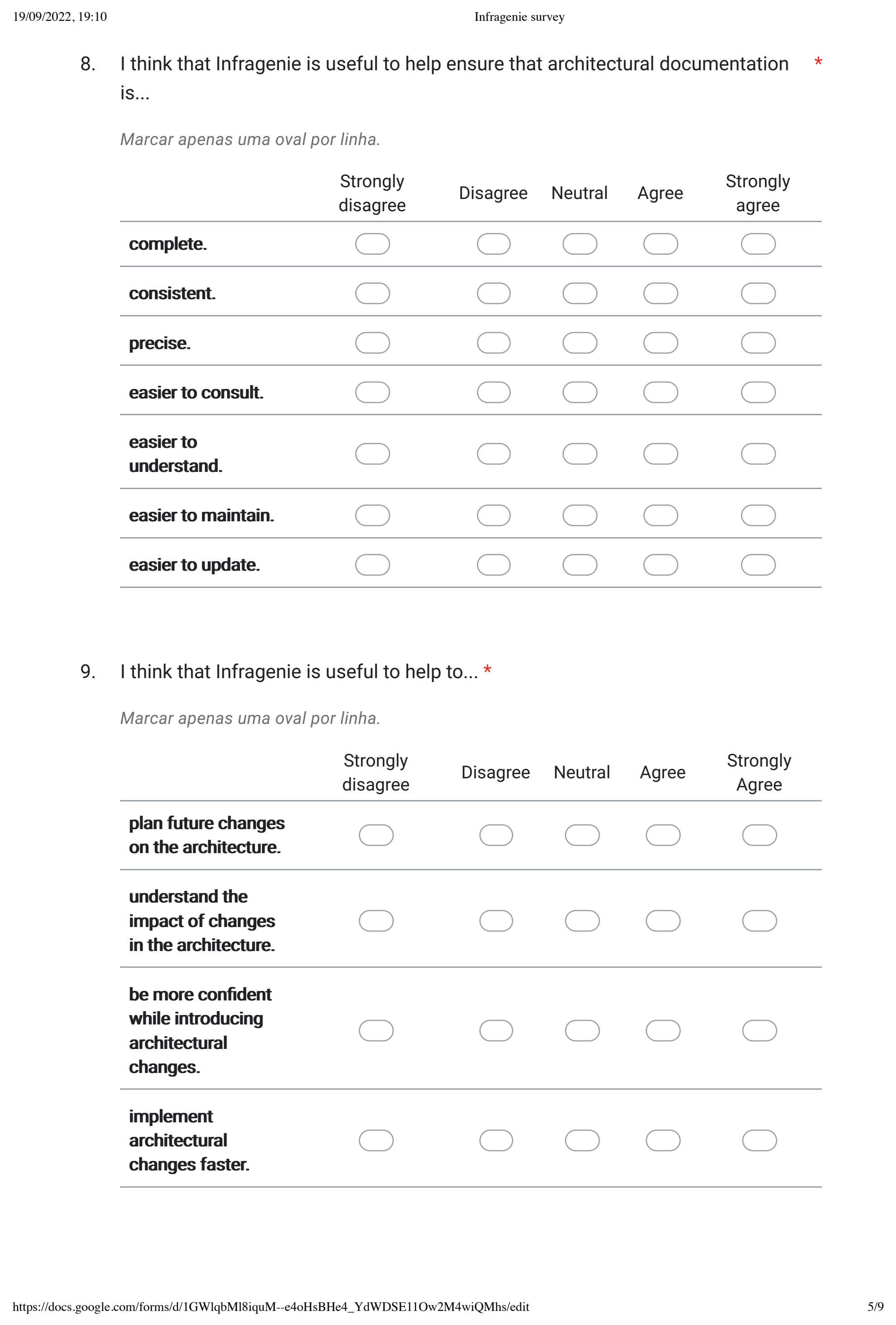
7. I think that with Infragenie, it is easy to... *

Marcar apenas uma oval por linha.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
create architectural diagrams.	◯	◯	◯	◯	◯
keep architectural diagrams updated.	◯	◯	◯	◯	◯
keep architectural diagrams consistent with the source-code.	◯	◯	◯	◯	◯
add new information (not inferred from the source-code) to the diagrams.	◯	◯	◯	◯	◯
change the generated architectural diagrams.	◯	◯	◯	◯	◯
verify how changes may impact the architecture (e.g. adding new services, changing network ports, etc.).	◯	◯	◯	◯	◯

Infragenie usefulness

Rate your level of agreement with the following statements.



19/09/2022, 19:10 Infragenie survey

8. I think that Infragenie is useful to help ensure that architectural documentation is... *

Marcar apenas uma oval por linha.

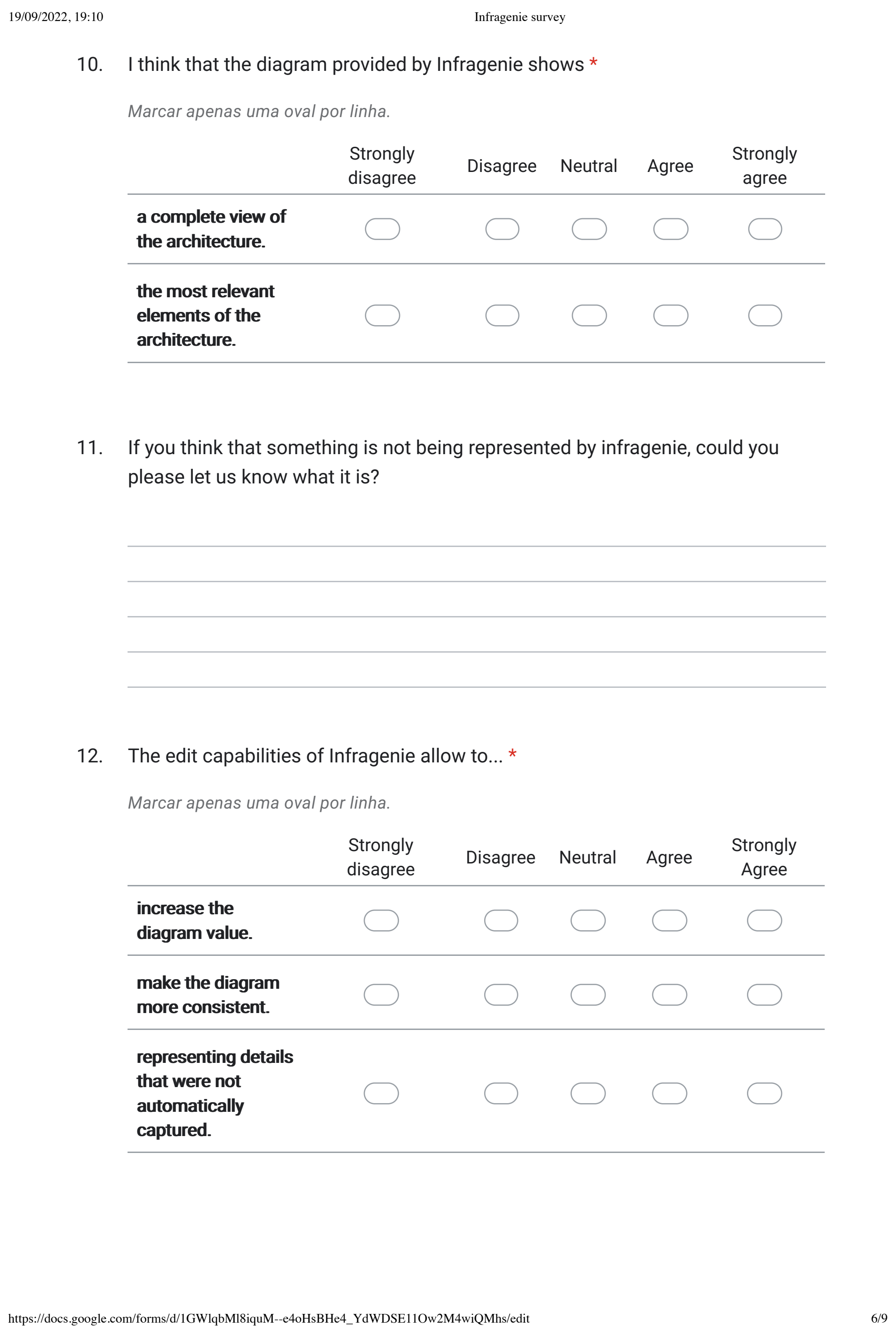
	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
complete.	○	○	○	○	○
consistent.	○	○	○	○	○
precise.	○	○	○	○	○
easier to consult.	○	○	○	○	○
easier to understand.	○	○	○	○	○
easier to maintain.	○	○	○	○	○
easier to update.	○	○	○	○	○

9. I think that Infragenie is useful to help to... *

Marcar apenas uma oval por linha.

