Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Interactive Elicitation of Resilience Scenarios in Microservice Architectures

Christoph Zorn

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Sebastian Frank, M.Sc. Mir Alireza Hakamian, M.Sc. Dr.-Ing. André van Hoorn
Commenced:	November 24, 2020
Completed:	May 24, 2021

Abstract

Context. Elicitation of requirements in software systems is a complex task. Hence, it requires knowledgeable software architects and other domain experts. Especially in distributed environments such as microservice-based systems, this is the case. Components of the system are developed in isolation and commonly share no prerequisites. Therefore, a precise specification is necessary to ensure the availability and performance of the individual components.

Problem. In practice, requirements elicitation of non-functional attributes is often neglected or does not follow a systematic approach. If the elicitation process is performed at all, stakeholders meet in physical group meetings. Here, the presence of domain experts with profound knowledge of the inspected system is required. Such an approach is very time-consuming and can result in high costs. Furthermore, participants are required to cover all essential use cases in a limited number of meetings.

Objective. This work proposes Resirio, a requirements chatbot supporting resilience engineers in the elicitation process of resilience-related attributes. Previous works have introduced interactive solutions for requirements elicitation. However, they lack formalization and precise elicitation methodologies. The proposed approach should make the elicitation process more effective for software architects and more accessible to inexperienced requirements engineers.

Method. Resirio ensures that requirements are elicited in a systematic approach and stored in template-based Architecture Trade-off Analysis Method (ATAM) scenarios. We use a Computational Hazard and Operability (CHAZOP)-based hazard analysis technique to inspect metrics in traces from Zipkin and Jaeger. Hazards identified by the analysis help to examine stimuli that may lead to deviations from the optimal behavior of the system. With the stimulus as an input, users create ATAM scenarios in a conversation with Resirio. Response and response measure, which are parameters of the scenario, are formalized based on Metric Temporal Logic (MTL).

Result. In a user study with software architects, engineers, and researchers from industry and academia, we evaluated Resirio's usability, effectiveness, and support. We compared the user's interaction with the prototype and examined elicited scenarios during the study. The developed prototype gives novice requirements engineers a foundation for fast requirements elicitation but requires advanced features for expert users to define more precise scenarios.

Conclusion. We showcase an interactive solution that enables quick and easy elicitation of resilience-related requirement attributes. Following the systematic CHAZOP-based approach, hazard are identified in traces from Zipkin and Jaeger. In a conversation with Resirio, stakeholders of a microservice-based software system are assisted in refining ATAM scenarios. Results from the user study show that resilience engineers prefer the fast and direct input of *Quick Replies* to written text.

Kurzfassung

Kontext. Das Erheben von Anforderungen in Softwaresystemen ist eine komplexe Aufgabe. Deshalb ist das Wissen von Softwarearchitekten und anderen Fachexperten erforderlich. Speziell in verteilten Systemen wie Microservices ist dies der Fall, da Komponenten losgelöst voneinander entwickelt werden und oftmals keine Voraussetzungen teilen. Aus diesem Grund ist es nötig, eine detaillierte Spezifikation zu erstellen, um die Verfügbarkeit und Performanz der einzelnen Komponenten zu gewährleisten.

Problemstellung. In der Praxis werden Anforderungen oftmals vernachlässigt oder folgen keinem methodischen Ansatz. Werden Anforderungen erhoben, dann passiert dies in der Regel in Präsenzmeetings im Beisein von Experten mit tief greifendem Wissen über das untersuchte System. Dieser Prozess erfordert sehr viel Zeit und kann teuer werden. Überdies sind Teilnehmer dazu angehalten, die wichtigsten Anwendungsfälle in einer begrenzten Anzahl an Meetings zu finden.

Zielstellung. Diese Arbeit stellt Resirio vor, einen Chatbot, der Analysten während der Anforderungserhebung unterstützt. Andere Arbeiten haben interaktive Lösungen für die Anforderungserhebung vorgestellt, welche allerdings keine präzise oder nur mangelnde Formalisierungen verwenden. Der vorgestellte Ansatz soll die Anforderungserhebung für Softwarearchitekten beschleunigen und zugleich zugänglicher für unerfahrene Analysten machen.

Methode. Resirio stellt sicher, dass Anforderungen mit einer systematischen Methode erhoben und in ATAM-Szenarien gespeichert werden. Mit einer auf CHAZOP basierenden Methode werden Metriken aus Traces von Zipkin und Jaeger untersucht. Risiken, die durch die Analyse identifiziert werden, helfen dabei, Stimuli zu erkennen, welche zu einem Abweichen vom Normalzustand einer Software führen können. Mit dem Stimulus als Eingabe können Nutzer ATAM-Szenarien in einer Konversation mit Resirio erstellen. Die Response und das Response Measure, welche Parameter des Szenarios sind, werden basierend auf MTL formalisiert.

Resultat. In einer Nutzerstudie mit Softwarearchitekten, Ingenieuren und Forschern aus Industrie und Wissenschaft wurde Resirio auf Benutzbarkeit, Effektivität und Hilfestellungen untersucht. Die erstellten Szenarien und Interaktion zwischen Nutzer und Chatbot wurden miteinander verglichen. Der entwickelte Prototyp kann unerfahrenen Nutzern einen schnellen Einstieg in die Anforderungserhebung ermöglichen, erfahrene Nutzer benötigen allerdings erweiterte Features.

Zusammenfassung. Wir präsentieren eine interaktive Lösung zur schnellen und einfachen Anforderungserhebung von Elastizitätsattributen. Risiken werden mithilfe des auf CHAZOP basierenden Ansatzes aus Traces von Zipkin und Jaeger erkannt. In einer Konversation mit Resirio werden Nutzer bei der Verbesserung von ATAM-Szenarien unterstützt. Ergebnisse der durchgeführten Studie zeigen, dass Analysten den schnellen und direkten Weg der sogenannten *Quick Replies* gegenüber Texteingaben bevorzugen.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API Application Program Interface. 27

ATAM Architecture Trade-off Analysis Method. iii

CF Cloud Foundry. 58

CHAZOP Computational Hazard and Operability. iii

CLI Command Line Interface. 53

CSS Cascading Style Sheets. 57

CTK Chaos Toolkit. 39

CTL Computation Tree Logic. 30

DAG Directed Acyclic Graph. 14

DAPI Dapper API. 28

DF Dialogflow. 54

DG Directed Graph. 48

FMEA Failure Mode and Effect Analysis. 10

FT Fault Tree. 14

FTA Fault Tree Analysis. 4

GORE Goal-Oriented Requirements Engineering. 36

GQM Goal-Question-Metric. 37

GUI Graphical User Interface. 13

HAZOP Hazard and Operability. 10

HTML Hypertext Markup Language. 57

HTTP Hypertext Transfer Protocol. 28

IP Internet Protocol. 46

JS JavaScript. 57

JSON JavaScript Object Notation. 29

KAOS Knowledge Acquisition in autOmated Specification. 2

LISA Low-level Interaction Safety Analysis. 22

LTL Linear Temporal Logic. 30

MRMM Microservice Resilience Measurement Model. 39

MS Microservice. 43

MSA Microservice Architecture. 10

MTL Metric Temporal Logic. iii

NLG Natural Language Generation. 31

NLP Natural Language Processing. 31

NLU Natural Language Understanding. 36

OCL Object Constraint Language. 37

PSF Property Specification Framework. 31

PSP Property Specification Pattern. 30

REST Representational State Transfer. 53

RPN Risk Priority Number. 10

SHARD Software Hazard Analysis and Resolution in Design. 21

STAMP Systems-Theoretic Accident Model and Processes. 20

STPA System-Theoretic Process Analysis. 10

SUS System Usability Scale. 73

SVG Scalable Vector Graphic. 55

TCP Transmission Control Protocol. 62

UI User Interface. 29

UUID Universal Unique Identifier. 62

1 Introduction

Failures occur in any software system. Outages of programs can prove costly for software providers with damaging costs often beyond millions of dollars [Cha05; DNA+21]. Besides financial consequences, failures can damage the reputation of responsible companies in the long term or even lead to life-threatening situations for people [JH19; Som17]. Protection against any type of failure is virtually impossible and requires stakeholders to define precise function boundaries and quality of service metrics. One way to ensure the long-term resilience of a software is to apply the method of resilience engineering. Resilience describes a system's ability to recover from failure [WAVV12]. The task of a resilience engineer in the domain of software is to build a system for failure [BT12; HWL06]. In the case of a failure, weak components can be identified and prepared for future risks and hazards. A software hazard is defined as a state of a system that, in combination with worst-case conditions, leads to a situation of danger [Lev95]. The importance of hazard analysis in a system is to classify them by likelihood and effect. For example, a non-functional climate control has less dramatic effects than a wrong rocket flight path. A software's resilience is not always reflected in a full test coverage or optimal performance in a test environment. Rather than testing individual components of a system, it is more important to have a holistic view of the system and inspect its behavior during productive conditions. Specification of non-functional requirements is crucial for any software system that aims for resilience [KEM20]. It gives the stakeholders, especially software architects and engineers of the system, the possibility to rely on an agreed set of predefined values. Furthermore, it simplifies the search for faults, leads to saving of running costs, and improves the overall quality of the software [HBB+11].

Creating a precise specification is a challenging task, even for experienced software developers [NE00]. The effort becomes increasingly complex the more stakeholders are involved [PSR10]. In practice, the aspect of requirements specification for resilience attributes is frequently neglected or not done at all, as a reason for lack of time and inexperienced personnel [NL03; PEM03]. If no specification is defined for the individual components of the software, it can not operate predictably. As a result, the system runs in a state of undefined behavior, and failures might not be detected. In traditional approaches, the process of requirements elicitation is done in group meetings where the stakeholders meet in person. Participants of these meetings are required to have deep knowledge of the system. Usually, the task of resilience elicitation is done by analysts, system architects, or experts in resilience engineering that are solely concerned with this task. However, particularly resilience-related non-functional requirements attributes are usually not well specified [CNYM12]. Due to a lack of a holistic view of the software, most developers and engineers do not get into contact with the elicitation of specifications. Therefore, they lack experience in these domains. Also, the process of elicitation requires a lot of resources, namely expert personal, and a lot of time. Traditional approaches perform group meetings or workshops that span over several hours or days [HT11; KA00]. Meetings can result in costly expenses the longer they take. Another problem with the traditional approaches is that they all focus on the top-down perspective. Requirements that are specified during the design time have to be completed in the group meetings. However,

requirements may change during development, and it is necessary to update the specifications. As this practice requires a high amount of effort, we want to show that an interactive solution for resilience requirements is more effective. Other works have proposed interactive solutions for non-functional requirements elicitation through conversational interfaces. Approaches presented by Pérez-Solder et al. [PGL17] and Surana et al. [SGS+19] offer highly interactive chatbots. Work presented by Rietz [Rie19] uses the formal approach of laddering to accelerate the feedback loop while Arruda et al. [AMSW19] utilize the goal-oriented approach of Knowledge Acquisition in autOmated Specification (KAOS). However, their approaches are either lacking insights about the inspected software systems through system metrics or miss a structured methodology to specify elicited requirements.

The approach introduced in this work aims for a different strategy than used in the traditional approaches while improving the drawbacks of existing works. Rather than defining the specifications before the system's design, it should be possible to specify them at any stage of the system development lifecycle. A semi-automated bottom-up approach in combination with a conversational interface is realizing this. Benefits include accelerated elicitation and an underlying systematic methodology. On the one hand, the components of the systems are known at any time. On the other hand, requirements are stored in scenarios that can be modified as the system changes. Also, an instrumented software can run an automatic analysis to identify and update changing parts of the system. Furthermore, scenarios offer a high degree of formalization, making it easier for requirements engineers to derive system specifications.

Scenarios that we use in this work are based on the works of Bass et al. [BCK03] who refined the scenario description of the ATAM method developed by Kazman et al. [KKB+98]. ATAM presents a structured method to mitigate risks and evaluate trade-offs of architectural quality attributes against each other. Elicited quality attributes are described in different types of scenarios. A use case scenario describes a user's intended interaction with the system [KKB+98]. Growth scenarios represent anticipated future changes, while exploratory scenarios are used to explore the system boundaries [KKB+98]. Exploratory scenarios usually utilize the non-functional requirements of a system. However, these types of scenarios are defined loosely and can only be understood by humans. Bass et al. [BCK03] refined the exploratory scenario description through a fixed set of parameters. Each parameter can specify a particular property of a software system, such as a component or its quality attributes. Such a more formal description provides a better understanding of scenarios and helps to accelerate the creation of scenarios. This work aims to extend the description of a resilience scenario by Bass et al. and automate the creation of such scenarios through an interactive interface. In the following work, we will use the name ATAM-based scenarios and resilience scenarios interchangeably.

The developed approach was evaluated in an expert user study with software architects and engineers from industry and academia. The emphasis of the study was to identify benefits in the elicitation process through an interactive solution and assess the quality of the results produced by the prototype. As novice resilience engineers and expert software architects participated in the study, we compared the prototype's supportive features for each user group. Furthermore, we evaluated the effectiveness and usability of the presented approach and received suggestions for features and future improvements.

The evaluation of the prototype shows that it provides benefits for both less experienced and expert users. Feedback from all participants suggests that the presented approach can be used to produce ATAM-based scenarios very quickly. Furthermore, study participants of the study reported a

learning effect on requirements elicitation during the study. However, there are also drawbacks to the prototype. On the one hand, text interactions with the chatbot were not received positive due to the weak text recognition. On the other hand, participants reported that the analysis method of the prototype poses to few suggestions to create meaningful ATAM scenarios. Comments on the future improvement of the prototype propose an its integration with the traditional method of group meetings. Combining the interactive approach with knowledge from multiple requirements engineers is expected to accelerate the traditional approach while producing more valuable results.

1.1 Scope of this Work

The scope of this work is to simplify the requirements elicitation process by introducing a conversational interface in the form of a chatbot. This chatbot is built on a specifically tailored hazard analysis method for microservices based on CHAZOP, utilizing the format of ATAM scenarios. The analysis uses a system description extracted from traces of Zipkin[1] and Jaeger[2]. Non-experts in the domain of requirements elicitation and expert resilience engineering should interact with this chatbot in written English text. In the conversation with the bot, a user can continuously improve scenarios that serve as a specification for system components. These scenarios contain a description that represents a defined state of the whole system.

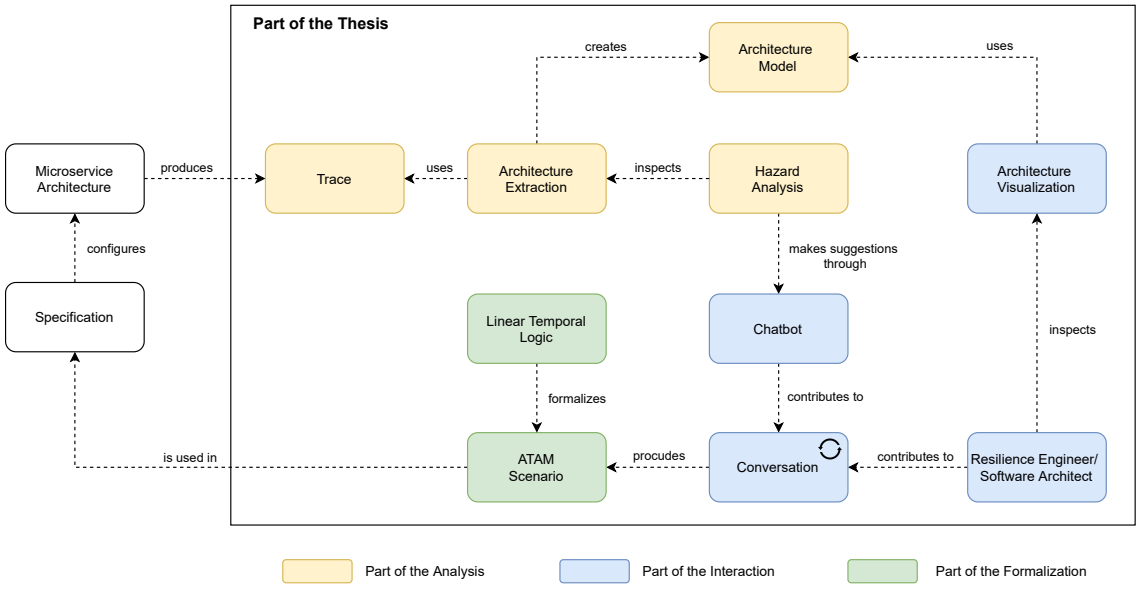


Figure 1.1: Methodologies and processes involved in the approach to reach the objective.

[1] https://zipkin.io/
[2] https://www.jaegertracing.io/

1.2 Objective of this Work

This work's objective is to extend on existing approaches in the area of requirements elicitation. Due to the absence of automation and the need for manual work in these approaches, a strong need for acceleration exists and better access for non-experts is necessary. As this problem touches multiple areas of interest, a division into smaller challenges can be made. The visualization of the objective and the division of concerns is given in Figure 1.1.