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
plan future changes on the architecture.	○	○	○	○	○
understand the impact of changes in the architecture.	○	○	○	○	○
be more confident while introducing architectural changes.	○	○	○	○	○
implement architectural changes faster.	○	○	○	○	○

https://docs.google.com/forms/d/1GWlqbMl8iquM--e4oHsBHe4_YdWDSE11Ow2M4wiQMhs/edit 5/9



10. I think that the diagram provided by Infragenie shows *

Marcar apenas uma oval por linha.

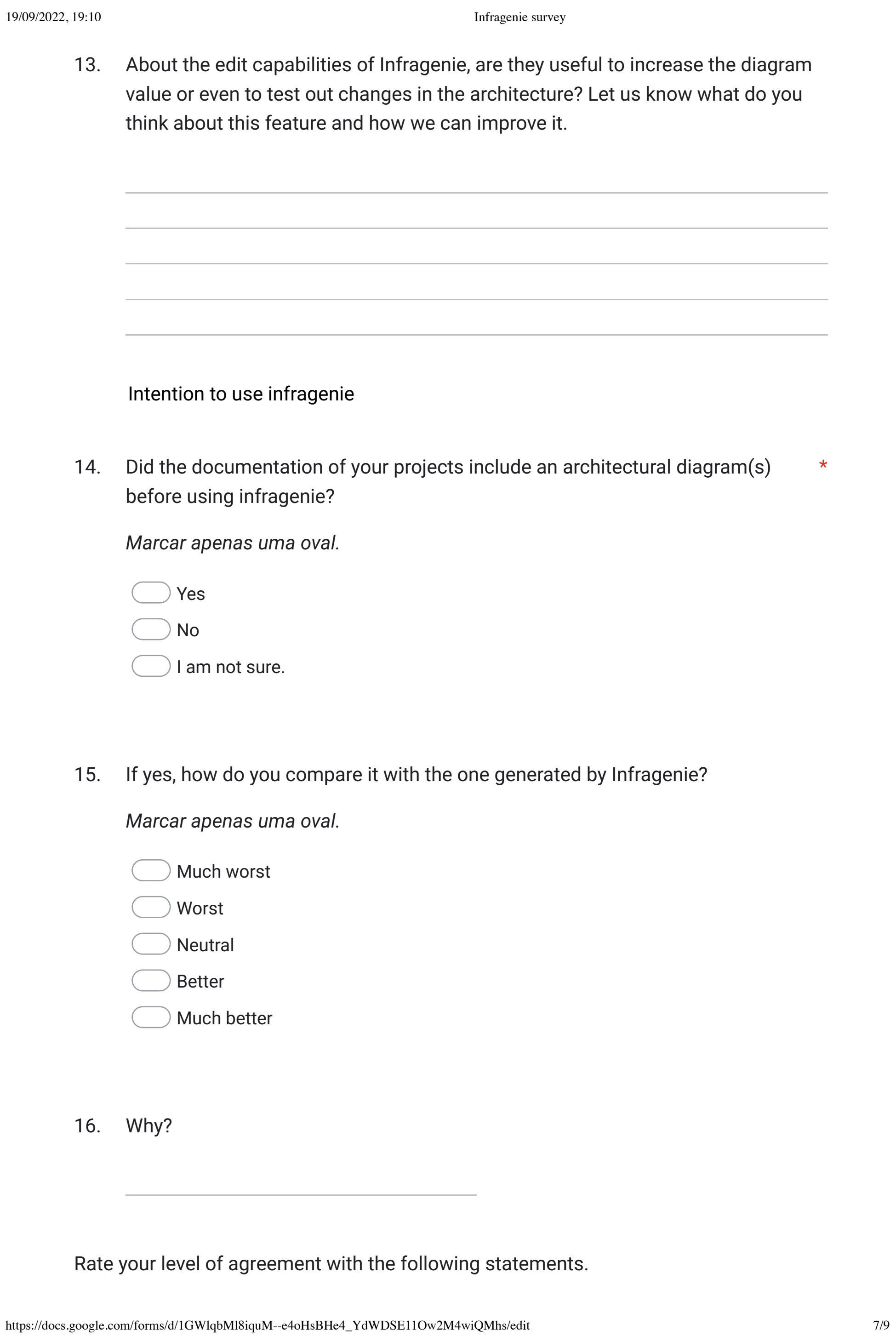
	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
a complete view of the architecture.	◯	◯	◯	◯	◯
the most relevant elements of the architecture.	◯	◯	◯	◯	◯

11. If you think that something is not being represented by infragenie, could you please let us know what it is?

12. The edit capabilities of Infragenie allow to... *

Marcar apenas uma oval por linha.

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
increase the diagram value.	◯	◯	◯	◯	◯
make the diagram more consistent.	◯	◯	◯	◯	◯
representing details that were not automatically captured.	◯	◯	◯	◯	◯



19/09/2022, 19:10 Infragenie survey

13. About the edit capabilities of Infragenie, are they useful to increase the diagram value or even to test out changes in the architecture? Let us know what do you think about this feature and how we can improve it.

Intention to use infragenie

14. Did the documentation of your projects include an architectural diagram(s) before using infragenie? *

Marcar apenas uma oval.

- Yes
- No
- I am not sure.

15. If yes, how do you compare it with the one generated by Infragenie?

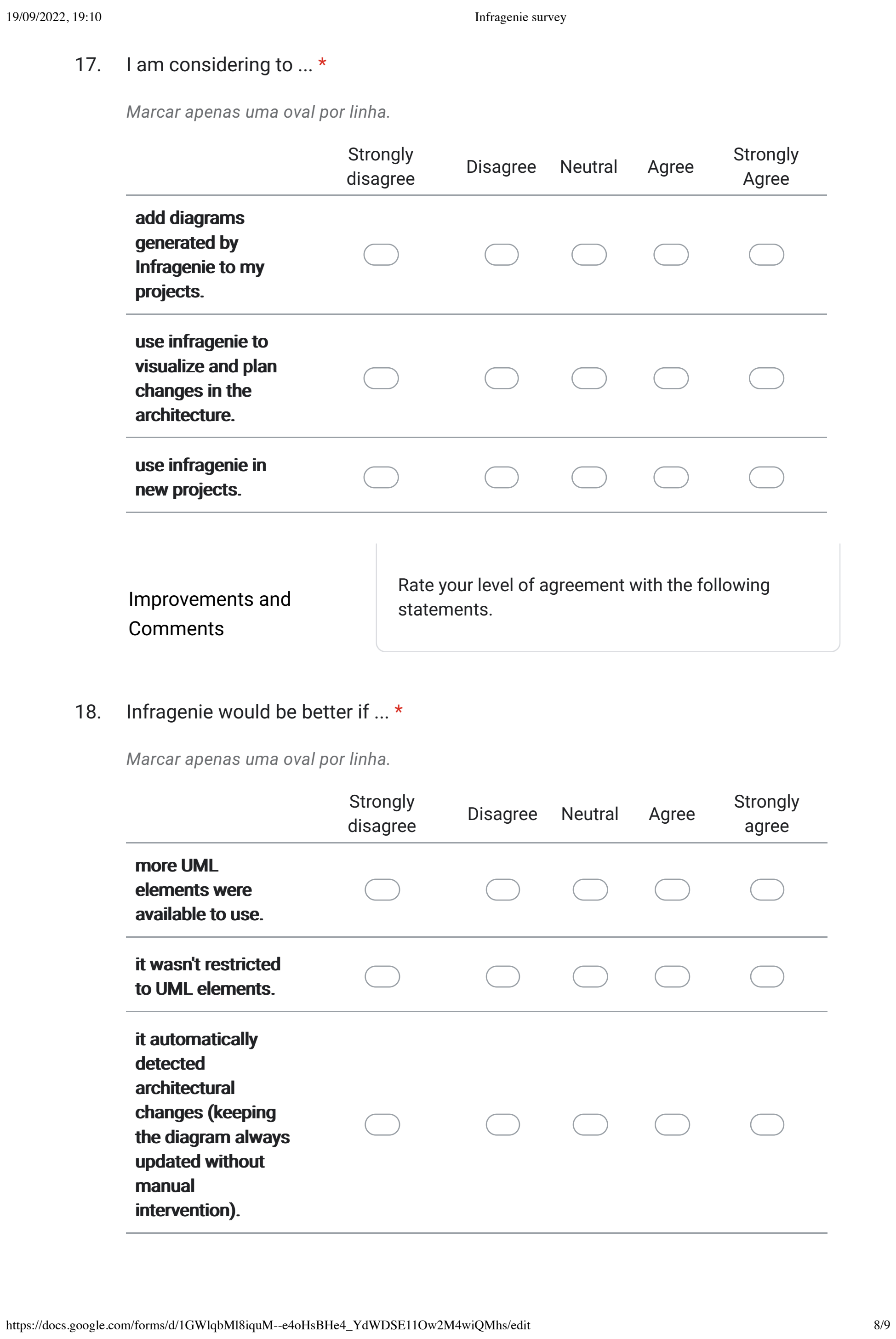
Marcar apenas uma oval.