First, faster elicitation should be achieved through the application of an automated hazard analysis method. Several existing hazard analysis methods (i.e., CHAZOP, Fault Tree Analysis (FTA)) can deliver weaknesses of a system but rely on participants' knowledge during the process. Automation for the hazard analysis approaches should be achieved with the extraction of architectural information of an inspected microservice-based system. Architecture extraction tools do already exist but have to be connected with hazard analysis methods.

Secondly, resilience scenarios have to be created with the results from the automated hazard analysis. Previous works from Kesim et al. [KW20] have adapted the scenario design from Bass et al. [BCK03] and used the elicitation approach of ATAM on an existing software system through an in-person workshop. The result of the automated hazard analysis is used to fill the structure of the ATAM-based resilience scenarios.

Thirdly, these resilience scenarios should be defined through formalization for a more detailed underlying description. Requirements elicitation is mostly done informal and is dependent on the experience of participants and experts involved. This work applies formalization based on MTL and related works to specify precise resilience scenarios.

Finally, for the easier use of the presented approach, a chatbot based on the work of Beck [Bec20] is extended and adapted to make it work within the scope of this thesis. The chatbot allows for easier interaction between a user and the inspected microservice-based software system.

An expert user study is conducted to evaluate our approach. Results from the study help to highlight the benefits that this work provides.

1.3 Contributions

This thesis is concerned with a multitude of different topics and has a different focus for each aspect. In the following, each contribution of this work is highlighted.

- Architecture model extraction from Zipkin and Jaeger traces.
- CHAZOP-based hazard analysis using the architecture model and runtime metrics information of traces.
- Extension of the concrete quality attribute scenario based on Bass et al. [BCK03].
- Implementation of a prototype for interactive elicitation of resilience attributes for ATAM-based scenarios through the help of a chatbot [Zor21a].
- Execution of a user study with novice and expert users from industry and academia to assess the capabilities of the developed solution.

Thesis Structure

The next chapters of this thesis are divided as follows:

Chapter 2 – Foundations: The applied research methodology, tooling and methods that are used in other chapters of this thesis are given. Furthermore, methods in the area of requirements engineering, tracing, hazard analysis, and linear temporal logic are introduced.

Chapter 3 – Related Work: Introduces other work about requirements engineering and requirements elicitation that is related to this thesis.

Chapter 4 – Concept and Integration: The concept for the practical part of this thesis is given.

Chapter 5 – Resirio: The implementation of the concepts introduced in Chapter 4 is described. The designed architecture, used technologies, and interaction of components is highlighted.

Chapter 6 – Evaluation: Presents the design, execution, and results of the conducted user study. The results emphasize usability, effectiveness, and quality of the presented approach.

Chapter 7 – Conclusion: The thesis is concluded by summarizing the previous chapters. Collected results are discussed, and an outlook on future work is given.

Appendix A – Study Documents: The supplementary material for the user study is given. The study invitation letter, consent form, task description, and questionnaire are given.

Appendix B – Screenshots of the Prototype: Screenshots of the prototype are provided.

The implementation of the presented prototype is publicly available on GitHub [Zor21a] and Zenodo [Zor21b].

2 Foundations

This chapter presents the foundation of this work. Relevant methodologies and technologies used in this thesis are given. First, the focus is set on hazard analysis methods. Afterward, methods for architecture extraction are presented. In the end, methods for formalization and language transformation are given. Each section introduces a category of related methodologies or technologies that we want to utilize in the practical work of this thesis. Section 2.1 describes how foundation literature was collected. Section 2.2 introduces ATAM. ATAM presents a way to mitigate risks and evaluate trade-offs of architectural quality attributes against each other. We want to use the idea of resilience scenario presented by ATAM to collect resilience-related quality attributes in a structured way. Section 2.3 establishes an overview of hazard and risk analysis methodologies that were inspected on benefits and drawbacks and the use for this thesis. One of the presented methodologies or a related one builds the basis for analysis in the practical part of this work. A tabular comparison is provided at the end of the section that highlights the benefits and drawbacks of each method. Section 2.4 introduces the tracing technology and prominent technologies. Results from tracing tools are used to create an architectural representation of a microservice-based software system. Section 2.5 focuses on formalization through temporal logic. Formalization is used to verify parameters of ATAM scenarios.

2.1 Literature Research Methodology

The starting point for the literature research of this thesis were scientific articles and notes on existing technologies related to the topic of hazard analysis given by the supervisors of this work. Used articles were explored further until a good enough understanding was collected to make assumptions and opinions on related topics. The first goal was to access the primary sources of scientific papers. For this, references were inspected on authors of the primary sources. Google Scholar[1] was used to search for the authors' names and their works. The focus was on handbooks and official guidelines of the topics by the original authors. If the original source of the topic was available, a summary was created. Further, the contents of primary sources were inspected and scanned for citations of related works. Here, the selection was done by (i) relevance (given through Google Scholar), (ii) number of citations, and (iii) names of the author(s). When a topic of relevance was found in the secondary sources, a new search with the corresponding titles and authors was done. For secondary literature the search engine Google[2] was used. Results from the search engine were scanned for blog posts or websites from authors of primary or secondary sources.

[1] https://scholar.google.com/
[2] https://www.google.com/

In the following, we give an example of how we found out details on the FTA methodology and its authors. FTA was used in a scientific article provided by the supervisors ([KWK+21]). The authors did cite an article with the name *System Safety and Computers: A Guide to Preventing Accidents and Losses Caused by Technology* by Nancy G. Leveson. A search for the title of this article did not retrieve any beneficial results, so we inspected the original author's page on Google Scholar. There, we found the original name of the published paper and created a new search with the original name. Once we found the paper in PDF form, we scanned the paper for the occurrence of the words FTA. We read the corresponding paragraphs and concluded that this work was not a primary source for FTA, since other authors of FTA were mentioned ([LGTL85]) that predate the article by Leveson. We looked for the article with another search and repeated the procedure to scan for paragraphs with the words FTA. There we found the mentioning of an IEC standard ([Com+90]) and a reference to the original authors (HA Watson et al. [Wat+61]). Unfortunately, this reference leads only to a citation and no article or book. We started a new search on Google Scholar with the terms *Fault Tree Analysis* and sorted the results by relevance. Scanning through the most cited references, we found agreement that the original author of FTA are indeed Watson et al. [Wat+61]. The most relevant results also pointed to the handbook of Vesely et al. [VGRH81] which we used as a secondary source for two reasons. The publish date (1981), and the number of citations the handbook has (1886). If we were unsure about the authors' statements in the handbook, we looked into other articles about the same topic.

When searching for technologies, no scholarly articles were found in most cases. Open-source platforms or the organizations' websites that provided the technology were inspected with Google. User guides, documentation, and code repositories of the technology were inspected. If a public version of the technology was available we installed and tested it.

In the following, we provide an example how we inspected the Zipkin tracing tool. The most relevant result retrieved by Google leads to the official Zipkin page (`https://zipkin.io/`). The content of the page offers a short introduction about the tool and provides links to the official GitHub repository (`https://github.com/openzipkin/zipkin/`). On the GitHub page further documentation exists on how to install and execute the tool.

2.2 Elicitation

Requirements elicitation is the practice of finding requirements of a system from stakeholders. This process is usually done in group meetings through brainstorming and workshops to identify use cases and specifications for a system. Requirements also help to define the boundaries and operational metrics of the system [Poh10]. The task of elicitation is non-trivial since it depends on the expertise and experience of participants of the meeting. In the following, the ATAM is introduced, which is a method that defines a structured process to compare different software architectures against each other based on requirements and quality attributes. Furthermore, a summary of a practice report is given where the authors used the ATAM approach in an industrial practice report.

2.2.1 Architecture Trade-Off Analysis Method

The ATAM is a structured analysis technique for architectures in the domain of software development [KKB+98]. The technique lets quality attributes like performance, security, and availability of an architecture compete against each other to find different architectural styles. A decision to choose one architectural style over another may benefit one or multiple attributes but at the same time is worsening one or multiple other ones [KKC00]. Decision-making on the architectural style is dependant on the stakeholders and therefore does have not only technical but also social reasoning. A safety-critical application may prefer the maximum amount of security while renouncing in performance. Quality attributes do not always have to be from a technical domain but can also represent outside influences from management such as time or cost [BM+01].

During the phases of ATAM depicted in Figure 2.1 practitioners of ATAM have to inspect existing requirements of software or create them if they do not exist. As the goal-oriented process tries to maximize the value of specific quality attributes, constraints have to be designed that describe the minimally acceptable trade-off for all inspected attributes. Furthermore, requirements have to be evaluated on worst-case scenarios that have to be elicited first.

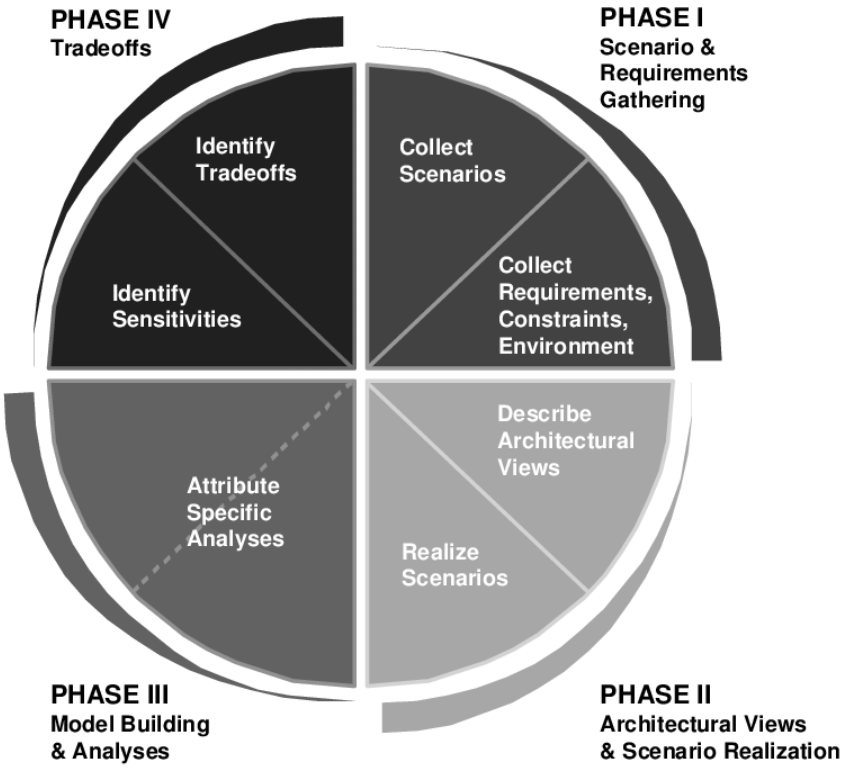


Figure 2.1: Phases of ATAM taken during the analysis [KKB+98].

In an experience report from Kazman et al. [KBK+99] the authors show that the decision between two competing architectures can have significant differences in the results of attribute measurement. An attributes measurement can increase by multiple orders of magnitude in one architecture while

being completely unacceptable in the other. Therefore, the process of ATAM is done in an iterative attempt to make unimportant attributes not irrelevant. The process has to be done repeatedly until measurements for worst-case scenarios have reached an acceptable value.

2.2.2 Elicitation in Practice

In a student research project, Kesim et al. [KW20] performed an industrial case study on the evaluation of a Microservice Architecture (MSA) with a focus on the resilience attributes of the system. In a workshop, they used the ATAM approach in combination with fault tree analysis to identify potential hazards in the inspected system. Furthermore, they elicited requirements from the stakeholders of the MSA. These requirements were then transformed into scenarios based on the scenario template from Bass et al. [BCK03]. As final step scenarios were transformed into automated chaos experiments to evaluate the resilience of the system. These experiments were executed once with resilience patterns and once without. Metrics collected during both executions showed that applied resilience patterns have a positive effect on the resilience of the system.

This work utilizes the process of ATAM with the elicitation of quality attributes of architecture from users. Results that are gathered from users are then filled into the template format of the resilience scenarios. The template scenario format is based on the structure introduced by Bass et al. [BCK03].

2.3 Hazard and Risk Analysis

Several methods exist to analyze the quality attributes of a system. This section summarizes those that are applicable in the context of this work. In the following, hazard analysis methods are introduced that have been established outside of software development. Some of these approaches have been adapted to computer systems. These methods are highlighted because they are regularly used in the industry and therefore can contribute to the outcome of this work [Bay15; Kle99]. The focus lies on the automation of the techniques and at what point in the systems lifecycle they can be used. Hazard and Operability (HAZOP), FTA, and Failure Mode and Effect Analysis (FMEA) are explained in great detail because they count to the most used hazard analysis techniques [Bay15]. Furthermore, all three techniques exist for a long time and provide many resources for research. In addition, these techniques have inspired risk analysts to extend, adapt, and refine the approaches. Therefore, it is essential to understand the underlying methodologies used in HAZOP, FTA, and FMEA. Techniques mentioned in this section were chosen because they provide aspects of hazard prioritization (Risk Priority Number (RPN) of FMEA), cascading effects (FTA), and isolated views on the hazard (constraints in System-Theoretic Process Analysis (STPA)). Some of these methods (i.e., RPNs) are applied in the practical part of the thesis.

2.3.1 Resilience

Resilience is one of the main concepts of MSAs. The definition of resilience covers multiple aspects of software development that can be found in other software architectures besides microservices. In software development, resilience describes a software system's ability to recover and maintain a

state of failure [HWL06]. This behavior is sometimes called robustness. The type of failure can have many origins. Attackers may maliciously inject errors, failures can occur by accident, or the system experiences a change of states, either by inside or outside stimuli.

Especially in microservice-based systems, it is interesting to identify the interplay between services. A problem in one component should not cascade through the whole system and cause it to fail [New15]. Instead, the problem should be isolated by resilience mechanisms that prevent the issue from propagating further so that the rest of the system can still operate [YDW+19]. Besides the absorption of the error, a software system should still provide an acceptable level of service to its users. If the service that experienced the failure is a critical component of the whole system, it is crucial to recover this service. If no resilience mechanism can return the service to its optimal behavior or maintain its availability at all, the only option is to restart the service.

2.3.2 Hazard and Operability Study

The HAZOP is a guide word-driven, team-based, qualitative risk analysis technique as defined by IEC standard 61882 [SI01]. It is a structured and systematic analysis method to examine a defined system for operation risks, environmental risks, and hazards. HAZOP is generally considered an effective and straightforward hazard identification method [WJG01]. The team-based effort relies on predefined roles such as the study leader, scribe, engineer, operator, vendor specialist (optional), and maintainer (optional). Although it is not required that all roles are occupied, any participant should be familiar with the analyzed system. The size of the team should be limited from four to eight people. Furthermore, HAZOP depends on subject matter experts that have profound knowledge and have the ability to predict the behavior of the system when deviations occur[WJG01]. The goal of a HAZOP study is to identify hazards and operability problems by following one simple principle. “Risk events are caused by deviations from the systems design specification” [FH94]. The IEC standard defines the central terms of HAZOP as follows [SI01]:

- Hazard: Potential source of harm. A hazard may lead to multiple forms of harm.
- Harm: Physical injury or damage to the health of people or damage to assets or the environment.
- Risk: A combination of the probability of occurrence of harm and its severity.

Identifying these risks, hazards, and weaknesses are done by a bottom-up process that is separated into four phases [Kle99; SI01].

1. Definition: In the first phase, team members are chosen, and responsibilities are assigned. As mentioned before, all members should have an understanding of the system that is analyzed. Furthermore, the scope of the study is defined that sets the goals and boundaries.
2. Preparation: Here, the outline of the study is planned, and data of the system is collected. In addition, a schedule is created, and time planning is done. The team members choose the guide words appropriate for the domain of the system. Common examples are: no, more, less, as well as, part of, reverse, other than. It is important to note that these guide words should reflect possible deviations that can occur. Therefore, the selection of guide words is usually done by domain experts.

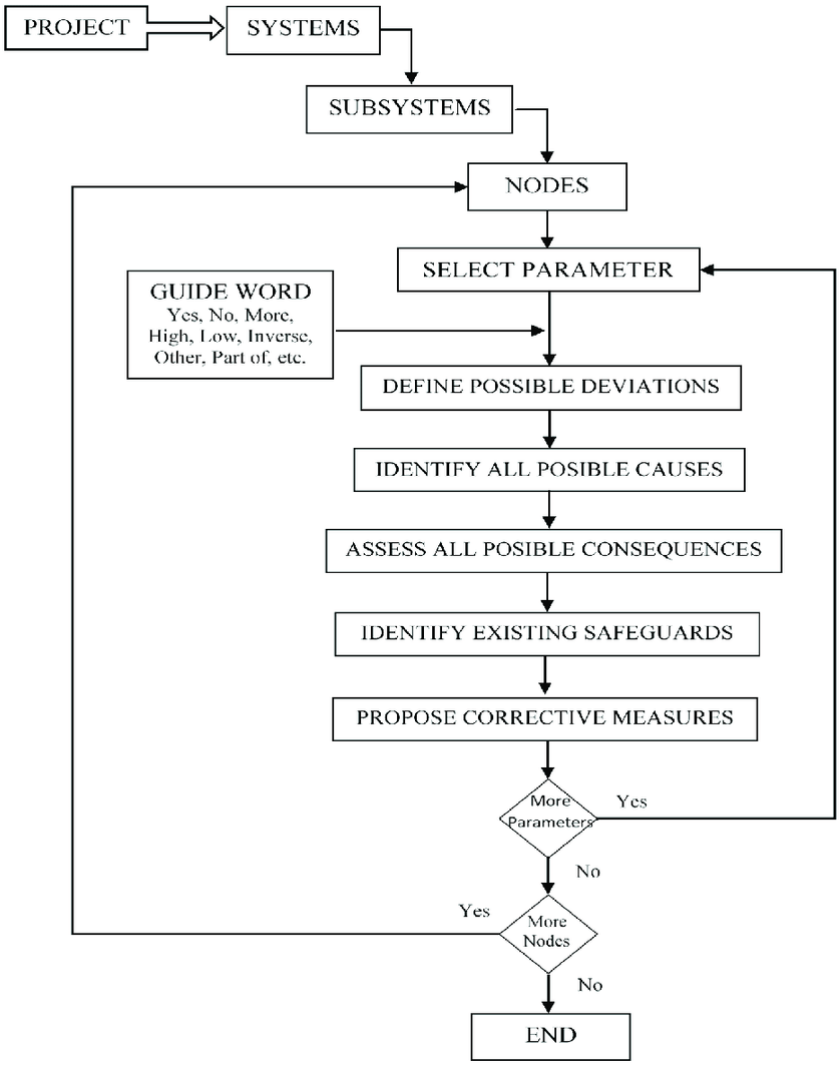


Figure 2.2: Steps taken during the examination phase of HAZOP [FGGB17].

3. Examination: In the first step of the examination phase, the inspected system is divided into logical components, which allows for a more targeted risk analysis. Based on the analysis domain, it is also possible to inspect the system's design, environment, or procedures besides the components. In the next step, guide words are applied to the components of the system, and possible causes and consequences of the deviation are brainstormed. In the last step, safeguards and protection mechanisms are identified that can be applied to the component to protect it from future deviations of the same kind possibly. An example of how guide words in combination with the assessment of consequences and causes can be used is shown in Table 2.1. The process of the examination phase is depicted in Figure 2.2.

Guideword	NO	LESS	MORE	OTHER THAN	…
Component	Flow	Flow	Flow	Flow	…
Deviation	No flow	less flow	More flow	Complete substitution	…
Cause(s)	Centrifugal pump lost priming or valve is closed totally	Valve closed partially	Water passes through the valve event when its closed	Wrong delivery from vendor or wrong material is chosen	…
Consequence(s)	Resulting in a highly concentrated solution	Resulting in highly a concentrated solution	Overflow of batch mixer	Depends on the substitution	…
Safeguard	Monitoring of the centrifugal pump	Monitoring of the centrifugal pump	Install level indicator	Introduce check on chosen material before use.	…

Table 2.1: A typical result of a HAZOP analysis that is concerned with the flow of a centrifugal pump [ONK17].

4. Documentation: In the last phase of the HAZOP, the results of the examination phase are recorded, and a final report is generated. This report should be done appropriately based on the skills and knowledge of the stakeholders involved. Outcomes of the report can be the mentioned guide word table or a cause and consequence matrix where causes and likelihood for each hazard are put into contrast.

Several benefits and drawbacks of HAZOP were identified, which are listed in Table 2.2. The contents of the table relies on experiences from practiioners of HAZOP. For example Hulin et al. [HT11] highlight the time needed for group meetings and resulting high costs. From their experience, a HAZOP meeting lasts about one and a half days. Therefore, based on the group size, risk assessment costs between six and twelve person-days. Khan et al. [KA00] criticise the effectiveness of HAZOP and present an approach to accelerate it through a Graphical User Interface (GUI). Paul Baybutt [Bay15] criticises HAZOP for its reliance on the practitioners experience which oftentimes is subject to human errors.

2.3.3 Computational Hazard and Operability study

CHAZOP sometimes also referred to as control systems HAZOP or computer HAZOP is an extension to HAZOP that is concerned with the hazard analysis of Distributed Control Systems [FH94]. In the traditional HAZOP, only physical components of a system are inspected because non-physical components are expected to never fail [And91]. CHAZOP does not extend the methodology of HAZOP but rather limits the inspected system components to software and hardware of computers.

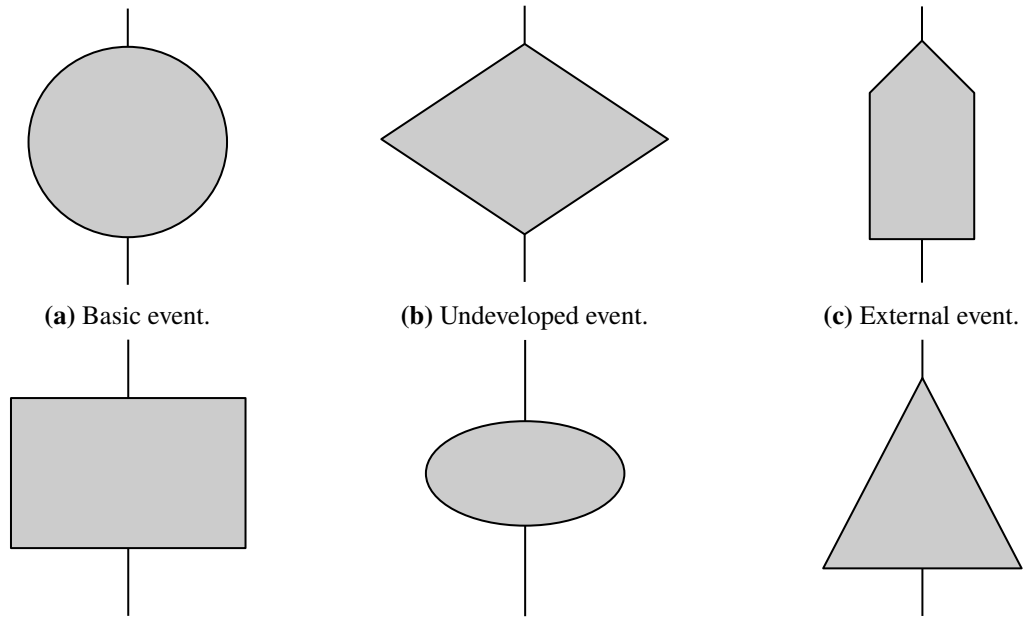
Benefits	Drawbacks
Simple and intuitive analysis method.	Requires at least one team member to be a domain expert.
Brainstorming methodology yields fast results.	Risks that were not discussed still exist.
Same process for every system.	Safeguards may introduce new risks.
Division into components makes it easier to identify areas of risk.	No interplay between components is analyzed.
Helps to identify hazards that are otherwise difficult to detect.	Guide words are chosen by team members.
	There exist no means to assess the effectiveness of safeguards.
	It is not possible to rank hazards by severity in the original HAZOP.
	Guide word table may contain redundant information.

Table 2.2: We identified benefits and drawbacks of using HAZOP to analyze risks and hazards.

2.3.4 Fault Tree Analysis

FTA was first introduced in 1961 by Bell Telephone Laboratories [Wat+61] and provided a graphical top-down approach for root cause analysis. FTA investigates the dependencies of events in a systems design and the possible outcome of failures in these events [RS15]. The result of the FTA is a Fault Tree (FT) that visualizes interconnected events in a Directed Acyclic Graph (DAG) . Components of a fault tree are events, gates, and transfer symbols.

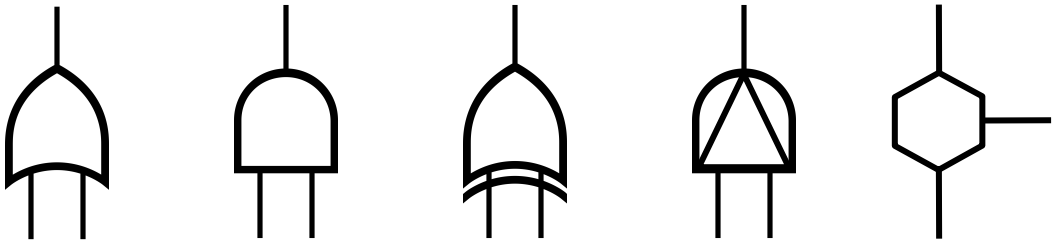
Events represent the failures of a sub-component of the observed system. There exist several events that have different meanings. Vesely describes in the Fault Tree Handbook all events in detail [VGRH81]. In the following, a summary was created. An overview of current events is depicted in Figure 2.3. The most important event is the top event that describes an undesired event or fault that is under inspection. The top event, sometimes called the root event, is the starting point of the FTA. The analysis is done backward, from the root event (top to bottom) to determine a root cause that is present in one of the leaves of the tree [Kab17]. Figure 2.3a shows a basic event that has no further development and is therefore at the leaves of the tree describing an initial error [VGRH81]. An undeveloped event represents insufficient information about an event and is shown in Figure 2.3b while an external event is shown in Figure 2.3c sometimes called house event is expected to occur outside of the inspected system components and does not manifest an error [VGRH81]. In Figure 2.3d an intermediate event is depicted. It follows as a result of a single previous event or multiple events combined by logic gates. The conditional event shown in Figure 2.3e is used to record a condition or restriction to any logic gate [VGRH81]. Finally, there exists the transfer event pictured in Figure 2.3f that is used to indicate that an FT is continued further in another place, or the current FT is part of an existing FT.



(a) Basic event. **(b)** Undeveloped event. **(c)** External event.

(d) Intermediate event. **(e)** Conditional event. **(f)** Transfer event.

Figure 2.3: Types of events that can occur in a FT.



(a) Or gate. **(b)** And gate. **(c)** Exclusive or gate. **(d)** Priority and gate. **(e)** Inhibit gate.

Figure 2.4: Types of gates that represent how a failure propagates through a FT.

Besides events, there also exist *Gates* which are based on Boolean logic and combine events with each other. The evaluation of gates is done from the bottom to the top. Gates usually have one output and can have two-to-many inputs. The fault on the output of the OR gate occurs if at least one input event occurs [VGRH81]. In contrast to this, the AND gate requires all input events to occur in order for the output fault to happen. The output event of the exclusive OR gate will only occur if exactly one input event occurs. Apart from the typical OR, AND, and XOR gates shown in Figure 2.4a, Figure 2.4b, and Figure 2.4c respectively, there are two more gates only used in the FTA. On the one hand, the priority AND gate propagates the error to the output if both input events occurred in a specific order. On the other hand, there is the inhibit gate that propagates exactly one input event to precisely one output event when a condition attached to the third edge is fulfilled. These gates are visualized in Figure 2.4d and Figure 2.4e.

Based on the IEC standard 61025 an FT consist of the following steps [Com+90]:

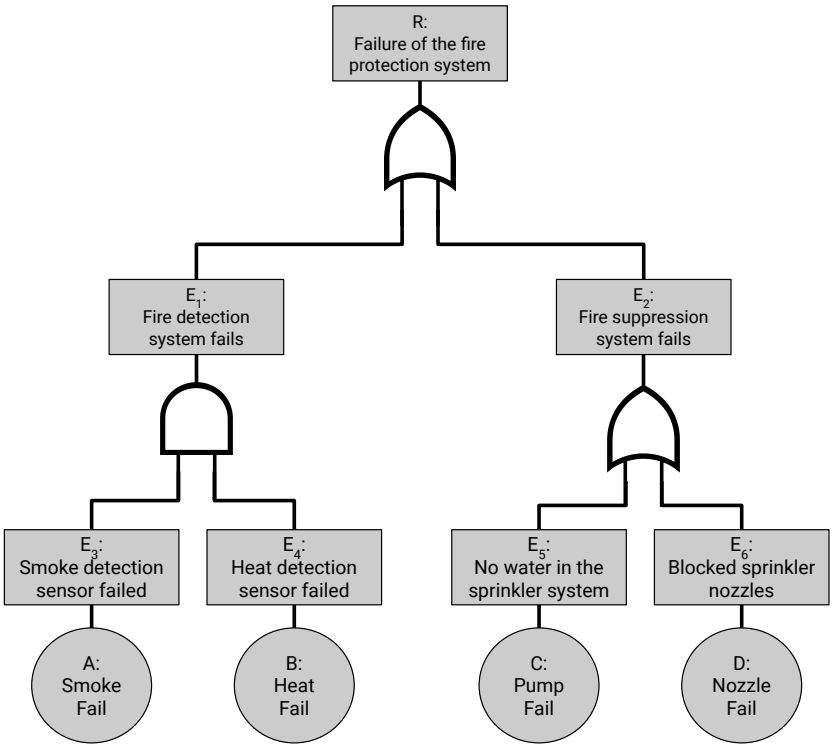


Figure 2.5: A fault tree that analyses the failure of a fire protection system [And98].

1. System definition: In the first step, the analyst should define the scope of the analysis and define the outcome of the analysis. It should also cover definitions of what constitutes to a failure and the boundaries of the system. Similar to the HAZOP approach defined in Section 2.3.3 the analysis requires domain experts that should already be familiar with or have the ability to get familiar with the design of the system to highlight functions and operations of the system [Com+90].
2. Fault tree construction: The second step consists of defining the top event and the following construction of the FT. It should be noted again that this process can only be done by analysts who understand the system.
3. Evaluation of the FT: The evaluation of the FTA is split into two parts:
   a) Qualitative evaluation: The qualitative evaluation is concerned with the detection of (minimal) cut sets and (minimal) path sets [RS15].

      The *minimal cut set* is the smallest number of events which, if they occur in combination, will cause the root event to fail as well [VGRH81]. In the following an example of the fire protection system is given which is depicted in Figure 2.5, where an OR gate yields the addition and an AND gate in the multiplication of the input events:

- Logical notation of all sub trees:

$R = E_1 + E_2$ (Fire protection system fails)
$E_1 = E_3 * E_4$ (Fire detection system fails)
$E_2 = E_5 + E_6$ (Fire suppression system fails)
$E_3 = A$ (Smoke detection fails)
$E_4 = B$ (Heat detection fails)
$E_5 = C$ (Pump fails)
$E_6 = D$ (Nozzle fails)

- Substitution of E_1 and E_2:

$R = E_3 * E_4 + E_5 + E_6$

- Substitution of E_3, E_4, E_5 and E_6:

$R = A * B + C + D$

R is the root event, E_x with $x \in 1, 2, 3, 4, 5, 6$ an arbitrary event, while A, B, C, D represent the failure of a component, thus the set of all minimal cut sets of the FT is $\{A * B, C, D\}$. This means that either the whole fire detection system fails or one component of the fire suppression system fails, in order for the whole system not to work anymore.

The result of the *minimal path set* tells which undesired event one has to prevent for the whole system still to work properly [VGRH81]. It uses a complement of the original root event R' which describes the opposite of R (the fire protection system does not fail). The original FT is not inspected anymore, but the complemented tree, which is called the dual of the original FT. In this tree, all the OR gates are substituted by an AND gate and vice versa. Furthermore, the complement of the top event equals the combination of the complement of all minimal cut sets that were identified previously. This yields the following formulas:

- Complemented root event:

$R' = (A' + B') * C' * D'$ (Fire protection system does not fail)

- Distributive law:

$R' = A' * C' * D' + B' * C' * D'$

This time R' is the complement of the root event, and A', B', C', D' are the complements of the component failures. The set of all minimal path set is therefore $\{A' * C' * D', B' * C' * D'\}$. Relating this result to the example, one can make the statement that both components of the fire suppression system (C and D) must work, and at least one component of the fire detection system (A or B) must be functioning.

b) Quantitative evaluation: In the quantitative analysis, the probabilities of the failure events are inspected. The evaluation should yield the probability for the top event to fail. Vesely [VGRH81] introduces failure probability models that consider time dependence for constant failure rates per hour or constant failure rates per cycles. Since this would exceed the scope of this work, it is not focused further on but rather shown how to calculate the probability for the top event after the probabilities are applied to the basic events.

$P_i(t)$ is defined as the probability for cut set i at time t that leads to the failure of the system. Furthermore is $p_e(t)$ the probability of a component e to fail at time t. Again, the fire protection system is used as a basis for the following calculation. If it is assumed that component failures occur independent of each other one gets the following probabilities for each cut set:

$$P_{AB}(t) = p_A(t) * p_B(t)$$
$$P_C(t) = p_C(t)$$
$$P_D(t) = p_D(t)$$

In order to calculate the probability of the systems failure PS at time t one has to add up the probabilities of all cut sets:

$$PS(t) = P_{AB}(t) + P_C(t) + P_D(t)$$

Finally the importance and therefore the significance of a cut set and its components can be inspected by applying the following formula:

$$I_{AB}(t) = \frac{P_{AB}(t)}{PS(t)}$$
$$I_C(t) = \frac{P_C(t)}{PS(t)}$$
$$I_D(t) = \frac{P_D(t)}{PS(t)}$$

Here I yields the importance of a cut set by dividing its probability to fail at time t by the probability of the whole system to fail. Hence, the more significant the importance, the higher is the significance of the components of the cut set.