- Much worst
- Worst
- Neutral
- Better
- Much better

16. Why?

Rate your level of agreement with the following statements.

https://docs.google.com/forms/d/1GWlqbMl8iquM--e4oHsBHe4_YdWDSE11Ow2M4wiQMhs/edit 7/9



19/09/2022, 19:10 Infragenie survey

17. I am considering to ... *

Marcar apenas uma oval por linha.

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
add diagrams generated by Infragenie to my projects.	◯	◯	◯	◯	◯
use infragenie to visualize and plan changes in the architecture.	◯	◯	◯	◯	◯
use infragenie in new projects.	◯	◯	◯	◯	◯

Improvements and Comments

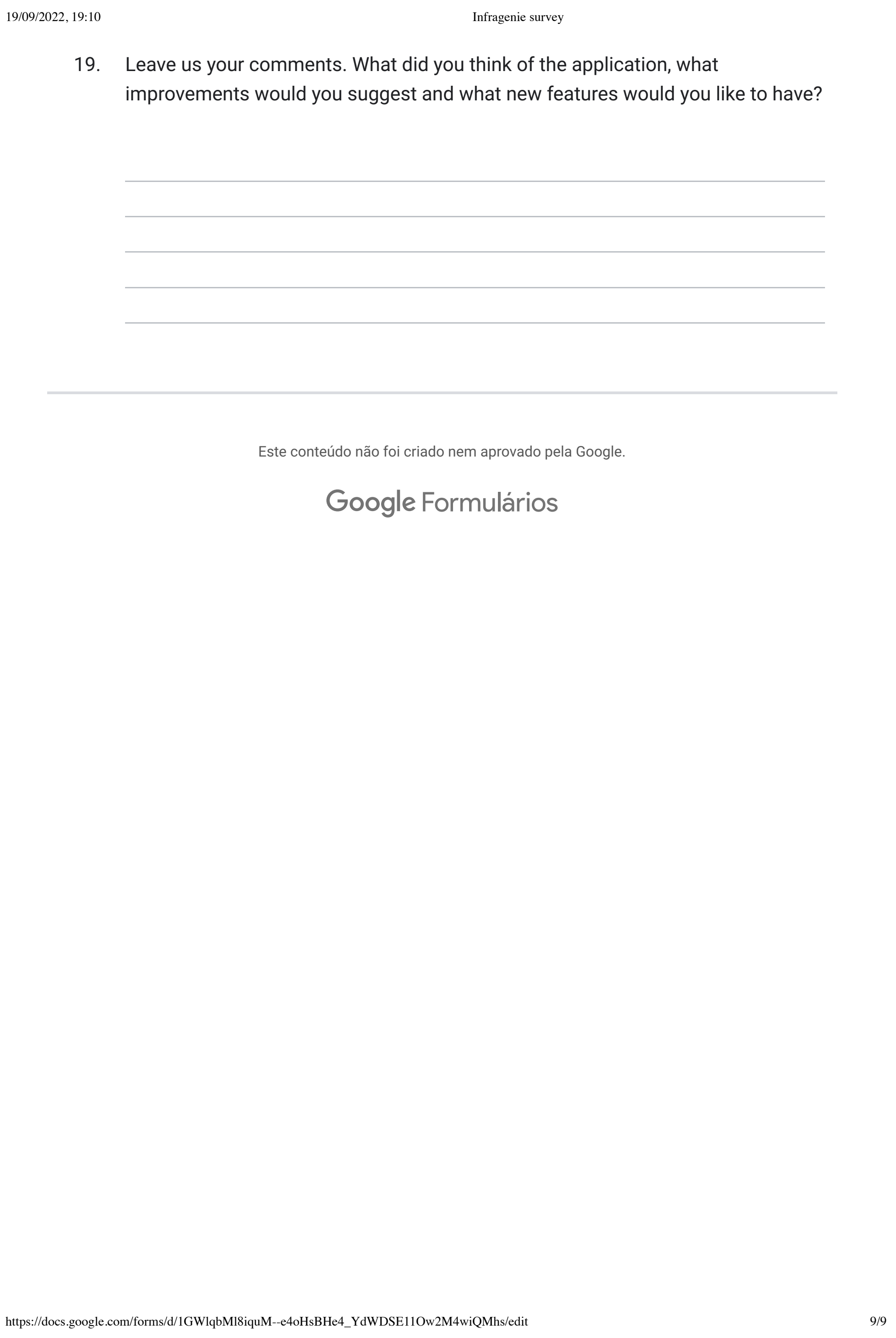
Rate your level of agreement with the following statements.

18. Infragenie would be better if ... *

Marcar apenas uma oval por linha.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
more UML elements were available to use.	◯	◯	◯	◯	◯
it wasn't restricted to UML elements.	◯	◯	◯	◯	◯
it automatically detected architectural changes (keeping the diagram always updated without manual intervention).	◯	◯	◯	◯	◯

https://docs.google.com/forms/d/1GWlqbMl8iquM--e4oHsBHe4_YdWDSE11Ow2M4wiQMhs/edit 8/9



19. Leave us your comments. What did you think of the application, what improvements would you suggest and what new features would you like to have?

Appendix C

Data crossing

This appendix aggregates all the graphs that represent relations between the data collected to evaluate the tool and the user knowledge about UML and plantUML.

C.1 Knowledge vs Ease of use

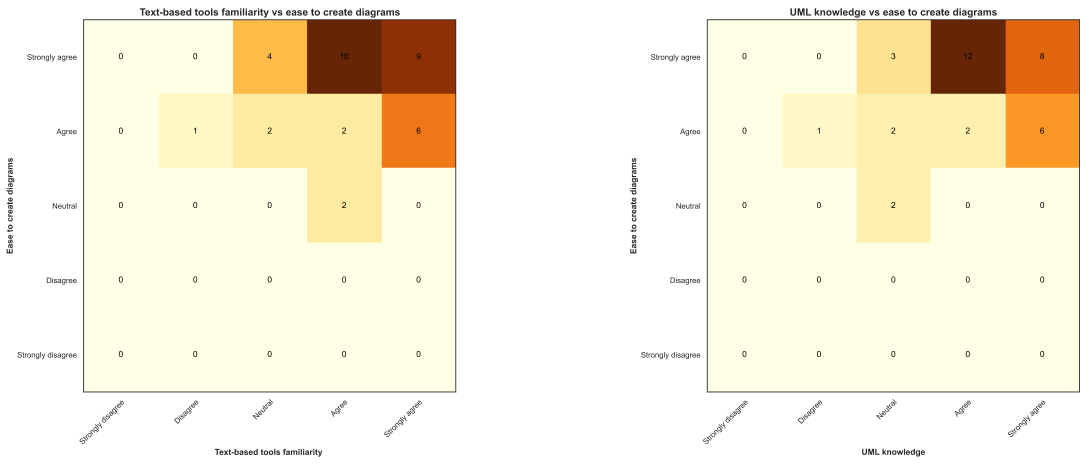


Figure C.1: Relation between user knowledge and ease to create diagrams.

C.2 Knowledge vs Usefullness

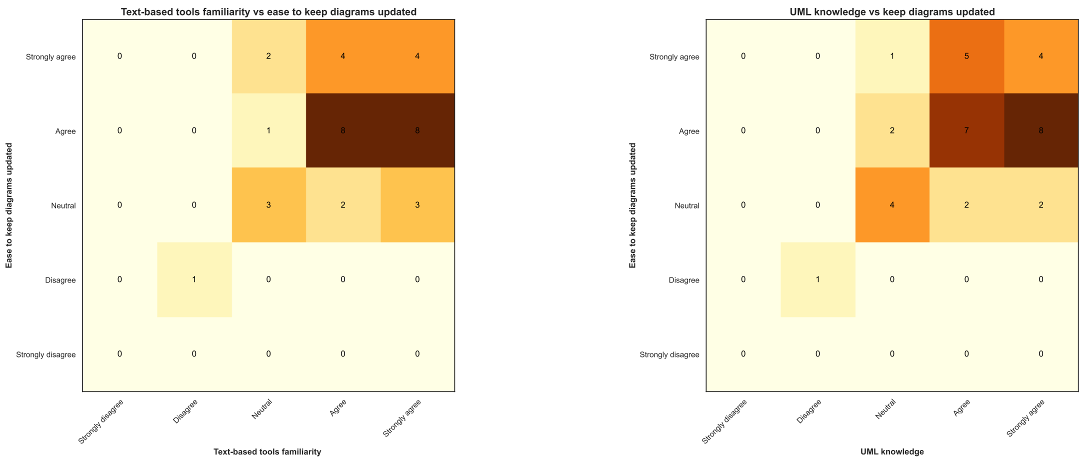


Figure C.2: Relation between user knowledge and ease to keep diagrams updated.