4. Assessment of reliability, improvements, and trade-offs: A final report of an FTA should contain a summary of the previous three steps based on the IEC standard [Com+90]. It should reflect on the system definition by discussing the boundaries that were chosen and the outcome of the qualitative and quantitative analysis. Since the latter is only defined loosely, there is no clear guide on doing this step by step. However, the results of the analysis, especially the importance score, should give the analyst enough data to evaluate system-critical components and make suggestions on the quality of the system as a whole.

Benefits	**Drawbacks**
Failures can be identified quickly through visualization.	Requires domain expert with deep knowledge on the system.
Outside components can be visualized in another diagram.	FT diagrams can grow quickly very large.
Cascading effects are directly visible.	No interplay between components is analyzed.
	Outcome relies on expertise of participants.
	No recommendation on safeguard mechanism has to be done.
	No explicit ranking of failures by severity exists.

Table 2.3: Benefits and drawbacks we identified of using FTA to analyze risks and hazards.

2.3.5 Failure Mode and Effect Analysis

FMEA is a bottom-up hazard analysis method that is done early in a system's design phase. It can be traced back to 1962 when it was mentioned under a different name by the National Space Agency [Nea62]. Later in 1980, it was standardized by the US military [Sta80] and has since been adopted by various areas of engineering like healthcare, nuclear power, or software [Ben09]. FMEA is a team-based analysis method that requires experts who know the inspected system.

In a group meeting, potential failure modes are brainstormed, and root causes are identified. A failure mode is defined as how a failure is observed, while the cause is defined as a defect that is the reason for failure [God00]. An indenture level describes the importance of a failure mode. A high indenture level represents a more complex component, while a low indenture level describes a more simple component. Furthermore, a failure mode can have an effect (consequence). A *local effect* has consequences within an inspected component. A *next higher level effect* affects components that are outside the inspected component. An *end effect* describes the consequences on a high indenture level [Sta80]. Described by Ben-Daya FMEA a failure mode is rated by three factors [Ben09]:

- Severity: the consequence of the failure when it happens
- Occurrence: the probability or frequency of the failure occurring
- Detection: the probability of the failure being detected before the impact of the effect is realized.

Multiplying these three factors defines the failure mode represented by the RPN. The higher the RPN, the more critical the failure is. When all potential failure modes are ranked by their respective RPNs a suggestion is made on how to reduce or eliminate the effect of a failure, starting with the most critical failure mode [Ben09]. The result of FMEA is documented in a predefined evaluation sheet.

2.3.6 System-Theoretic Process Analysis

In contrast to the previously introduced hazard analysis methods, the STPA is a relatively new technique developed by Nancy Leveson in 20011 [Lev11]. STPA is based on Systems-Theoretic Accident Model and Processes (STAMP) also developed by Leveson in 2004 [Lev04]. STAMP and STPA are designed to work for software environments and focus rather on functional models than physical ones. STAMP is a causality model that follows three basic concepts [Lev16]:

- Emergence and hierarchy: Components of a system can be split into fundamental properties that can not be reduced further (i.e., a function). Such a component is called an emergent property. Also, components may belong to different levels of a system hierarchy defined by the system's control processes of the system [Lev16].
- Communication and control: Components of different hierarchy levels require interfaces to communicate with each other. "STAMP uses the concept of imposing constraints in system behavior to avoid unsafe events or conditions rather than focusing on avoiding individual component failures" [Son12].
- Process models: A controller inspecting a process requires a model to make assumptions about the correctness of calculations. If the behavior of the process deviates from the model description, either the model is not correct, or the process is producing wrong results.

Based on the previously mentioned concepts of STAMP, STPA introduces a two-step process to identify hazards [Lev11; Lev16]:

1. Definition of Hazards and related Safety Constraints: The starting point is a scenario that describes a potential hazard and safety constraints. Levenson describes the following scenario:
   - *A person is present in the doorway when the door is closing* (Hazard).
   - *An elevator door must not close while anyone is in the doorway* (Constraint).

   Based on the scenario, the process continues with the following steps [Son12]:

   a) Develop Hierarchy Safety Control Structure: A model of the hierarchies in a system is defined. Components of this model communicate over channels and follow constraints.

   b) Identify potentially inadequate control actions: Define the inadequate actions (hazards) that violate a constraint and may lead the system into a hazardous state.
2. Occurrence of potential inadequate control actions: A hazard can have four different types of occurrences:

   a) Control input or external information is wrong or missing.

   b) Inadequate control algorithm produces wrong results.

c) Process model or sensor may be faulty from the beginning of the execution.

d) Actuators and controlled processes from the outside might not be delivered or do not exist.

STPA provides an advantage over other mentioned analysis tools. The model defined during the analysis process enables analysts to inspect it rather than filling a blank template. Furthermore, the interplay of components is highlighted in STPA in contrast to HAZOP or FTA. The define hierarchies provide further separation between less relevant and critical hazards.

2.3.7 Software Hazard Analysis and Resolution in Design

The Software Hazard Analysis and Resolution in Design (SHARD) analysis method was first introduced in 1994 [MP94] and named SHARD in 1995 by McDermid et al. [MNPF95]. SHARD is a variant of the HAZOP analysis method and solely used in software systems. The method is used to inspect the safety-related behavior of software components and is a top-down technique. SHARD arose from the question if HAZOP is transferable to other contexts than the chemical industry [Pum99]. Especially the use of guide words in a software context was considered to be ineffective [Pum99]. The process and necessary activities of the approach are depicted in Figure 2.6. The execution of SHARD is performed similarly to the one in HAZOP. It consists of a ten-step process with the following activities:

1. Understand the design
2. Select information flow
3. Describe flow and define intended behavior
4. Ensure intended operation is safe
5. Use a guide word to suggest a deviation
6. Investigate cause(s)
7. Investigate effect(s)
8. Examine detection, protection, and mitigation
9. Decide on acceptability and make a recommendation
10. Summarize the analysis

Steps 1–4 implicitly prove that SHARD requires domain experts to participate in the risk analysis process. As the technique follows the top-down approach, at least one analyst must be able to identify and understand individual components of the system. Figure 2.6 suggest that the SHARD approach is conducted with three independent teams that interact only through the results of their respective process steps. However, members of a specific team may take part in the activities of the others as well [MNPF95]. Steps 5–9 enclose the necessary tasks to create an entry in the SHARD table, which is part of the outcome. Each individual step yields the content of a column in the table. These processes are similar to the one necessary in HAZOP. Step 10 is used to transcribe the results

of the previous steps into a final report document. As recommendations proposed in step 8 may not prove to be suitable or unsafe, it is possible that the processes from steps 1–10 may be repeated. However, the documentation of previous results should not be updated [Pum99].

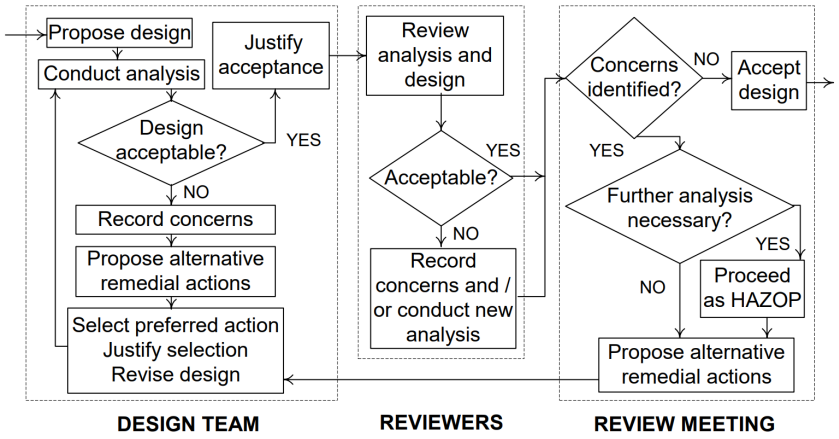


Figure 2.6: Outline of the SHARD analysis as presented by Pumfrey [Pum99].

The application of SHARD is similar to that of HAZOP as it uses a variation of the HAZOP of guide words. However, the method provides only a fixed set of guide words [DGH+14]. Each guide word contains one or multiple subcategories that refine the description of a system deviation. The most prominent guide words of SHARD are *Omission*, *Commission*, *Early*, *Late*, and *Value* which are comparable to the guide words of the HAZOP approach. Refinement of the *Omission* include *partial omission* or *total omission*. The refinement can also contain a temporal component that describes the iterations of a guide word. Examples for iterations are *single*, *repeated*, *periodic*. A complete example of an actual SHARD hazard would be *Repeated, partial omission of service X* which is much much accurate than *NO service X* in the description format of HAZOP. It should be noted that some refinement categories are only applicable for a specific guide word. It is also possible that a guide word is used for risks that carry no meaningful interpretation [Pum99]. One example is a hardware interrupt that represents a stimulus that has no effect. Such a case is recorded with an empty entry (NA/-) in the SHARD table as described in Table 2.4.

As the analysis method is a top-down technique, SHARD is meant to be used to design safety-related issues rather than the safety analysis of a software system during runtime [MNPF95]. Therefore, it can not prove formulas or the probability of a failure like FTA.

2.3.8 Low-level Interaction Safety Analysis

Low-level Interaction Safety Analysis (LISA) was developed in 1998 for a military avionics system [Pum99]. LISA inspects low-level hardware deviations and events in the interplay with software programs of different integrity levels [CM01].

LISA was first used in a case study to inspect a cockpit display system. Previous testing with different analysis methods showed critical as well as non-critical hazards could occur in the system. In some cases, the cockpit display showed deviations for drawn symbols which could lead to hazardous situations in the real world [Pum99].

Guideword	Omission	Commission	Early	Late	…
Deviation	No output records sent.	Additional output records sent.	-	Output records sent late	…
Cause(s)	Failure of OUT process or failure of CAN bus	Spurious CAN messages.	-	Failure of scheduling software or CAN bus overloading	…
Co−effector(s)	-	-	-	-	…
Detection	None possible	Detect out-of-bounds sequence ID	-	-	…
Effects	a,b,d,e	b,c,d,f	-	b,e	…
Recommendation	Ensure reliability of hardware and implementation of bus. Must at least 1 order of magnitude better than hydraulic components of the system.	Check integrity of message ID generation in OUT process. Check CAN protocol with CRC check.	No meaning	Check integrity of synchronisation. Check causes of CAN bus saturation	…

Table 2.4: A SHARD analysis table for a computer-assisted braking system [Pum99]. Effects a–f are defined outside the scope of the analysis.

The purpose of LISA is to create a partitioning for hardware and software components of the system and produce arguments of acceptability for projected failures [MP00]. The results should allow the execution of processes from different integrity levels to run on the same hardware [WJG01]. The analysis method is structured as a four-step process:

1. Agree principles for acceptability
2. Assemble source material
3. Analyze timing events
4. Analyze physical resources

The first step is used to determine the optimal state of the inspected system. The definition of acceptability criteria is system dependent [Pum99]. Examples for acceptability criteria are “There is no cause for the suggested failure”, “The failure does not affect the correct operation of critical functions”, and “Low-integrity processes can not alter high-integrity processes” [Pum99]. Before the definition of acceptability criteria, the components of the system have to be classified into integrity levels. Each component is assigned with an integrity level, with *Class 1* being the highest. Assigning the integrity levels is dependant on the inspected software. However, the number of levels

should be kept to a minimum [Pum99]. Since this is not an official process step in the LISA method, it is expected that the components were assigned the integrity levels during the design of the system. Examples of a *Class 1* component is the operating system or the kernel, while middleware is, for example, assigned as a *Class 2* component.

In the second step, analysts collect documentation material about the inspected system. On the one hand, this material contains high-level design descriptions, diagrams, and specification documents about the software. On the other hand, technical details are gathered, such as the use of system resources (i.e., CPU registers, I/O devices), or documentation manuals [MP00]. As in the first step, this procedure depends on the complexity of the inspected system. Based on the results collected in the second step, the third step is initiated.

In the third step, timing events such as hardware interrupts, system timers, inter-process communication, and communication with other devices or systems are inspected. For each occurrence of a timing event, the state of the system has to be documented to define the respective categories such as startup, regular operation, or shutdown [MP00]. When all unique timing events are recognized, a failure mode analysis based on HAZOP and SHARD is applied. Since the definition of deviations, causes, and effects works the same way as in SHARD it is not explained in further detail here.

The last step of the LISA procedure is concerned with the classification of hardware resources and definitions of deviations. First, hardware components that share commonalities are grouped together. Examples are memory pages, processor subsystems, and registers. Secondly, these groups are given a classification. The available classes are *intrinsically critical*, *primary control*, *secondary control*, *non-critical*, and *unused* [MP00]. After the classification of the hardware groups, identifying deviations, causes, and effects as in the third step is repeated for every group. Finally, the acceptance criteria defined in the first step of the procedure are inspected if they cover the deviations of the hardware groups. LISA expects analysts to document their findings for each individual step. However, no official technique is proposed.

	Organisational						Methodical						
Hazard Method	In-person group meeting	Expert presence required	Documentation required	Final document	Complexity/Effort	During what lifecycle	Domain of origin	Iterative	Textual or visual approach	Qualitative	Quantitative	Hierarchy of failures	Ranking of failures
HAZOP	✓	✓	~	✓	**H**	→|	Chemic/Atomic Industry	~	**T**	✓	✗	✗	✗
FTA	✓	✓	✗	✗	**M**	→|, |→|, |→	Aerospace	✗	**V**	✓	✓	✗	~
FMEA	✓	✓	~	✓	**L**	|→	Aerospace	~	**T**	✓	✗	✗	✗
STPA	✓	✓	~	✓	**M**	|→	Software Engineering	~	**T+V**	✓	✗	✗	✗
SHARD	✓	✓	✓	✓	**H**	→|	Software Engineering	✓	**T**	✓	-	-	-
LISA	✓	✓	~	~	**H**	|→|, |→	Military Industry	✗	**T**	-	✓	-	-

✓ = provides/requires the property; ✗ = does not provide/require the property; ~ = partially provides/requires property; - = no statment can be made;

T = textual approach; **V** = visual approach; **T+V** = combination of textual and visual approach

→| = before development; |→| = during development; |→ = after development

L= low effort; **M**= medium effort; **H**= high effort

Table 2.5: The properties we identified during inspection of the risk and hazard analysis methods. Findings are based on the official documentation of the methods. Application of these methods in practice may deviate from the proposed design.

Table 2.5 concludes the findings of hazard and risk analysis techniques used by different industries. Based on the information contained in this table, it was decided to use the HAZOP method for the analysis in the prototype. This is emphasized further in Chapter 4.

2.4 Tooling for System Description

In the following, technologies for resilience engineering are introduced. The focus lies on tracing tools, which provides the technical basis for this work. In Section 2.4.1 the tracing technology is outlined. Its use in distributed environments and performance testing are given. Section 2.4.2, Section 2.4.3, and Section 2.4.4 describe the functionality of the tracing tools used in this work.

2.4.1 Tracing

Tracing is a specialized method of recording log messages of a computer program. The purpose of tracing is to diagnose applications, find weaknesses, and optimize performance bottlenecks [BCD+06]. It can help discover the statement, branch, or path coverage and therefore be used in testing as well [BL94]. Tracing is also used for debugging and can be compared to profiling as both techniques compute a spanning tree of the inspected program, either explicitly or implicitly [BL94]. Other use cases in distributed tracing are resource attribution, anomaly detection, and workload modeling [SFSG14]. Traces can be extracted from a program either during development by a developer, at compile-time through a compiler, or dynamically at runtime through binary translation [BCD+06]. In general, tracing can be used when the application shares at least some commonality, such as the execution environment or application stack. Distributed applications, which do not share common hardware, runtime environment, or operate on different protocol layers, have to be traced with non-conventional tracing tools such as X-Trace [FPKS07].

Two general approaches for tracing exist. On the one hand, there is black box-based tracing. It is used when components of a program are unavailable as source code or part of vendor-specific libraries at remote locations [AMW+03]. Black box tracing comes at the cost of accuracy but provides a boost in performance. On the other hand, there is annotation-based tracing. It instruments a program with instructions at specific code locations. Although annotation-based tracing might reduce the performance through the observer effect of the inspected program, it delivers more accurate results [SNZP16]. The impact of the observer effect can be reduced by the introduction of sampling, where traces are recorded only at specific intervals. However, this comes also at the cost of accuracy. Another influencing factor for annotation-based tracing is a biased engineer who implements traces at locations of his interest [Lar90]. Besides performance overhead, tracing sizes pose another challenge. As profiling applications may collect more than a million traces, the allocation and storage of the recorded data require sophisticated solutions [Lar90]. Here another trade-off decision has to be made between accuracy and performance.