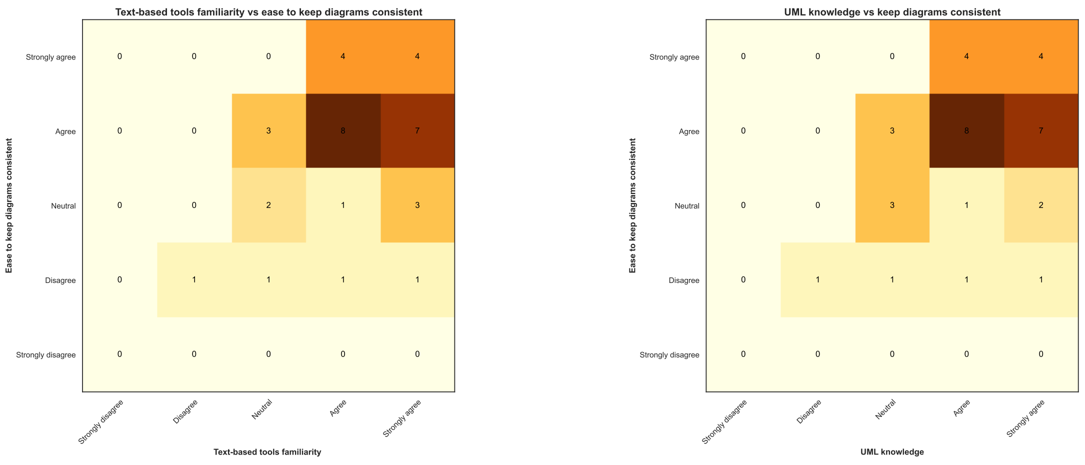


Figure C.3: Relation between user knowledge and ease to keep diagrams consistent.

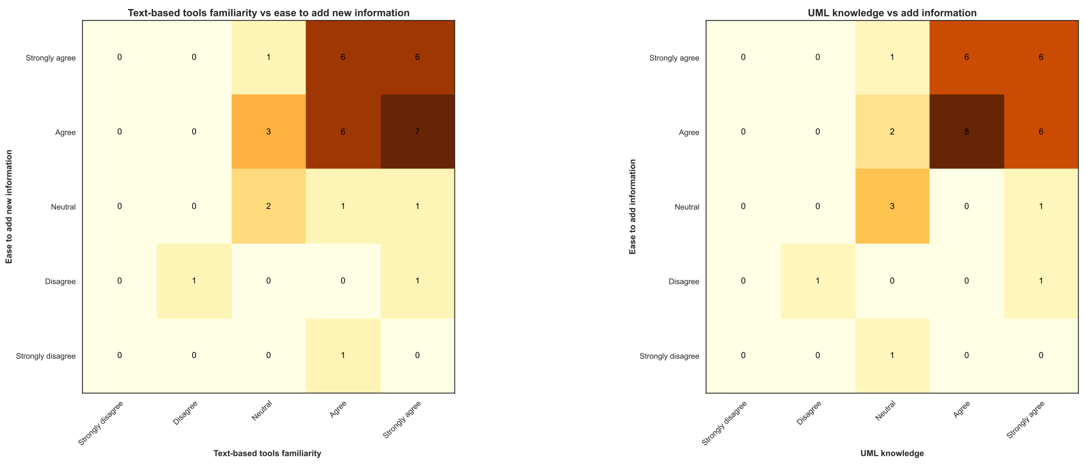


Figure C.4: Relation between user knowledge and ease to add information.

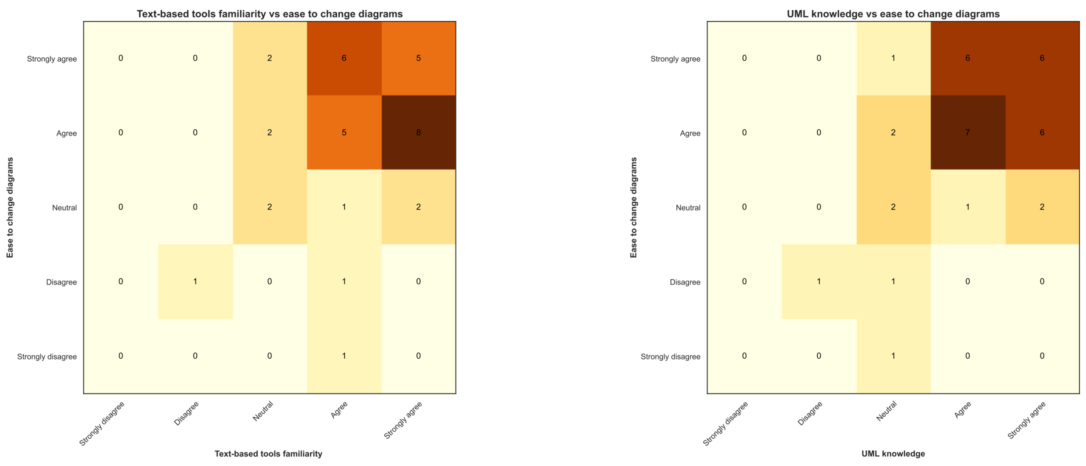


Figure C.5: Relation between user knowledge and ease to change diagrams.

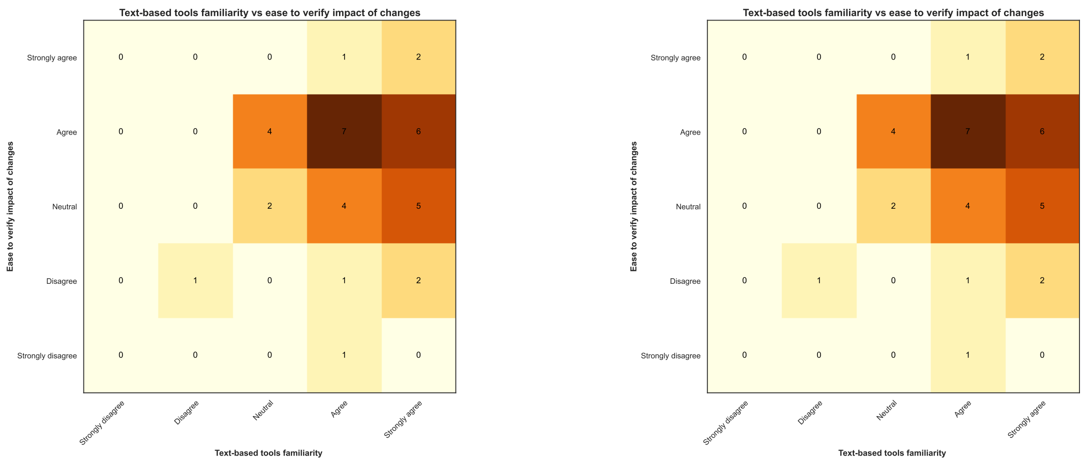


Figure C.6: Relation between user knowledge and ease to verify impact of changes.

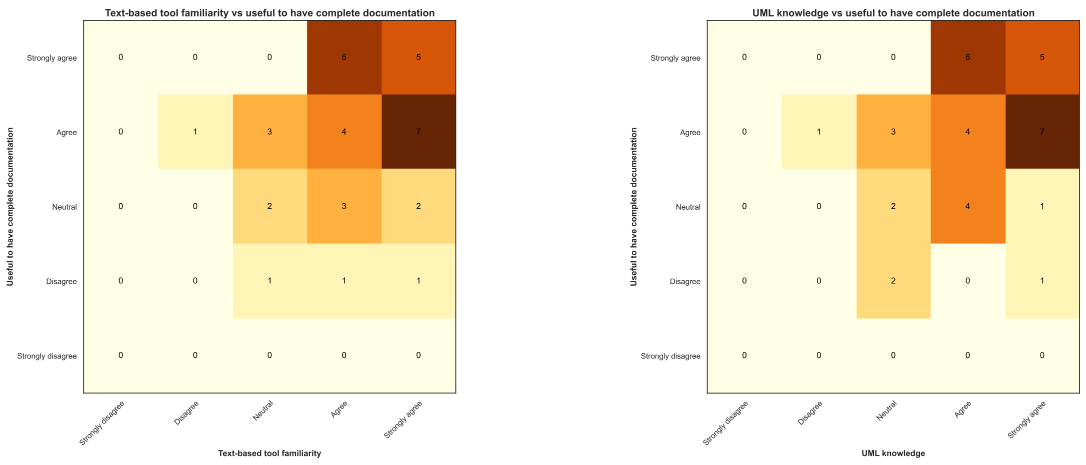


Figure C.7: Relation between user knowledge and usefullness to ensure complete documentation.

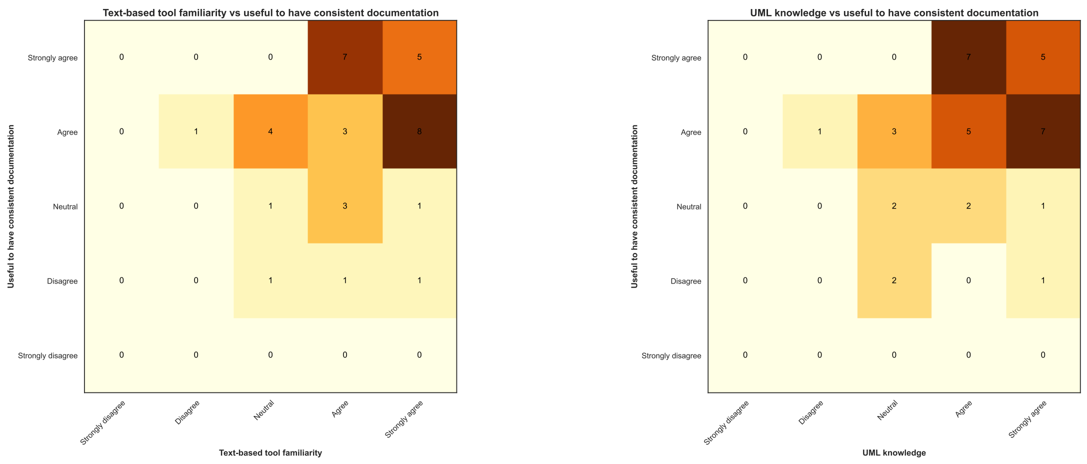


Figure C.8: Relation between user knowledge and usefullness to ensure consistent documentation.

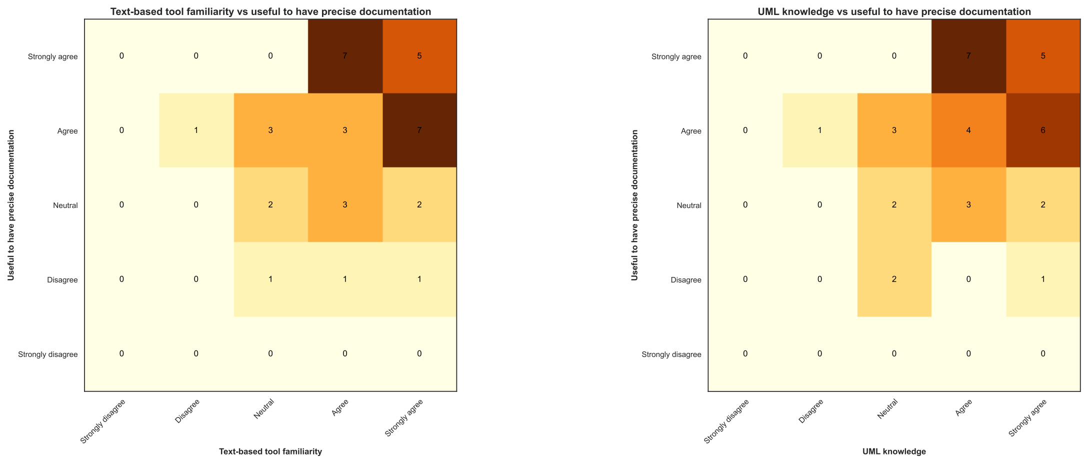


Figure C.9: Relation between user knowledge and usefullness to ensure precise documentation.

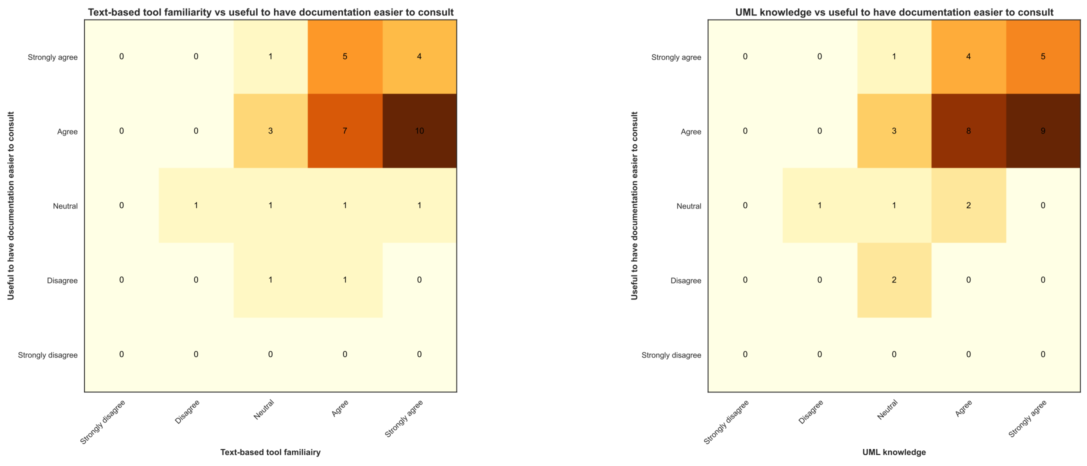


Figure C.10: Relation between user knowledge and usefullness to ensure documentation is easier to consult.

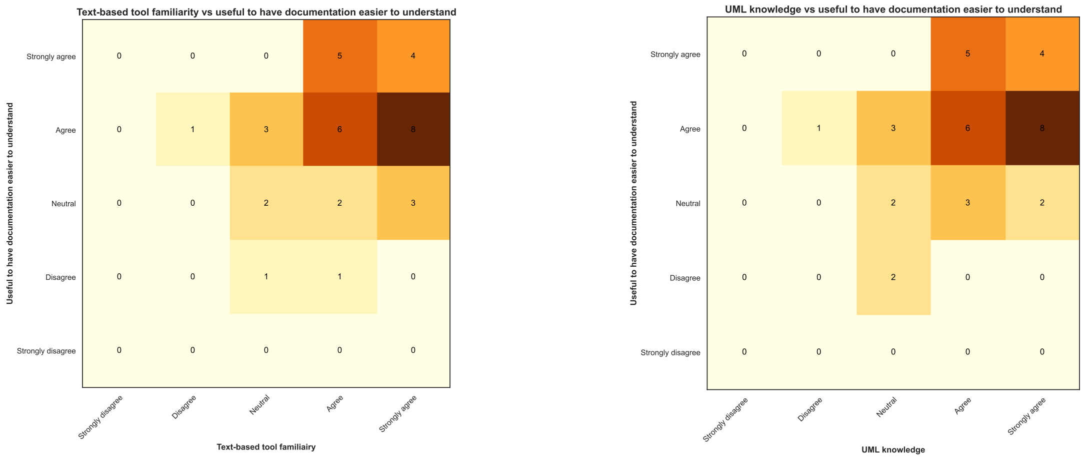


Figure C.11: Relation between user knowledge and usefullness to ensure documentation is easier to understand.

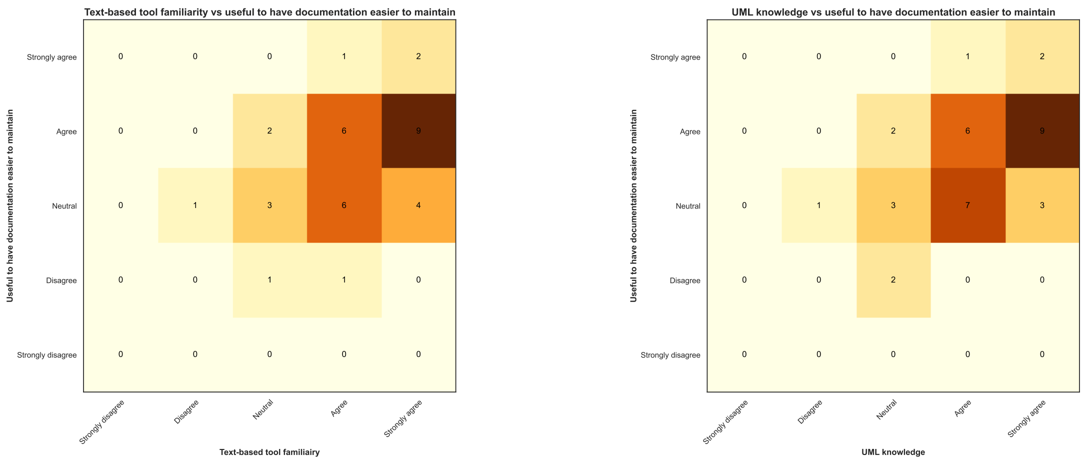


Figure C.12: Relation between user knowledge and usefullness to ensure documentation is easier to maintain.