A trace represents the course of a function propagating through a system containing recursive calls or calls to other nested functions [BL94]. Contents of a trace are usually the name of the operation or the location in the source code, time spent in this operation, hierarchical relationships to parent operations, child operations, or other traces [Lar90]. Since no standard exists for the structure of a

trace, information may be vendor-dependent. In distributed tracing, the notion of a span has been established, which originates from Dapper (introduced in Section 2.4.2) and describes a function, or more general, an operation [SBB+10].

Besides performance, Oliner et al. [OGX11] have identified visualization as one major challenge of log analysis and tracing. The logical structure of a trace is represented as a DAG where each operation is a node that has a hierarchical relationship with other nodes through a directed edge. However, there exist several approaches to visualize a DAG [SFSG14]. Swimlanes, better known as Gantt charts, focus on the time dependency of individual traces and highlight the sequential and concurrent workflows of operations. Flow graphs aggregate calls that share part of the same workflow. Call graphs which are usually used in traditional profiling, show the call stack of a trace and are less precise since they neglect forks and joins of concurrent calls [SFSG14]. Calling trees and context trees combine the execution profiles of multiple operations into one tree structure that represents all possible execution paths.

Open tracing projects exist that try to unify trace formats or provide an Application Program Interface (API) for tracing tools. Instrumentation libraries of these projects implement design patterns that allow developers to select a specific tracing strategy while still collecting data in a standard format.

2.4.2 Dapper

Dapper, developed in 2008 and presented in 2010, has been developed from the need to have a low overhead distributed tracing software framework [SBB+10]. Since its introduction, it has influenced other well-known tracing solutions like Zipkin (Section 2.4.3) and Jaeger (Section 2.4.4) that show similarities in design and execution. With the upcoming concept of distributed architectures and the idea of microservices in the early 2000s, Google required more insights about their distributed applications and better insight for root cause analysis [BDH03]. Tracing (Section 2.4.1) had been established before 2000, yet it only worked for applications running on a single machine. Solutions for distributed tracing did exist before Dapper but lacked in high-performance and application transparency [SBB+10]. Therefore, the authors of Dapper set (i) low overhead, (ii) application transparency, and (iii) scalability as their primary objectives.

The idea behind Dapper is to help engineers at Google detect root causes and pinpoint a failure to a particular component of a distributed application. Distributed applications are usually implemented isolated, on hardware at different locations, and oftentimes in multiple programming languages. An engineer might be unfamiliar with the infrastructure of the application or with certain details of a specific service. Tracing helps to log execution times and additional metrics of nested function calls to create a chain of events that can be easily visualized and inspected. In favor of more accurate results, Dapper implements annotation-based tracing through instrumentation. In contrast to black-box tracing, it yields more application transparency [SBB+10]. This means that the software that is inspected has to add instructions at every location of interest. This is enabled through programming language-specific instrumentation libraries.

A chain of events that represent the nested function calls is described as a tree by Dapper. Every node in such a tree represents a basic unit of work and is known as a span. A span can have a start- and stop-time – representing the beginning and end of the execution, a unique id, and a parent span. A span without a parent span is known as the root span.

Instrumentation libraries exist for C++ and Java. The implementations of these libraries have less than 2000 lines of code, resulting in a small overhead from linked binaries for the instrumented application. The language-specific instrumentation libraries do all support the same concepts. Programs that follow a thread-based implementation manage nested calls with trace context stored to the thread-local storage. Programs that follow an event-based asynchronous implementation store the trace context to event callbacks. Such a trace context is implemented with low resources and is easily copyable, thus yielding better performance [SBB+10]. Application of Dapper in practice showed a runtime overhead of less than one % [SFSG14]. Instrumentation libraries enable developers to annotate traces with custom key-value pair entries. Examples are the previously mentioned service names and the unique id of a span. In practice, annotations have a maximum limit to block significant amounts of data [SBB+10]. Furthermore, the libraries provide adaptive sampling to avoid the so-called aggressive sampling of many log messages over a short period of time.

Traces are stored in collectors that represent a database on disk storage. The underlying model for these databases in Dapper is Google's Bigtable, a wide-column storage format [CDG+08]. Each row stores exactly one trace, where each column stores the data of one span. Since each span consumes less than one kilobyte of storage, a trace report can be assembled in less than a minute for 98 % of all cases [SBB+10].

A trace report of Dapper is visualized in the Dapper API (DAPI) that provides the user with a selection of time ranges and service names. Based on the search criteria, the DAPI delivers a list of traces matching the time range and the name of the service. If a detailed summary of a trace is desired, the DAPI can show the user a call tree of the nested function calls, time spent in a specific function, and a diagram of a cost metric (i.e., request size, or recursive queue time).

2.4.3 Zipkin

Zipkin is a low-overhead, open-source, distributed tracing framework developed in Java which is based on Google Dapper (Section 2.4.2) and was initially developed by Twitter [Twi12]. It allows the instrumentation of a targeted software system and is specialized in analyzing distributed systems and therefore suitable for inspection of MSAs. Microservices only communicate over API calls via Hypertext Transfer Protocol (HTTP). Therefore, the Zipkin instrumentation focuses exclusively on HTTP requests.

For the Zipkin approach to work, each microservice is instrumented with the Zipkin library that wraps the services' HTTP requests with additional HTTP headers that store unique identifiers, timestamps, and other metrics that necessary to construct a trace [Zip12]. The HTTP headers are updated whenever a microservice issues an outgoing request or receives an incoming request. Besides the original receiver, the request is sent to a Zipkin reporter. This reporter forwards the modified HTTP headers to the Zipkin collector that stores the request in storage for later usage and

filtering. Zipkin supports Apache Cassandra[3], ElasticSearch[4], and MySQL[5] as storage models. A JavaScript Object Notation (JSON) API does exist to extract individual traces and related information from these databases.

Zipkin is available as a standalone version and supports containerization with Docker[6]. Once it is built and started, Zipkin provides a GUI that is accessible via a web User Interface (UI) that visualizes the hierarchy of traces and related metrics such as execution times of spans and collected annotations that were collected during execution of the instrumented application. Besides, it allows engineers to inspect a dependency graph that shows the relationship between operations. If developers of a microservice-based system are concerned with an unusual longer duration of a request, they can inspect a Zipkin trace and see where most of the time was spent. They can pin down the problematic service.

Zipkin is now a community effort and developed open-source since its' source code has been presented to the public. Besides the officially supported storage models, community supported ones exist as well (i.e. Apache Kafka[7], Logz.io[8], or Scouter APM[9]). The same holds for instrumentation libraries. The officially supported programming languages are C#, Go, Java, JavaScript, Ruby, Scala, and PHP. The community-supported programming languages extend this list with C, C++, Python, Lua, and Scala.

2.4.4 Jaeger

Jaeger is a Zipkin-based distributed tracing framework that emerged from Uber in 2015 [Jae15] and was declared an open-source project in 2017 [Shk15]. The framework originated from the need to monitor and analyze microservice applications at Uber. It is based on the Zipkin tracing tool (Section 2.4.3). Early implementations of Jaeger used components of Zipkin such as the UI because engineers were already familiar with Zipkin [Shk15]. As a result, Jaeger was compatible with Zipkin and still provides backward compatibility to it.

Features from Zipkin are extended and refined by Jaeger. First, trace collectors can be enabled to use request multiplication during a trace recording. This allows for the creation of stress tests and emulation of user behavior during development. Secondly, Jaeger provides three different sampling strategies for trace collection [Shk15]:

1. Full sampling, which collects every trace.
2. Probabilistic sampling, which collects traces at random but with a fixed probability.
3. Limited sampling, which collects traces at a fixed time interval.

[3] https://cassandra.apache.org/
[4] https://www.elastic.co/
[5] https://www.mysql.com/
[6] https://www.docker.com/
[7] https://kafka.apache.org/
[8] https://logz.io/
[9] https://github.com/scouter-project/scouter

These sampling strategies can be applied dynamically based on the load of microservices to avoid excessive trace collection. Thirdly, the Zipkin-based UI was redesigned. In general, the UI provides the same amount of features as Zipkin. However, while maintaining simplicity, it offers more insights to trace contents and is structured more straightforward to use.

2.5 Formalization

As a basis for the formalization of scenarios, Linear Temporal Logic (LTL) and frameworks based on it are used. LTL provides means to define the logical condition in time-dependent sequences. In this work, formalization helps to define a clear and unambiguous reasoning model for the specification of a system.

2.5.1 Linear Temporal Logic

LTL was introduced as a unified approach for program verification by Amir Pnueli in 1977 [Pnu77]. It is based on predicate logic and provides means to describe a sequence of events and the eventuality of their occurrences. The syntax uses a subset of logic operators and defines temporal operators to express dependence between two or more separate events. The initial use of LTL was to perform proofs in model checking [CES86] but is also used in proofs of concurrent programs and description of invariances [Pnu77].

Computation Tree Logic (CTL) extends LTL by introducing a branching model [CE81]. Rather than LTL, it explores multiple outcomes on different timelines, which means that non-deterministic executions are inspected as separate sequences. The disparity of states at different times is modeled with a tree structure. CTL extends the syntax of LTL by further logical operators and reduces the effort of writing verification formulas for some constructs.

In contrast to CTL, MTL is concerned with time-constrained events [Koy90]. MTL is another extension of LTL also concerned with proofs of model checking however only for timed system. A timed system introduces variables to describe a systems state at a certain point in time or for a time span between two states [AH91].

LTL and its successors are used to describe the contents of resilience scenarios in a formal manner. More about the MTL method is presented in Chapter 4.

2.5.2 Property Specification Patterns

Dwyer et al. [DAC99] introduced 1999 Property Specification Patterns (PSPs). The specification patterns are described as a necessity to solve reoccurring problems in verification and are expressed in a structured format. There are three separate pattern categories (occurrence, order, and compound) that are further divided into more specific categories. Each category covers a different modality of an event. The benefit of using PSPs is to help engineers to identify crucial properties by applying patterns to a system [DAC99].

The approach of PSPs was extended by the authors of the Property Specification Framework (PSF) [AGL+15] which allows for a more detailed description of patterns in more fine-grained categories. Furthermore, the authors introduce the PSPWizard, a user interface to automate the creation of LTL formulas based on a natural language catalog. This way, template-based English sentences, containing invariances for software functions, can be transformed into formulas of various temporal logic formats and verified more efficiently.

2.6 Conversational Interfaces

In Section 2.6.1 a short overview about Natural Language Processing (NLP), text recognition, and Natural Language Generation (NLG) is given. Section 2.6.2 introduces Dialogflow, a technology utilizing both NLP and NLG, to provide a chatbot interface for natural conversations.

2.6.1 Natural Language Processing and Generation

Modern NLP methods are based on neural networks to understand and extract valuable information from textual interactions between humans and computers [Gol16]. As natural language is based on grammatical rules, a NLP framework uses parsing to extract building blocks from sentences and classify the contents [NOC11]. Categorized excerpts of a sentence can be used to attach meaning to a sentence through additional resources such as databases and knowledge storages. Difficulties in NLP lie in the recognition of informal sentences that usually reflect everyday spoken language. In these cases, it is crucial to provide context to the conversation to connect sentences during a conversation [NOC11]. Modern NLP methods rely heavily on neural networks, which are network constructs that can be trained based on mathematical formulas through example sentences. However, trained neural networks provide only functionality for domain-specific conversations and can usually not be used in a general purpose context [Gol17].

Natural Language Generation is a sub-field of artificial intelligence and linguistics concerned with building computer systems that can create meaningful and grammatically correct texts, preferably in human languages [Hor01]. In contrast to NLP, NLG system use rules of a human language to create texts. These can be used to support documents with annotations, reports, and messages. Created texts can be used to highlight the significance of images or tables. The field of NLG has become popular, especially in combination with the embedding of graphical elements into texts [Hor01]. Reiter and Dale [RD97] describe how human-computer interaction and cognitive science have influenced the field of NLG regarding the information degree in texts. This shows that not only correct sentences are meaningful but the quality and the richness of the contained information as well.

2.6.2 Dialogflow

Dialogflow is a natural language processing framework developed initially by Speaktoit (new called Dialogflow[10]) and currently developed and maintained by Google [SA20]. The main focus of Dialogflow is to provide users with an assistant application that can understand intents from text messages or voice commands.

Integrations built into Dialogflow can be used as API for other applications to use. This includes social media platforms such as Facebook or instant messaging services like Telegram. With the current boom of home automation and smart devices, Dialogflow has also seen a thrust in popularity [Lee18]. Since the beginning of 2020, Dialogflow has introduced Dialogflow CX beta, which is meant to be used for complex conversations. Dialogflow ES supports the design of more straightforward conversations yet offers the same functionality as Dialogflow CX [SA20].

The basic concepts of Dialogflow are summarized as follows based on the official documentation [Goo16]: An *Agent* serves as the central node for the language platform. On the one hand, it is used to detect the intents of the users. On the other hand, it instantiates clients for intents, sessions, and contexts, which are used to handle traffic of their respective categories. An *Entity* is the basic building block of an *Intent* and can be defined as fixed text or regular expressions. An example for an *Entity* is the name of a car manufacturer. *Entities* are used in training sentences (i.e. *What car models does Porsche sell?*) that help the *Agent* to match the user text against an intent. If an *Intent* is matched, the agent returns the information about it to the backend. Within this response, a list of *Contexts* is contained. A *Context* can be used to steer a conversation between two or more intents. The output *Context* of one intent can be the input *Context* of another one. A *Context* contains the *Entities* and respective types of the *Entity* as a key-value pair. Finally, an *Event* is used to trigger an intent from outside the conversation. This can be realized by the user input, webhooks, or third-party services.

[10] https://cloud.google.com/dialogflow

3 Related Work

This chapter establishes related works. In Section 3.1, interactive solutions for requirements elicitation are presented. Most of these interactive solutions aim to simplify the requirements process, especially for inexperienced requirements engineers. All of the presented approaches use conversational interfaces to ease accessibility. Therefore, a big focus lies in text recognition and classification of text input. Section 3.2 focuses on approaches that use structured requirements elicitation methods. In contrast to the interactive solutions, the focus lies on precise specifications. Both of the presented methods in this section present different methodologies to achieve their goal.

We think that the objectives of the interactive solutions and the ones that aim for preciseness benefit our work. Advantages and drawbacks for each approach are highlighted, and it is explained how we want to realize extensions or improvements to unify both objectives.

For related works, we used the same search strategy as in Chapter 2. If supervisors provided related works, we looked for primary sources and searched with acronyms of the tools (i.e., CORDULA) combined with the relevant research area (interactive elicitation). Such a search always yielded only one clear result. Other sources were found through searches on Google Scholar. For the chatbot-based interactive solutions, we used the keywords *requirements*, *elicitation*, *interactive*, and *chatbot*. Furthermore, we inspected references in the related works section of found sources.

3.1 Chatbot-assisted Elicitation

In the following, solutions related to this work are presented. The focus of these tools lies in interactive requirements elicitation through the use of chatbots. Section 3.1.1 presents a chatbot utilizing the laddering approach. In Section 3.1.2 CORDULA is introduced. CORDULA provides interactive requirements elicitation through a chatbot in combination with manual improvement in a user interface. Section 3.1.3 and Section 3.1.4 chatbots with similar techniques. Surana et al. [SGS+19] use a less formalized requirement approach, while Arruda et al. [AMSW19] utilize the KAOS methodology. Section 3.1.5 describes the function of SOCIO, a chatbot for requirements elicitation in architecture meta-models. In Section 3.1.6 a Discord chatbot is presented that helps the community of the Pharos programming language find experts for complex questions.

3.1.1 Designing a Conversational Requirements Elicitation System for End-users

Rietz outlines in his research that there is a strong need for novel elicitation approaches [Rie19]. Current techniques, such as the traditional elicitation interview style, are challenging to apply in practice. They impose a high amount of time on participants, require locations to meet, and the presence of expert analysts. However, elicitation meetings are the most effective method to

perform requirements engineering [DJ10]. Rietz proposes a solution for all three problems in the form of a conversational interface. Chatbots can support novice end-users in requirements engineering since they interact in natural language with the users. Furthermore, they can assists inexperienced and expert requirements engineers by answering questions and providing further help with previously aggregated data. Chatbots, also remove the need for in-person meetings and lengthy meetings [Rie19].

Besides the chatbot, Rietz introduces the laddering technique, a structured approach for collecting data from a human subject [RM95]. In a laddering interview, the subject is first asked abstract, high-level questions. Based on the answers the subject gives, more precise questions can be asked that. In return, this yields to more precise answers by the subject. Examples of laddering questions are, *Why do you like X?*, *What benefits gives X to you?*, *What drawbacks brings the absence of X to you?*. Laddering aims to create a knowledge base that is represented in a tree structure, where abstract questions are on the root of the tree, and more specific details are shown on the leaves. A path in this tree is called an attribute, consequences, value (ACV) chain [VIC06]. The LadderBot developed by Rietz can ask ladder questions by mimicking the elicitation interview style. Besides the chat window, the prototype consists of a second interface that displays interactive ACV chains that let users inspect the created requirements. The presented approach is shown in Figure 3.1.