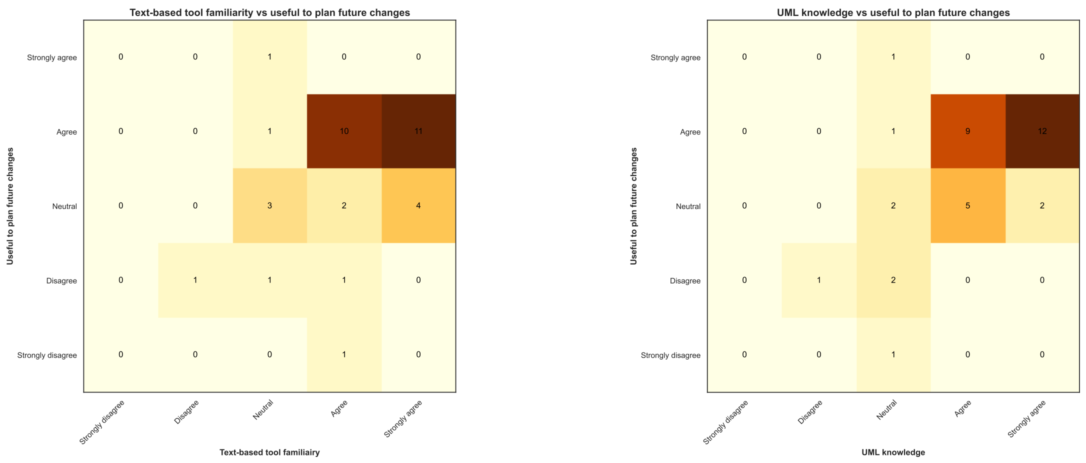


Figure C.13: Relation between user knowledge and usefullness to plan future changes.

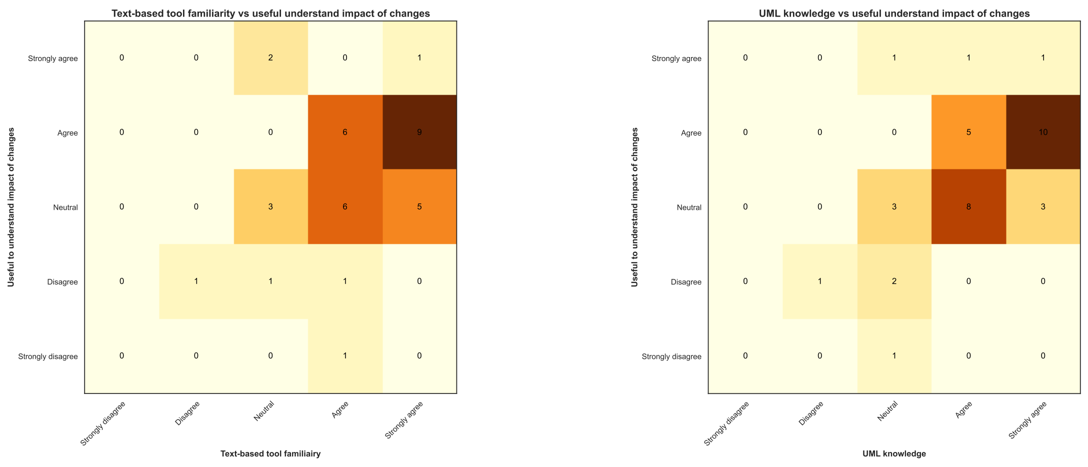


Figure C.14: Relation between user knowledge and usefullness to understand impact of changes.

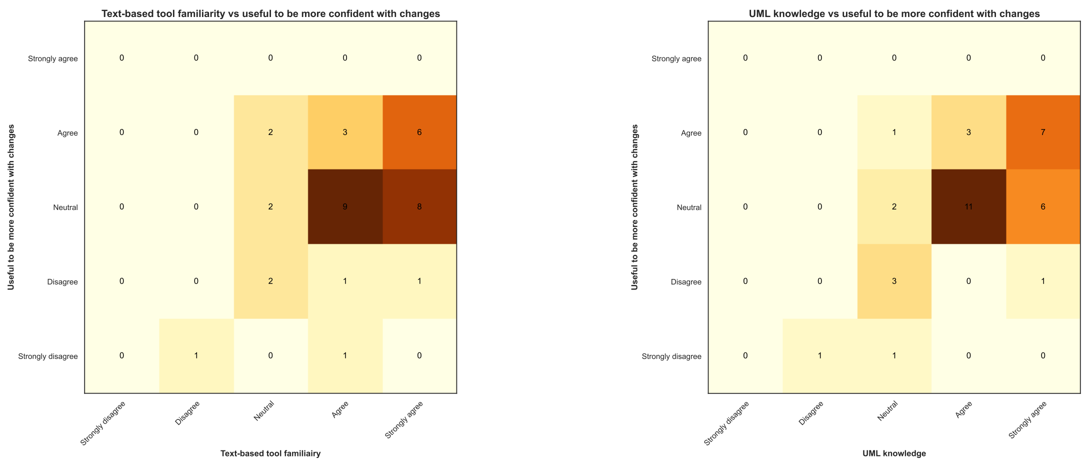


Figure C.15: Relation between user knowledge and usefullness to introduce changes with confidence.

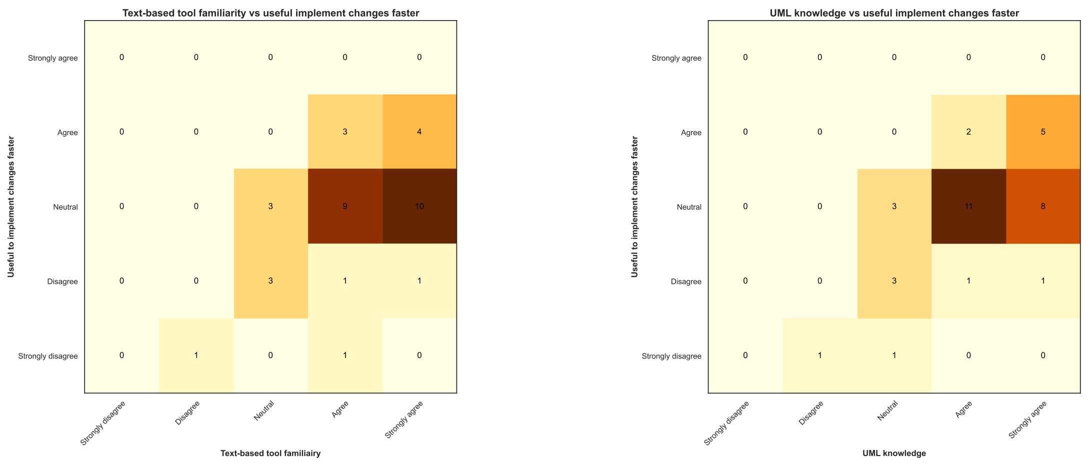


Figure C.16: Relation between user knowledge and usefullness to introduce changes faster.

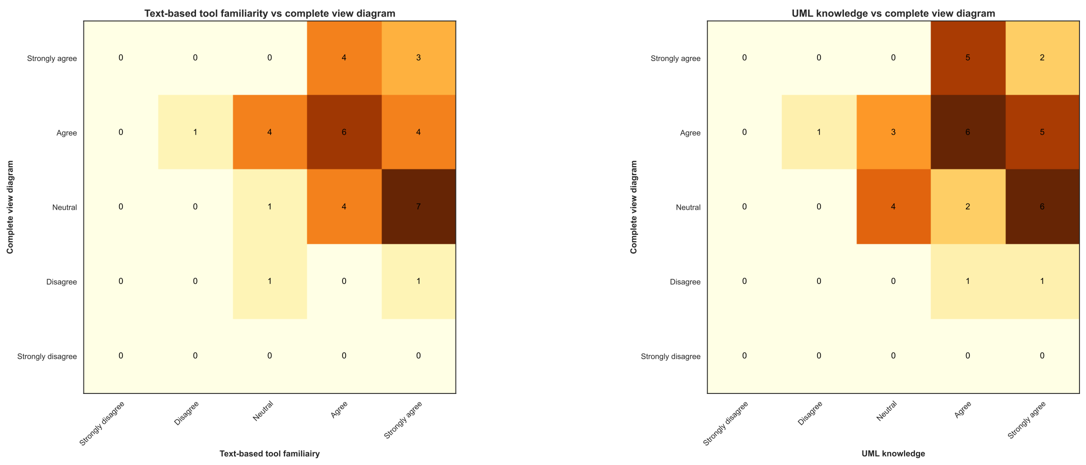


Figure C.17: Relation between user knowledge and showing a complete diagram.