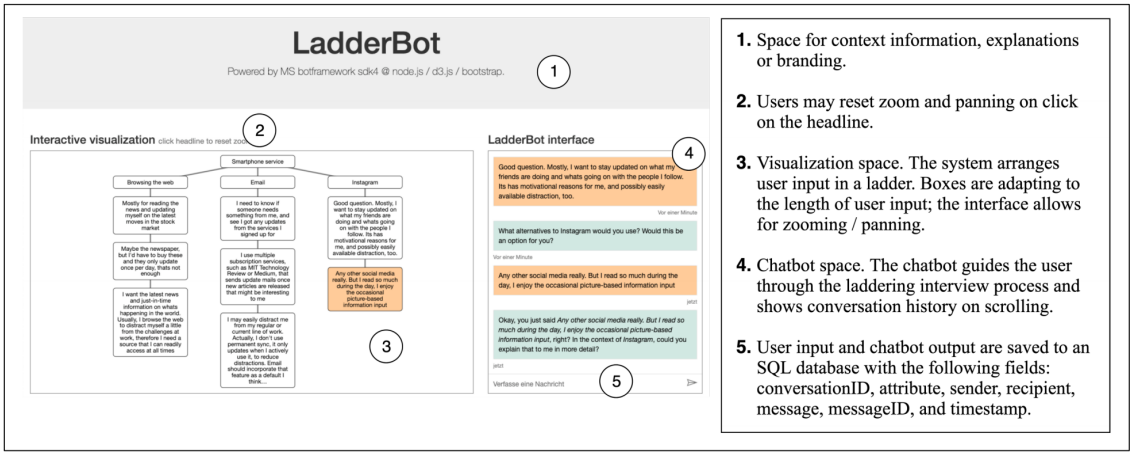


Figure 3.1: Screenshot of the presented approach by Rietz [Rie19].

The utilization of the laddering approach provides at least two benefits to the LadderBot. On the one hand, the information given by the user can be quickly processed and stored in a knowledge base. On the other hand, the elicitation process feels natural to the users. However, it is questionable if the presented approach can cover requirements aspects present in a complex software system and how exactly a laddering interview could be performed in such a scenario. Furthermore, the presence of a human expert is still required in the conversation with the chatbot since the bot can not determine when the end of an ACV chain is reached. In addition, the presented approach does not seem to use a formalization scheme for data collected during the laddering interview. In contrast to the presented laddering approach we will use the ATAM-based resilience scenarios to collect requirements about software quality attributes.

3.1.2 CORDULA

The CORDULA approach tries to involve stakeholders actively in the requirements elicitation process. Friesen et al. [FBG18] let users of CORDULA specify their expectations of a software system in natural language. In contrast to other approaches, they identify vagueness in the user's intent since conversational interfaces offer the user some freedom to specify imprecise terms. CORDULA's user interface is composed of two views as depicted in Figure 3.2. On the one hand, is the interactive chat window, and on the other hand is the debugging window. In the chat window, users can manually configure the specification via text input. The chatbot identifies entities contained in the text and shows them to the users. They can then confirm or decline their specification. Imprecise specifications such as *a big file*, *reliable communication* are recognized by CORDULA. Such vague terms are stored in a knowledge base created by the authors. CORDULA can match user input against words and phrases stored in the knowledge base. Vague terms are highlighted in the UI with warning messages. It is immediately visible to the user if a given specification is misleading or incomplete. In the debugging view, the specification accepted by the users is shown. They can then manually modify the specification or delete those that are not of interest anymore.

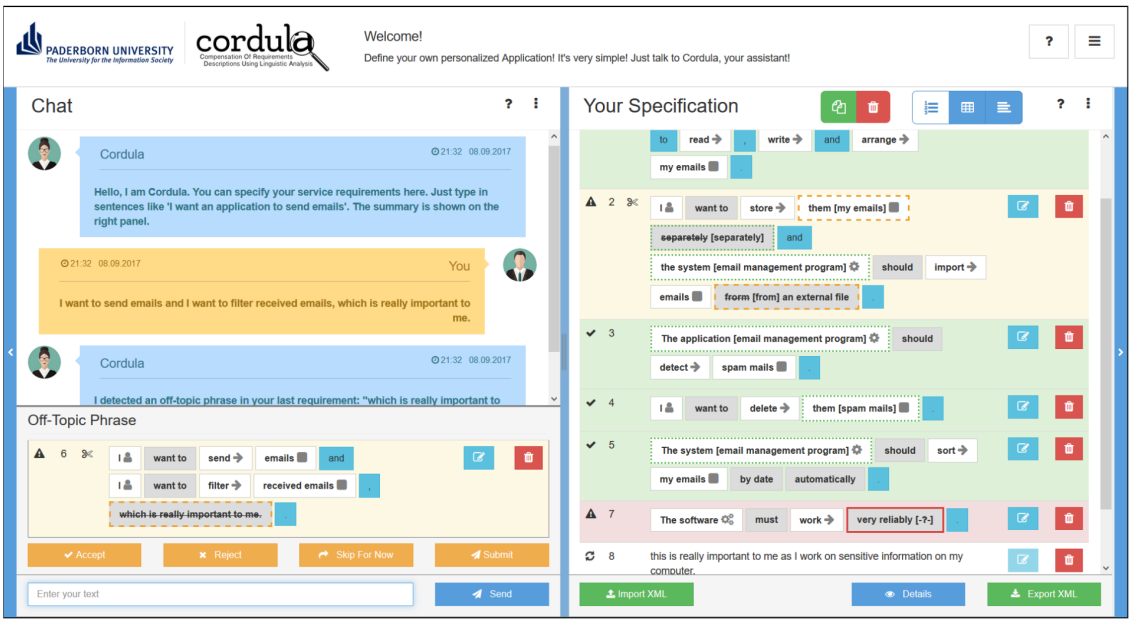


Figure 3.2: Screenshot of the CORDULA prototype [FBG18].

CORDULA requires users to write chat messages for specifications in a particular format, limiting the usage only to experts. Non-experienced users are probably not aware of specific sentence structures that are required to create a specification. However, the authors are aware of this drawback and write that they try to improve the tool further to make it more usable to novices. The prototype brings the benefit of interaction while still having the opportunity to modify elicited requirements. Our approach aims to remove the need for users to specify requirements in a specific format. Furthermore, we let the requirements analyst focus on the specification quality attributes of the inspected system rather than managing the elicited requirements.

3.1.3 Intelligent Chatbot for Requirements Elicitation and Classification

Surana et al. [SGS+19] present a chatbot for requirements elicitation to reduce the human effort necessary to create requirements. The approach uses a combination of Natural Language Understanding (NLU) and NLP to create interactions between stakeholders and a chatbot. The design shown in Figure 3.3 uses Qt[1] for the user interface, RasaNLU[2] for NLU and dialogue management, and spaCy[3] for the English language model to understand stakeholder intents and create text classifications. Similar to Dialogflow (introduced in Section 2.6.2), pattern-based training sentences are given to the language model. Based on the training sentences, user intents can be matched and entities extracted. Missing entities necessary to describe requirements can therefore be prompted to the user. The chatbot makes the distinction between functional and non-functional requirements extracted from the conversation.

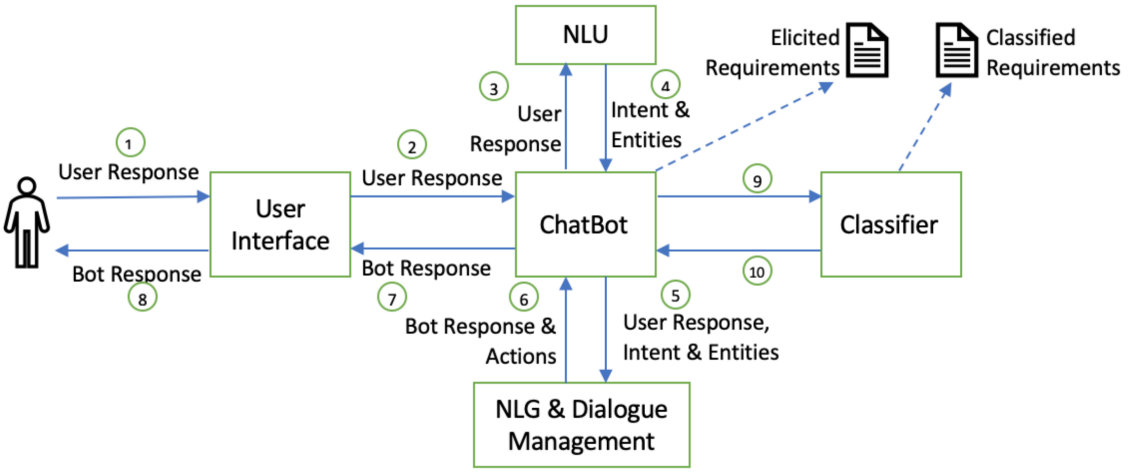


Figure 3.3: System design of the presented approach by Surana et al. [SGS+19].

The approach presented by Surana et al. [SGS+19] gives a clear division of concerns in the system design. However, they do not provide insights into the system that is subject to the elicitation process. A stakeholder must be familiar with the system and can use the chatbot only to write down requirements that were previously defined. The distinction between functional and non-functional requirements is helpful to the stakeholders, yet the distinction is based on fixed training phrases by the authors. Furthermore, requirements do not follow a formal method to save and measure metrics of the system.

3.1.4 KAOSbot

The KAOS framework [EGA13] is a Goal-Oriented Requirements Engineering (GORE) approach. GORE is used to define a goal and subgoals among stakeholders of complex, large-scale software systems [WOP09]. There is a primary goal in GORE that is decomposed into smaller subgoals,

[1] https://www.qt.io/
[2] https://rasa.com/
[3] https://spacy.io/

sometimes called soft goals. Based on the complexity of the inspected system, more layers of abstraction are needed to describe the central goal in more concrete subgoals. The KAOS framework is used to describe concrete subgoals of GORE with semi-formal and formal methods [EGA13]. Examples of definitions for a concrete subgoal are Object Constraint Language (OCL) or Goal-Question-Metric (GQM). Earlier works of Darimont et al. [DDML97] already implemented a goal-driven KAOS tool with a graphical user interface. However, the development was discontinued.

Arruda et al. [AMSW19] propose the KAOSbot to elicit, document, and modify goals and subgoals of the GORE approach. The KAOSbot utilizes KAOS framework in a natural conversation with a stakeholder of a software [AMSW19]. The focus of the KAOSbot is on less experienced professionals and novices in the area of requirements elicitation. The reasoning behind using the GORE approach is that it is widely used in software requirements projects [EGA11]. The chatbot itself is decomposed into two components. The first component is the user interface that presents a user with a chat window, a modeling area, and a legend. The second component is the bot service that extracts a user intent with NLP technologies. A user can interact with the bot by writing text into a text input in the chat window. In response, the chatbot answers with questions to refine the KAOS model or asking for further configurations. For the bot service, coreNLP is used to detect intents from the users in written English sentences. The identification of main goals, subgoals, and refinement instructions is using a tree-based approach. This approach detects different types of sentences (i.e., Predicate-Object-Prepositional Phrases, Subject-Predicate-Object-Prepositional Phrases) and determines the entities of the sentence based on a pattern-based algorithm. In an experiment where students acted as inexperienced requirements engineers, the ease of use, usability, and the potential future use of the KAOSbot was evaluated.

The authors mention that the user can modify the KAOS model in the UI during a conversation with the chatbot. However, it is not clear to the reader how this is realized, and how complex the inspected system can be. No used technologies, implementation details, and graph creation methods are mentioned. The pattern-based algorithm, based on the Stanford coreNLP framework, complicates the extraction of user intents. Sentences used in conversations with the bot have to match the expected quality to extract entities of user sentences. In case of misunderstandings with the KAOSbot, neither fallback mechanisms nor help methodologies to support inexperienced users are described. Furthermore, the readers are not given details about the tasks used during the study, which makes it hard to verify the tool's usefulness.

3.1.5 SOCIO

SOCIO is a modeling chatbot that can create architecture meta-models based on users' intents in a conversation with messaging services like Twitter and Telegram [PGL17]. SOCIO intends to simplify collaboration in modeling among multiple parties and the acceleration of this process. A user of SOCIO can interact with the chatbot via commands (i.e., /newmodel, /history) of the corresponding platform. Commands range from the creation of a new meta-model to showing the latest changes made to the meta-model. A user has to provide additional parameters to apply changes to the meta-model. The chatbot can identify these and perform the actions in the background. The Stanford coreNLP is used in the approach to identify natural language sentences and extract relevant nouns, verbs, and adjectives. The found entities are used in NLP rules where they are mapped against actions that define the modification of a meta-model. SOCIO supports the storage of traces that keep track of changes to the meta-model. Changes can be queried in the conversation with

the bot. The bot can also validate a meta-model. Response to the user, contain visualizations of the meta-model. The latest changes are highlighted in the visualization. Examples for features that should be added to a meta-model are *the age of a Person is an integer*, *Person is abstract*, *every Person must write at least one Test*. The first example adds a variable of type integer to the class Person. The second example defines the class Person as an abstract class. The third example creates a relation between the class Person and the class Test with the cardinality one-to-many. More implementation details on the function of SOCIO can be found on the official development website[4].

With the use of the Twitter platform, creations and modifications done to a model are visible to anybody. Furthermore, anybody can modify the meta-model. However, the use of the Telegram messaging service mitigates the risk of outside changes with a user access control. As the approach uses the coreNLP framework, sentences sent to the bot must contain a specific structure that has to contain all the necessary parameters to execute the action in the meta-model. We think that SOCIO brings the benefit of immediately visualizing user input in a meta-model architecture. However, the use of messenger-like services seems to complicate the interaction with the architecture. In our approach, we will not visualize an architecture meta-model but use the content of traces to represent the architecture. We will offer users of our approach the possibility to interact and inspect the architecture representation.

3.1.6 Building an Expert Recommender Chatbot

Developed for the Pharo programming language[5], Cerezo et al. [CKRB19] present a recommender chatbot integrated into the Discord[6] chat service. The chatbot is meant to support developers in finding a resolution for software development issues. The bot offers developer expertise which is realized via text mining of historical data from commits to the Pharo source codebase. Commit messages are parsed for term frequency and the author names. The bot's knowledge is classified into several categories that cover a specific aspect of the Pharos programming language. A question to the chatbot (i.e., *Who knows about object-oriented programming?*) delivers a list of expert developers that are most suitable to answer the asked question.

Although the approach of Cerezo et al. [CKRB19] is not concerned with requirements elicitation, it offers insight on how to create helpful chatbot responses to users with a question on a particular topic. However, the classification of categories based on the words contained in a commit message might not be accurate enough to answer more complex questions. We will use a knowledge base in our approach to store answers for user questions. Our approach will focus on questions about tracing, resilience attributes, and ATAM-based scenarios.

[4] https://saraperezsoler.github.io/ModellingBot/
[5] https://pharo.org/
[6] https://discord.com/

3.2 Requirements Elicitation Methodologies

This section introduces two approaches that focus on resilience requirements elicitation in the context of microservices. Section 3.2.1 presents a case study by Kesim et al. [KWK+21] which utilizes FTA and ATAM. The applied methodologies are used to create resilience scenarios that are transformed into chaos experiments. Section 3.2.2 the approach by Yin et al. [YDW+19] is given. They use the Microservice Resilience Measurement Model (MRMM) to define architecture goals for microservice-based software systems, which can be used in the GORE technique.

In contrast to the approaches presented in this section, we aim to automate the elicitation process of resilience-related requirement attributes. Both related works present a tool-assisted method, through the Chaos Toolkit (CTK) or the MRMM. However, there still exists the need for manual configuration and expert knowledge. With our approach, we want to show that it is possible to keep manual configuration to a minimum.

3.2.1 Scenario-based Resilience Evaluation and Improvement of Microservice Architectures: An Experience Report

In a case study, Kesim et al. [KWK+21] investigated the application of ATAM-based scenarios for resilience requirements elicitation in an industrial context. From research about chaos experiments in microservice-based systems, they found out that several approaches did not follow a systematic approach to identify resilience requirements [HRJ+16; KHFH20]. Therefore, they applied the FTA risk analysis technique to identify hazards. The ATAM method and FTA which was part of the case studies methodology are explained in Chapter 2.

The case study was conducted in several steps. In the first step, they set up a workshop meeting with stakeholders of a software system from an industry partner. During the meeting, they performed a FTA and created a FT of potential hazards that could cause degradation of the quality of service or violated the service level objectives. Since creating an ATAM-based scenario requires manual configuration, the stakeholders had to write down the individual parameters. From the created FT they derived the stimuli and the artifact of the ATAM-based scenario for every individual basic event of the FT [KWK+21] (ATAM is introduced in Section 2.2.1, Section 4.1 introduces the same scenario format as the case study). The rest of the scenario parameters were filled in during the conversation between workshop organizers and the stakeholders. They used existing metrics of previous measurements and service level objectives to define the response and the response measurement. Three chaos experiments were created with the CTK from the twelve elicited scenarios. Each chaos experiment covered scenarios of the same category. The workloads for the chaos experiments were produced by a separate load generator tool [KDK18]. While the inspected system was experiencing the normal workload, faults were injected. It was then inspected if the system could carry on processing all requests, or if degradation of performance was visible in the system metrics. Services that were failing or had a degraded level of performance were identified and then equipped with resilience patterns such as the retry pattern. Once the service was made more resilient, the chaos experiments were executed again. System metrics collected during the second run showed that the applied strategies helped services not fail or experience degraded performance.

The authors of the chaos experiment showed that by transforming resilience scenarios into chaos experiments, vulnerable parts of the system could be identified and tested against specifications created during an elicitation process. The authors acknowledge that the approach still requires manual effort. Requirements have to be collected during a group meeting through FTA in support of experienced architects. Further modifications for transforming requirements into ATAM scenarios and ATAM scenarios into chaos experiments are still necessary. This produces a high load on valuable resources such as the working hours of experts. However, the case study showed that chaos experiments created from ATAM scenarios could identify vulnerable parts of a microservice-based system.

3.2.2 On Representing and Eliciting Resilience Requirements of Microservice Architecture Systems

Yin et al. [YDW+19] state that requirements elicitation for microservices is oftentimes not done in a formal way resulting in unclear definitions on how resilient microservices should be [YDW+19]. On the one hand, current definitions of resilience leave room for interpretation. On the other hand, existing measurements of resilience can yield ambiguous results. Since the resilience of a microservice-based software is especially important in situations of risk (i.e., during software updates, deployment, or at high user loads), resilience mechanisms must consider the influences of all potential risks.

Yin et al. [YDW+19] propose the MRMM which embraces several metric formulas to quantify the service performance and service degradation. These formulas can be used to define resilience goals for microservices. Such a goal contains a performance attribute of a service that describes requirements to which a system should uphold. The validity of a goal can be computed at any time as long as the performance attribute of the service is measured in a monitoring system. Goals defined with the MRMM can be mapped to the KAOS approach of GORE, previously explained in Section 3.1.4 and then further used in the goal-oriented requirements approaches. Besides the MRMM, Yin et al. [YDW+19] introduce a framework for elicitation of resilience requirements on an existing microservice-based system that has a monitoring system installed.

The framework is divided into three stages. In the first stage, the metrics and services of interest are defined. Furthermore, relationships amongst and between services and metrics are established. The second stage produces performance benchmarks from the collected system metrics, which are then used in the MRMM to define requirement goals. In the third and last stage, the benchmark of the goals is compared to runtime measurements to detect degradations in the service metrics. If a degradation violates a requirements goal, the stakeholders must discuss the application of resilience mechanisms in the targeted services to prevent further violations. Once the resilience mechanism is implemented, the three-step process is repeated until no more degradations are determined. In a case study, the authors conducted an experiment with the SockShop[7] microservice demo application using the elicitation framework approach. Using the load testing tool Locust[8] they created user profiles that were, on the one hand, used as a performance benchmark, and on the other hand to artificially create performance degradation through high loads.

[7] https://microservices-demo.github.io/
[8] https://locust.io/

Yin et al. [YDW+19] based their goal-oriented framework on quantifiable metrics, making the approach very easy to use in combination with a monitoring system. An additional benefit of this is the ability to prove service degradation based on numbers. In isolated experiments, the service degradation can be repeated and modified to inspect the effects of changes on the system. The use of the goal-driven approach guarantees that it can be used well combined with other GORE approaches. However, the second step in the requirements framework still requires the presence of domain experts and manual elicitation of interesting metrics and the relationships between services and metrics.

4 Concept and Integration

This chapter presents the concept and integration for the proposed approach. Section 4.1 describes the scenario format we adapted from Bass et al. [BCK03]. Section 4.2 introduces how a generic tracing model was designed to unify the use of Zipkin and Jaeger. In Section 4.3 the procedure of the risk and hazard analysis is given. Lastly, in Section 4.4 an architecture format is proposed that visualized the architecture of an inspected Microservice (MS).

4.1 Scenario Format

The scenario description used in this work is based on Bass et al. [BCK03]. In Listing 4.1 the extended *concrete quality attribute scenario* based on Bass et al. is presented in the JSON format. A *concrete quality attribute scenario* in contrast to the *general quality attribute scenario* is only applicable to a specific system [BCK03]. Therefore, we also use more specific details in our own extended scenario description. A quality attribute, as defined by Bass et al. [BCK03] is a measurable or testable property that determines how well a system is performing under specific conditions. For this work, we only consider quality attributes concerned with the resilience of a MSA. We further limit the use of quality attributes in the analysis method (described in Section 4.3) to the metrics available through traces. However, the scenarios created with the prototype presented in Chapter 5 are not limited to the results of the analysis. Users of the prototype are able to extend the quality attributes through custom inputs.

Listing 4.1 The extended scenario description in the JSON format.

```
{
  "description": ...,
  "artifact": ...,
  "component": ...,
  "stimulus": ...,
  "source": ...,
  "environment": ...,
  "response": ...,
  "response-measure": {
    "normal-availability": ...,
    "normal-response-time": ...,
    "normal-response-cases": ...,
    "recovery-time": ...
  }
}
```

A resilience scenario in this work consists of eight parameters and between two and three sub-parameters. As mentioned before we extend the quality attribute ATAM scenario structure from Bass et al. [BCK03] and Kesim et al. [KWK+21]. Bass et al. define six parameters for the quality attribute scenario *artifact*, *stimulus*, *source*, *environment*, *response*, and *response measure*. Kesim et al. extended the scenario description by one additional parameter, the *description*, which gives a human-readable summary of the contents of the scenario. Our work extends the quality attribute scenario with the parameters *component*, *normal availability*, *normal response times*, *normal response cases*, and *recovery time*.

In the following we summarize the use of the scenario parameters. The (1) *description* offers a summary of the scenario and bears no technical details. However, it gives stakeholders a rough overview of what the scenario is about. (2) *artifact* and (3) *component* are always used in combination with each other. The *artifact* describes a functioning part of the system (i.e., service, instance, or operation) in a more general way. The *component* provides a more specific description (i.e., redis, database, getUser) and defines the actual name of the instance defined in the artifact. (4) *stimulus*, (5) *source*, and (6) *environment* are also used in combination. While the *stimulus* represents an arriving event at the system, the *source* describes the cause of it, and the *environment* the system state at the arrival of the *stimulus*. For the description of these three parameters in the prototype we use a template-based sentence fragment *({stimulus} caused by {source} ({environment}))*. An actual example would be *{service failure} caused by {a software bug} ({during deployment})*.

The (7) *response* represents how the system should respond to the *stimulus*. A *response* can be defined loosely as *the service should restart* or more precise *the service must retain a availability of 90%*. In practice it is more common to specify the *response* openly and define quantifiable metrics in the (8) *response measure* [BCK03]. The *response measure* provides a comparison of the system at a state of potential failure with the normal state. This way, it can be tested if the state of the system after the occurrence of the *stimuli* is violating the systems' quality of service. The *response measure* can be divided further into two and three sub-parameters, respectively. Since we use traces from Zipkin and Jaeger as input for a system description, we only distinguish between two types of *artifacts* – services and operations. Furthermore, the response time is the only available metric from the traces. A *response measure* for services has (8.1) *normal-availability* and (8.2) *recovery-time*. The *response measure* for operations has (8.1) *normal-response-time*, (8.2) *normal-response-case*, and (8.3) *recovery-time*. These parameters of the *response measure* come with the same compromise as Bass et al. describe their original quality attribute scenario. "[a] common form to specify all quality attribute requirements […] has the advantage of emphasizing the commonalities […] and the disadvantage of occasionally being a force-fit for some aspects of quality attributes." [BCK03]. We limit the use of the extended scenario structure to trace metrics in favor of a more detailed scenario description. The parameters *normal-availability*, *normal-response-time*, and *normal-response-cases* are used to describe a systems normal operation state. Quantitative measures are used to specify these parameters (i.e., 100ms or 99%) that can easily be checked against runtime metrics of the inspected system. In case that system metrics deviate from the given metrics of the normal operation state, a return to the normal state is necessary. The maximum tolerable time to return to the normal state is described by the *recovery-time* parameter, which also uses a quantitative measure (i.e., 5 minutes, or 4 seconds). If the *recovery-time* is exceeded and the system metrics do not align with the specification of the normal state, the system is in a state of failure.

In Section 4.1.1 we emphasize how the *response measure* can be formalized with MTL. Other parameters of the extended scenario description can not be formalized since they are not used in a measurable context.

4.1.1 Scenario Formalization

For the formalization of a scenario, we use one of the PSPs as first introduced by Dwyer et al. [DAC98]. A PSP describes a solution for reoccurring problematic events [DAC99]. PSPs were refined further by Autili et al. [AGL+15] and categorized into different pattern families. Furthermore, they introduce the PSPWizard that allows the transformation of template-based English grammar sentences into a temporal logic format. Semi-structured English sentences (i.e., If event {X} does occur after event {Y}, then state {S} must eventually hold.) can be configured with states and events through a GUI. The tool provides transformation of such sentences into multiple temporal logics (i.e., LTL, CTL, or MTL). In this work we only consider time-based PSPs to describe events that should occur after a certain time. Therefore, we only demonstrate the transformation to a formula in MTL. In the following an example for a *Response Pattern* in the semi-structured English grammar, and MTL is given:

Structured English Grammar:

Globally, if {A} then in response {B} within {X} seconds.

MTL:

$$\Box(A \rightarrow \Diamond^{0,X} B)$$

In our approach, we use the semi-structured format for the presentation of *stimulus*, *response*, and *response measure*. Furthermore, we use a restructured sentence format in favor of readability for users of the developed prototype (Within {X} after the occurrence of {A} {B}.). In the following we give an example of the parameters *recovery-time* (1.5min), *stimulus* (a failed request), *response* (a developer is notified) used in the semi-structured english sentence. Within {1.5 min} after the occurrence of {a failed request} {a developer is notified}.

4.2 Tracing Models

Traces from Zipkin and Jaeger are the basis for the architecture description. We use both of these technologies for two reasons. On the one hand, there is an existing basis of literature and academic work [BAG17; KMB+17; Mac17; SFSG14; Shk19]. On the other hand, both frameworks are widely supported and used in the industry and open-source projects. Further details and introduction on Zipkin and Jaeger can be found in Section 2.4.1.

Traces from either tool can be exported with the respective UI or via exporters from external programs connected to the trace collector. The exported document is stored in the JSON format and consists of four major categories.

- Traces: An exported document must contain at least one trace but may contain multiple traces. One trace summarizes multiple spans of one execution.
- Spans: One trace must contain at least one span but may contain multiple spans. One span can be mapped directly to one process and contains additional meta-information about the span's execution.
- Processes: A process represents an endpoint of the distributed system. A trace must have at least one process but may contain multiple processes. A process can be identified uniquely by combining service name, Internet Protocol (IP) address, and port number.
- Meta-information: Meta-information is made up of execution times, logs (sometimes called annotations), and tags. The content of logs is mapped against timestamps collected during function execution.

In Listing 4.2 and Listing 4.3 we give excerpts of two traces from both tracing tools. We omit the meta-information of logs (annotations) and tags in favor of readability. Unimportant data or a continuation of items in a list is omitted and hinted at with three dots (…). Listing 4.2 shows an excerpt of a Zipkin trace. A Zipkin trace consists of a list of spans. A span describes a unit of work inside a distributed system. It can be a remote HTTP call, or a local operation. Each span contains a field that identifies its associated trace by the field *traceId*. In addition, the trace contains the fields *name*, *timestamp*, and *duration* representing the name of the executed operation, the time it was started (as UNIX timestamp), and the amount of time it took (in microseconds). *localEndpoint* and

Listing 4.2 An example of an exported Zipkin trace in the JSON format.

```
[{
    "traceId": "9215cc9e6122b3ef",
    "id": "9215cc9e6122b3ef",
    "name": "getUser",
    "timestamp": 1607345595110503,
    "duration": 389007,
    "localEndpoint": {
      "serviceName": "frontend",
      "ipv4": "127.0.0.1"
    },
    "remoteEndpoint": {
      "ipv4": "172.18.0.1",
    },
    "annotations": [...],
    "tags": {...}
  }, {
    "traceId": "9215cc9e6122b3ef",
    ...
  }, ...
]
```

remoteEndpoint define the endpoints of the caller and the callee. The completeness of the endpoint contents depends on the configuration of the instrumentation library. It is possible that an endpoint can not be identified by its name but only via the IP address.

Listing 4.3 shows an excerpt of a Jaeger trace. In contrast to a Zipkin trace, the Jaeger trace consists of a list of traces that contain their associated spans as another list. An advantage of the Jaeger trace format is the hierarchical structure of traces and spans. Every span of a trace is stored in a separate list inside the trace object. Furthermore, process names are also given as key (*p1*) inside a span referring to the *processes* object. This provides a quick way to lookup process names while separating the traces from processes. Dependencies between spans are given explicitly in Jaeger, while dependencies in Zipkin have to be computed. All parameters of the Jaeger span, except the field *references*, can directly be mapped to parameters of a Zipkin trace. However, Jaeger uses different naming conventions than Zipkin for some parameters. The field *references* contains a list of references to other spans, or traces. The *refType* defines the hierarchy of the described trace in relation to the one referenced in the *references* field.

To unify both representations of Zipkin and Jaeger into one format, we introduce the design given in Figure 4.1. The *TracingModel* provides an interface for the *ZipkinModel*, and *JaegerModel*. Both models have to implement a parsing method that identifies services, operations, and their dependencies in the JSON file. The *Service* and *Operation* classes contain an optional object of

Listing 4.3 An example of an exported Jaeger trace in the JSON format.

```
{
  "data": [{
      "traceID": "118b01fd312719db",
      "spans": [{
          "traceID": "118b01fd312719db",
          "spanID": "14e1d17fdcf149dd",
          "operationName": "getDriver",
          "references": [{
              "refType": "CHILD_OF",
              "traceID": "118b01fd312719db",
              "spanID": "1524e11a8421c310"
          }],
          "startTime": 1606579665241063,
          "duration": 174788,
          "tags": [...],
          "logs": [...],
          "processID": "p1"
      }],
      "processes": {
        "p1": {
          "serviceName": "driver",
          "tags": [...]
        }, ...
      }
    }, ...
]}
```

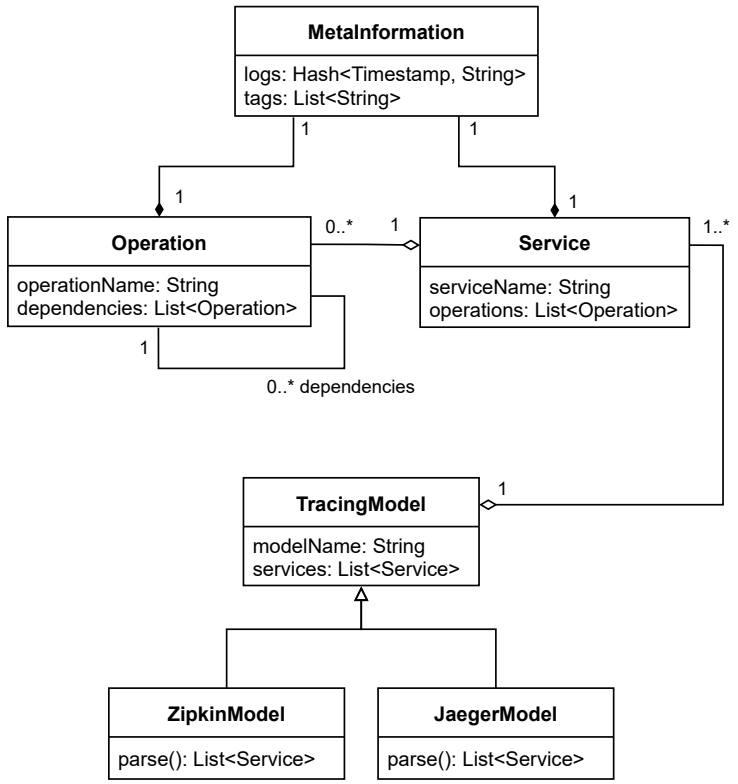


Figure 4.1: Class diagram of the trace model.

the class *MetaInformation*. The *MetaInformation* class includes the data of logs and traces that are not essential to create a graph. With an instance of *ZipkinModel* or *JaegerModel* it much easier to create a Directed Graph (DG) from the list of services.

4.3 Hazard Analysis

Section 2.3 gives an overview of established risk analysis techniques. HAZOP is a comparatively old technique for analyzing hazards. Since its development in the 1960s, it has seen several modifications. Variations of HAZOP led to analysis methods focusing on domains other than the chemical industry [BP93; FH94]. We are especially interested in CHAZOP, a variation aimed at software systems. The CHAZOP approach has also seen successors [FL14; HT11] focusing on specific quality attributes of a software. In this work, we apply a CHAZOP-based approach as a basis to identify hazards in tracing models (described in Section 4.2).