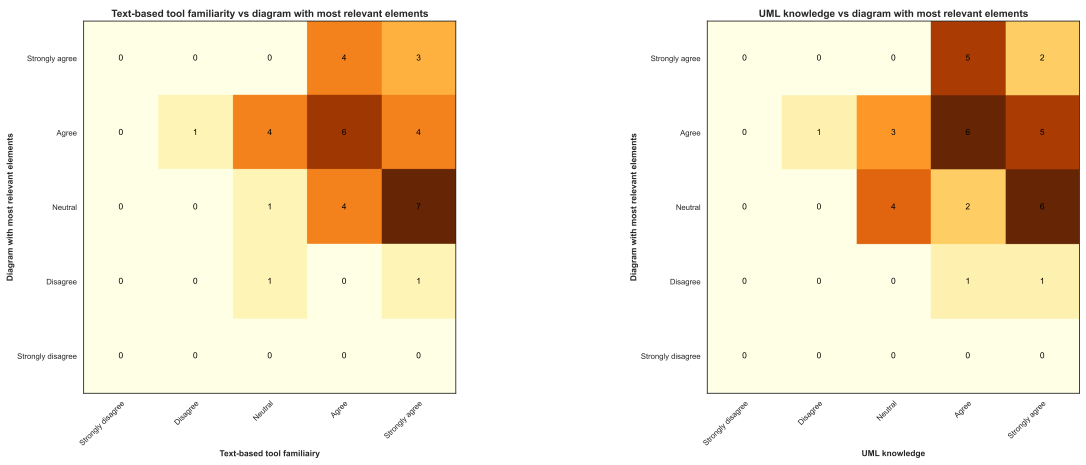


Figure C.18: Relation between user knowledge and showing a diagram with most relevant elements.

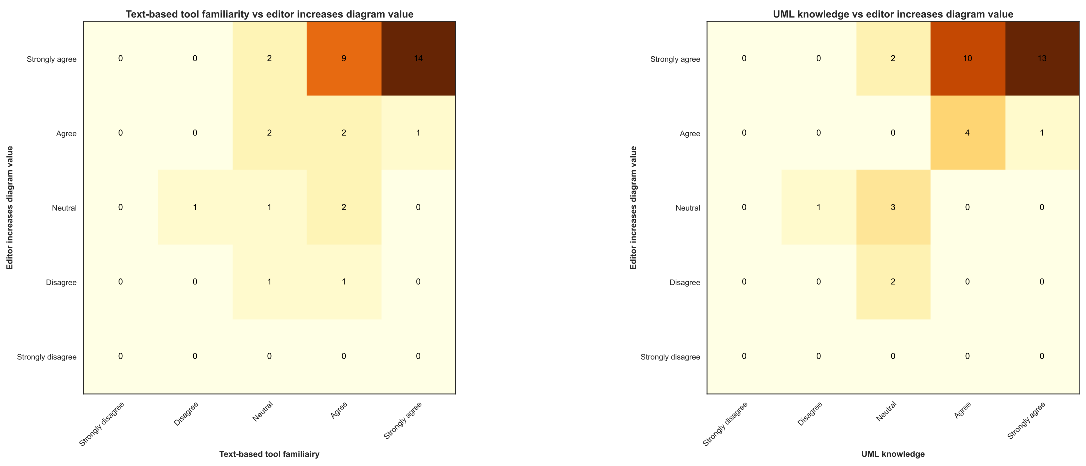


Figure C.19: Relation between user knowledge and editor increases diagram value.

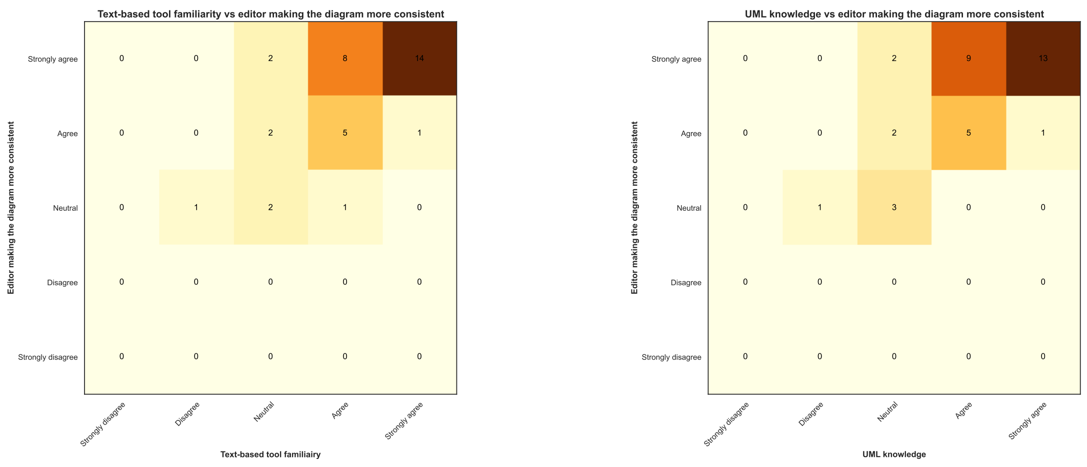


Figure C.20: Relation between user knowledge and editor making the diagram more consistent.

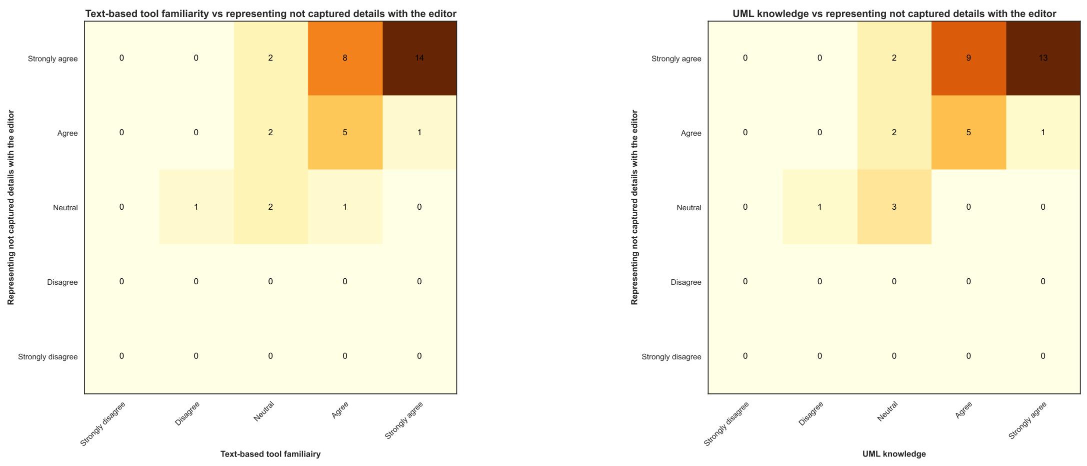


Figure C.21: Relation between user knowledge and editor ability to represent not captured elements.

References

[1] E. Aghajani, C. Nagy, O. Lucero, V. Márquez, M. Vásquez, L. Moreno, G. Bavota, and M. Lanza. Software documentation issues unveiled. *41st International Conference on Software Engineering (ICSE)*, 2019.

[2] Carlos Albuquerque, Kadu Barral, Filipe Correia, and Kyle Brown. Proactive monitoring design patterns for cloud applications. In *Proceedings of the 27th European Conference on Pattern Languages of Programs*, EuroPLoP '22, New York, NY, USA, 2022. Association for Computing Machinery.

[3] Periklis Andritsos and Vassilos Tzerpos. Information-theoretic software clustering. *IEEE Transactions on software engineering*, 2005.

[4] Cyrille Artho and Armin Biere. Combined static amd dynamic analysis. *Electronic Notes in Theorectical Computer Science*, 2005.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, Boston USA, Third edition, 2002.

[6] Terese Besker, Antonio Martini, and Jan Bosh. Managing architectural technical debt: A unified model and systematic literature review. *The Journal of Systems and Software*, September 2018.

[7] Binx.io. The Cloud Survey 2019. Technical report, Binx.io, Netherlands, 2019.

[8] Kyle Brown, Bobby Woolf, Cees De Groot, Chris Hay, and Joseph Yoder. Patterns for developers and architects building for the cloud, 2021.

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns, Volume 1*, volume 1. John wiley & sons, 2008.

[10] Jie-Cherng Chen and Sun-Jen Huang. An empirical analysis of the impact of software development problem factors on software maintainability. *The Journal of Systems and Software*, 2009.

[11] Filipe Figueiredo Correia, Ademar Aguiar, Hugo Sereno Ferreira, and Nuno Flores. Patterns for consistent software documentation. In *Proceedings of the 16th Conference on Pattern Languages of Programs*, pages 1–7, 2009.

[12] Félix Cuadrado, Boni García, Juan C. Dueñas, and Hugo A. Parada. A case study on software evolution towards service-oriented architecture. *22nd International Conference on Advanced Information Networking and Applications*, 2008.

[13] Lei Ding and Nenad Medvidovic. Focus: A light-weight, incremental approach to software architecture recovery and evolution. *Proceedings Working IEEE/IFIP Conference on Software Architecture*, 2001.

[14] Stéphane Ducasse, Michele Lanza, and Sander Tichelar. Moose: an extensible language-independent environment for reengineering object-oriented systems. *2nd International Symposium on Constructing Software Engineering Tools*, 2000.

[15] Neal Ford, Rebecca Parsons, and Patrick Kua. *Building Evolutionary Architectures: Support Constant Change*. O'Reilly, First edition, 2017.

[16] Martin Fowler. Event sourcing. Available at `https://martinfowler.com/eaaDev/EventSourcing.html`, December 2005.

[17] Rodrigo Fronseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. *4th Usenix Symposium on Networked Systems Design & Implementation*, 2007.

[18] Matthias Galster, Armin Eberlein, and Mahmood Moussavi. Early assessment of software architecture qualities. *International Conference on Research Challenges in Information Science, RCIS*, 2018.

[19] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[20] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. *ASE 2013*, 2013.

[21] Giona Granchelli, Mario Cardarelli, and Paolo Di Francesco. Microart: A software architecture recovery tool for maintaining service-based systems. *IEEE Internation Conference on Software Architecture Workshops*, 2017.

[22] Andrina Granic and Nikola Marangunic. Technology acceptance model in educational context: A systematic literature review. *British Journal of Educational Technology*, February 2019.

[23] Md Hasan Ibrahim, Mohammed Sayagh, and Ahmed E. Hassan. A study of how docker compose is used to compose multi-component systems. *Empirical Software Engineering*, 2021.

[24] Sungwon Kang, Seonah Lee, and Danhyung Lee. A framework for tool-based software architecture reconstruction. *International Journal of Software Engineering and Knowledge Engineering*, 2014.

[25] Kong. 2020 Digital Innovation Benchmark. Technical report, Kong, USA, 2020.

[26] Rensis Likert. A technique for the measurement of attitudes. *Archives of Psycology, No. 140, vol 22, pp 5-55*, 1932.

[27] Daniel Link, Ramin Moazeni, Pooyan Behnamghader, and Barry Boehm. The value of software architecture recovery for maintenance. *ISEC'19: Proceedings of the 12th Innovations on Software Engineering Conference*, 2019.

[28] Thibaud Lutellier, Devin Chollak, and Joshua Garcia. Software architecture recovery techniques using accurate dependencies. *ACM 37th IEEE International Conference on Software Engineering*, 2015.

[29] Tiago Maia and Filipe Correia. Service mesh patterns. In *Proceedings of the 27th European Conference on Pattern Languages of Programs*, EuroPLoP '22, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Onaiza Maqbool and Haroon A. Babri. Hierarchical clustering for software architecture recovery. *Transactions on software engineering, vol.30 N.11*, 2007.

[31] Robert C Martin. *Clean architecture: a craftsman's guide to software structure and design*. Pearson, 2017.

[32] Cyrille Martraire. *Living Documentation by design, with Domain-Driven Design*. Addison-Wesley Professional, First edition, 2019.

[33] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology Volume 11 Issue 1*, 2002.

[34] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[35] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *Transactions on software engineering*, 2006.

[36] Liam O'Brien and Christoph Stoermer. Architecture reconstruction case study. *Architecture Tradeoff Analysis Initiative*, 2003.

[37] Liam O'Brien, Christoph Stoermer, and Chris Verhoef. Software architecture reconstruction: Practice needs and current approaches. *Technical Report CMU/SEI 2002 TR 024*, 2002.

[38] Brian Pando and Jose Castillo. Plantumlgen: A tool for teaching model driven development. *7th Iberian Conference on Information Systems and Technologies (CISTI)*, June 2022.

[39] Bruno Piedade, João Pedro Dias, and Filipe F. Correia. An empirical study on visual programming docker compose configurations. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Bruno Piedade, João Pedro Dias, and Filipe F Correia. Visual notations in container orchestrations: an empirical study with docker compose. *Software and Systems Modeling*, 21(5):1983–2005, 2022.

[41] Reinhold Plösch and Sun-Jen Huang. The value of software documentation quality. *14th International Conference on Quality Software*, 2014.

[42] Paul Ralph, Nauman Bin Ali, Michael Felderer, et al. Empirical standards for software engineering research. March 2021.

[43] David Reis and Filipe F Correia. Dockerlive: A live development environment for dockerfiles. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–4. IEEE, 2022.

[44] David Reis, Bruno Piedade, Filipe F Correia, João Pedro Dias, and Ademar Aguiar. Developing docker and docker-compose specifications: A developers' survey. *IEEE Access*, 2021.

[45] RightScale. 2019 State of the Cloud Report, 2019.

[46] Claudio Riva. Reverse architecting: an industrial experience report. *Proceedings Seventh Working Conference on Reverse Engineering*, 2000.

[47] Banani Roy and T.C. Nicholas Graham. Methods for evaluating software architecture: A survey. *Technical Report No. 2008-545*, April 2008.

[48] Julio Sandobalin, Emilio Insfran, and Silvia Abrahao. ARGON: A Tool for Modeling Cloud Resources. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10797 LNCS(November):393–397, 2017.

[49] C.J. Satish and M. Anand. Software documentation management issues and practices: a survey. *Indian Journal of Science and Technology*, 2016.

[50] Sandra Schröder, Mohamed Soliman, and Matthias Riebish. Architecture enforcement concerns and activities - an expert study. *The Journal of Systems and Software*, August 2018.

[51] Kirill Shirinkin. *Getting Started with Terraform*. Packt Publishing Ltd, 2017.

[52] T. Sousa, H. S. Ferreira, and F. F. Correia. A survey on the adoption of patterns for engineering software for the cloud. *IEEE Transactions on Software Engineering*, page 1–1, 2021.

[53] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. Patterns for software orchestration on the cloud. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*, pages 1–12, 2015.

[54] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Messaging systems and logging. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, pages 1–14, 2017.

[55] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Automated recovery and scheduler. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–8, 2018.

[56] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: External monitoring and failure injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–8, 2018.

[57] SusanJamieson. Likertscales:how to(ab)usethem. *Med. Educ*, December 2004.

[58] Mary Sánchez-Gordón and Ricardo Colomo-Palacios. Characterizing devops culture: A systematic literature review. *Communications in Computer and Information Science book series (CCIS,volume 918)*, 2018.

[59] Nick Tune. Self-documenting architecture. Medium blogs. Available at `https://medium.com/nick-tune-tech-strategy-blog/self-documenting-architecture-80c8c2429cb8`, September 2020.

[60] Vassilios Tzerpos and R.C. Holt. Acdc : An algorithm for comprehension-driven clustering. *Proceedings Seventh Working Conference on Reverse Engineering*, 2000.

[61] Guilherme Vale, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. Designing microservice systems using patterns: An empirical study on quality trade-offs. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 69–79. IEEE, 2022.

[62] Aline Vasconcelos and Cláudia Werner. Evaluating reuse and program understanding in archmine architecture recovery approach. *Information Sciences*, 2010.

[63] Gerald M. Weinberg. *The Psychology of Computer Programming*. Dorset House, annual, subsequent edition edition, 1998.

[64] Meng-Jia Wu, Kelly Zhao, and Francisca Fils-Aime. Response rates of online surveys in published research: A meta-analysis. *Computers in Human Behaviour Reports 7 100206*, 2022.