Like the CHAZOP procedure shown in Figure 2.2, this analysis follows a particular set of steps. First, it is necessary to partition the system under inspection into smaller components. We achieve this, through the definition of the *TracingModel* described in Section 4.2 and the creation of an architecture format given in Section 4.4. Secondly, the CHAZOP approach requires iterating over all previously defined system components. In our approach, this is realized with the traversal of the

DG created in the *TracingModel*. In the third step, a parameter of the selected system component is inspected. As mentioned before, we focus on response times, the only available metric in traces from Zipkin and Jaeger. After the third step, we vary from the original CHAZOP procedure.

Instead of applying a deviation to the component's parameter based on expert knowledge, we examine response times from previous executions of the system and extract hazards this way. This is, in fact, much more precise than suggestions made by an expert resilience engineer in the original CHAZOP method. However, the examination of the metrics requires two prerequisites. On the one hand, it is necessary that the system did run before and went into a hazardous state. For some systems, this is not tolerable. On the other hand, it requires an algorithm to identify the hazard. We want to emphasize that the inspection of trace metrics and resulting hazards provides resilience engineers

Listing 4.4 CHAZOP-based procedure to identify hazards at components of the system, given in pseudo-code.

```
S is the list of services

for service in S do
  O is the list of operations in service

  for op in O do
    RF is the list of recorded response times of op filtered by 3 times the standard deviation

    # Response times differ by 50%
    if 1 - min(RF) / max(RF) > 0.5 then
      add ResponseTimeDeviation to op
    endif

    # Spike workload
    if length(RF) > 0 then
      add ResponseTimeSpike to op
    endif

    Ho is the list of hazards in op

    for hazard in Ho do

      if hazard = ResponseTimeSpike then
        add ServiceFailure to service
      endif

      if hazard = ResponseTimeDeviation then
        add DecreasedServicePerformance to service
      endif

  endfor

endfor
```

only with suggestions. Resilience engineers may still use their own knowledge about resilience to define their custom stimuli types. However, our approach can highlight critical components to resilience engineers that may require special attention.

We present an algorithm in Listing 4.4 in pseudo-code, that is able to identify different types of hazards. The list of services reflects the system components. For each dependant operation that the service has to other system components, previous execution times are inspected. We identify two types of hazards for operations, *ResponseTimeDeviation*, and *ResponseTimeSpike*. *ResponseTimeDeviation* describes response times that deviate by 50% filtered by three times the standard deviation. A *ResponseTimeSpike* on the other hand, detects outliers in the response times. If a hazard was detected in one of the service's operations the algorithm suggests *ServiceFailure* as a service hazard, in case of *ResponseTimeSpike*. In the case of *ResponseTimeDeviation* as an operation hazard, the algorithm suggests *DecreasedServicePerformance*. -based keywords are applied manually to the individual hazard types. Furthermore, the used number for filters and deviations is based on our own concept. The algorithm may be extended by other hazard types and dynamic calculations of filters and ranges for deviations.

It should be noted that the computation of two hazard types for operations is highly dependant on previous runs of the software and the number of calls between components. Results from the analysis algorithm are stored in the architecture format (described in Section 4.4). The content of a hazard includes its name, the applied keyword, artifact, component, inspected metric, and the value of the metric. From there on, we continue with the original CHAZOP procedure and iterate over the other system components.

4.4 Architecture Format

A new JSON format was created to utilize exported data from Zipkin and Jaeger. An example of the format is given in Listing 4.5 which has three major sections. Processes are mapped to nodes, spans are mapped to edges, and the analysis field contains the content produces by the hazard analysis.

- Nodes: A key-value lookup table is provided that maps a unique identifier against a process of the original trace. Every node is mapped to exactly one process. Furthermore, the name of the process is represented by the *label* parameter, and instead of the IP address, the unique identifier is used. The identifier is used as the key for better lookup performance in the graph creation and interaction. The *data* parameter contains the contents of both logs and annotations.
- Edges: Similar to the *nodes*, every edge is distinguished by a unique identifier that is mapped against a span. In addition, an edge contains the identifier of the calling process (*source*) and the identifier called process (*target*). As mentioned before, the identifier of either *source* or *target* can be used to quickly lookup the associated process's meta-information.
- Analysis: The *analysis* contains a list of potential hazards that were identified in the analysis of the traces. The content of the hazard is used by the chatbot to make suggestions to the user during the elicitation process.

Listing 4.5 The architecture and analysis description format. A trace model is converted into this representation.

```
{
  "nodes": {
    "0": {
      "id": 0,
      "label": "frontend",
      "data": {
        ...
      }
    },
    "1": {
      "id": 1,
      "label": "route",
      "data": {
        ...
      }
    },
    ...
  },
  "edges": {
    "0": {
      "id": 0,
      "label": "getRoutes",
      "source": 0,
      "target": 1,
      "data": {
        ...
      }
    },
    ...
  },
  "analysis": {
    "Response Time Deviation": {
      "artifact": "operation",
      "component": "getRoutes",
      "metric": "response time",
      "keyword": "differ by",
      "value": 0.7035524389304153
    },
    "Service Failure": {
      "artifact": "service",
      "component": "route",
      "metric": "throughput",
      "keyword": "no",
      "value": 0.0
    },
    ...
  }
}
```

5 Resirio

This chapter describes how the concepts defined in Chapter 4 were implemented into a prototype. The focus of this prototype is to enable interactive elicitation of resilience scenarios through a chatbot called Resirio. The architecture of the prototype is presented in Section 5.1. A rough overview is given on how the prototype operates. In Section 5.2 the design for the UI of the prototype is described. The design of interactive elements is depicted, and the interaction of a user with the prototype is described. Section 5.3 introduces the used technologies. Section 5.4 summarizes the technical details and implementation design of this work. In Section 5.5 a summary of the prototype is given.

5.1 Architecture

Figure 5.1 shows the three-layer architecture of the application and all components of the prototype. We decided to design the application this way because a classical three-layer application provides easy maintainability to developers and is easy to understand for other stakeholders. Furthermore, a three-layer application enables fast development and is the best fit for a prototype.

The prototype operates on three layers, the presentation layer, the application layer, and the data layer. The prototype's structure can be decomposed into two major modules, the *Interactive Interface* and the *Architecture & Analysis*. Both modules operate on the presentation layer (frontend) and the application layer (backend). The *Architecture & Analysis* uses the data layer for the storage of traces.

In the following we give a rough overview of a usage example. On the lowest layer, the data layer, traces are uploaded by the user and transformed into a *Model Description* (introduced in Section 4.2). This model is then further transformed into an *Architecture Description* (given in Section 4.4). In the application layer, the *Model Description* and *Architecture Description* are first analyzed on syntactic and semantic correctness. Secondly, both descriptions are analyzed in the *Hazard Analysis* (presented in Section 4.3) for hazards. If hazards are found, they are stored in the *Architecture Description* which is later used by the *Chatbot*. On the presentation layer, the *Architecture Description* can be accessed via a Representational State Transfer (REST) interface. It is also possible to export both descriptions from traces locally via a Command Line Interface (CLI).

The *Interactive Interface* operates only on the application layer and presentation layer. When a user accesses the *Client Interface* the chatbot instantiates the *Dialogflow Client*. This client enables the chatbot used in the *Interactive Interface* to access functionalities of the *Dialogflow Agent* through an API. The *Dialogflow Agent* is created in the Google Cloud[1] and contains intents of the chatbot

[1] https://cloud.google.com/

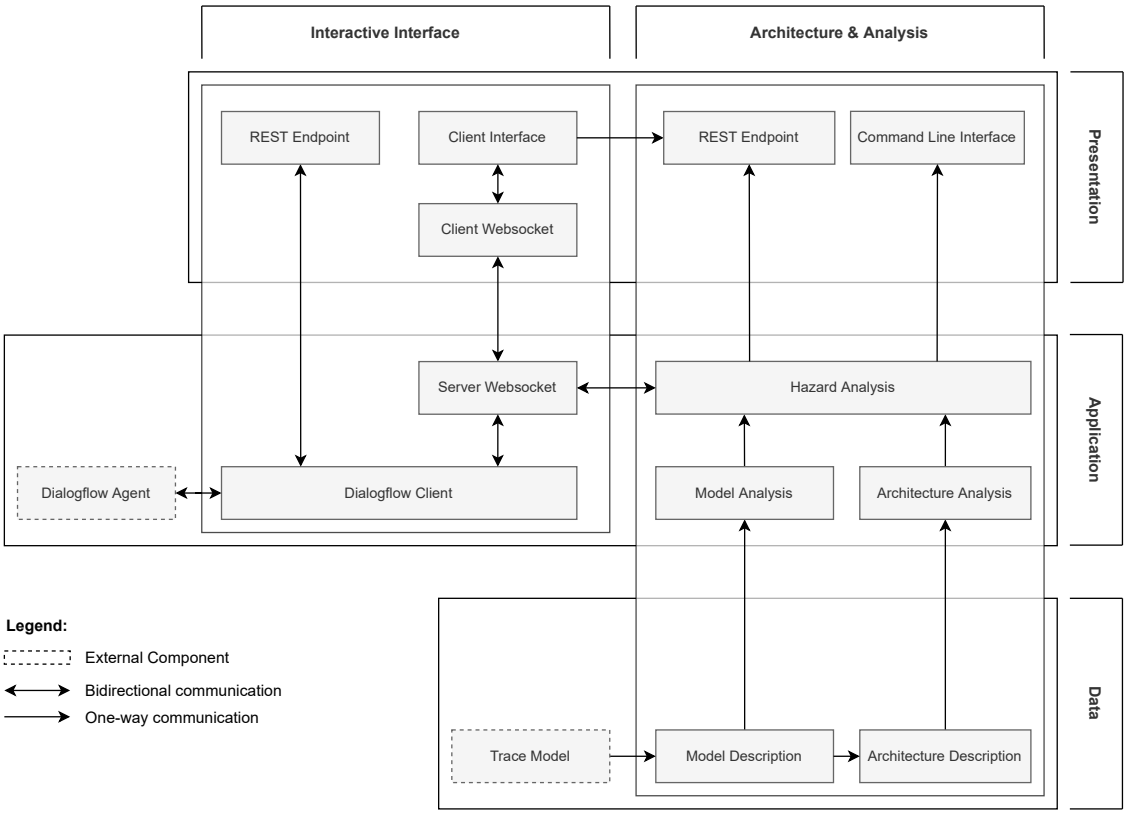


Figure 5.1: Three-tier architecture diagram of the prototype.

that are introduced later in this chapter. For the intent detection, which enables the text conversation between user and chatbot, another REST endpoint is provided. However, conversations with the chatbot are realized through the *Client Interface* and not the REST endpoint.

5.2 Design

Before defining the prototype's design, we inspected chatbot text-based integrations for Dialogflow (DF) (i.e., Dialogflow Messenger, Web Demo, Slack, and Telegram) on extension possibilities. This is described in more detail in Section 5.4.2. Testing the existing integrations showed that they were not extensible or the appearance could not be customized. The chatbot is the essential component of the prototype. Therefore, it needed to have a prominent portrayal in the UI. This is why we decided to design a custom chatbot integration.

Figure 5.2 depicts a sketch of the *Client Interface*. The essential requirement for this design was to have an easy customizable UI for users. It also had to be easy-to-use and needed to be extensible for potential future work. Each component of the *Client Interface* was designed to be easily replaceable, which accelerates the prototypical development. Another advantage of the single page layout design is the focus of attention that a user has while using the *Client Interface*. Users immediately

recognize their input by the response in either one of the major three subcomponents. The (1) architecture graph, the (2) configuration view, and the (3) chatbot as highlighted with red numbers in Figure 5.2.

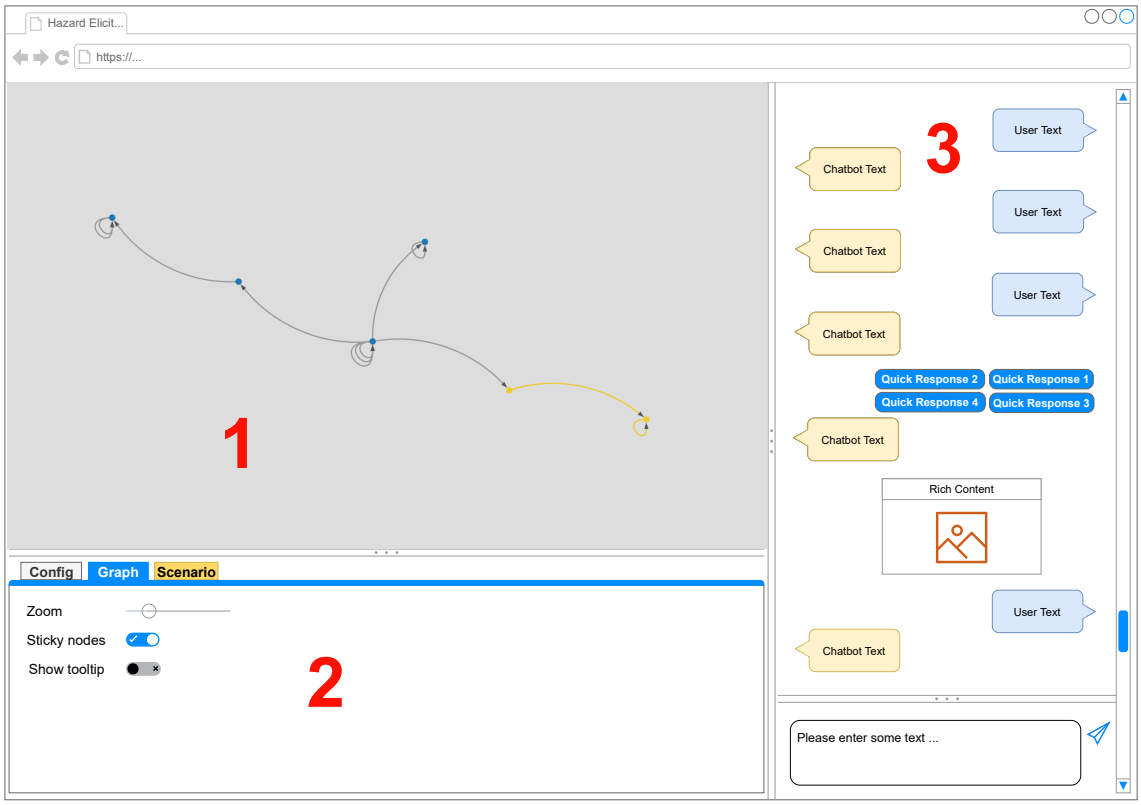


Figure 5.2: Sketch of the prototype's client interface.

The architecture graph component is designed to have a clear representation of a systems architecture while being easily usable. Hovering over nodes and edges gives users immediate feedback in the form of tooltips. A tooltip shows the label of the node or the edge that is stored in the *Architecture Description*. By design, only the node names are visible to the user once the graph is created. However, the option to show the labels of the edges is also given in the configuration view. Testing of various architectures showed, that the display of all labels from nodes and edges can become messy. In general, overlapping text is, hard to read. Therefore, we decided to disable the labels of the edges by default. An alternative to prevent text overlapping is to create a virtual background for the texts in the form of a Scalable Vector Graphic (SVG) rectangle and place it behind the text. Testing this showed that calculating the positions and rendering the rectangles does decrease the performance quite drastically. The problem also gets worse with an increasing number of nodes and edges. Further interaction through zooming and dragging the graph might not be immediately visible to users of the system. By adding a zooming widget to the configuration view, the users should implicitly recognize that this functionality is given in the prototype. Another hint that the architecture graph can be interacted with is given when the graph is first showing up in the SVG container of the subcomponent. With the creation of the graph, a simulation starts, placing the

nodes in the view with equal distribution. Furthermore, when clicking on nodes and edges, the chatbot is notified about the action and highlights the corresponding graph-artifact within the graph. This is also further explained in Section 5.3.

To highlight important graph-artifacts, we use the notion of priority. Each node or edge in the graph is assigned a color from white (low priority) to blue (high priority). The color palette is interpolated from the lowest priority to the highest priority in the graph. The priority is calculated during the creation of the graph. Its value depends on the number of incoming and outgoing edges and the number of hazards found on this graph-artifact. A service with many connections to other services has a higher priority than services with less connections. The priority of an edge is defined by the two services it connects. If hazards are found in a service or operation during the analysis, the service's priority is multiplied with the number of found hazards. We did not use an existing method for calculating the priority, but rather defined our own. It is loosely based on the formula of the RPN used in FMEA. The chatbot uses the priority for the selection of the artifacts. It suggests the five nodes and five edges with the highest priority to the users. In the following, the priority calculation for nodes and edges is given.

For every graph $G = (V, E), V$ is the set of nodes,
and E the set of edges with $E \subseteq \{(x, y) | (x, y) \in V^2\}$

Number of Hazards: $H(c), c \in V$ or $c \in E$
Number of outgoing Edges: $O(n), n \in V$
Number of incoming Edges: $I(n), n \in V$

Node Priority: $NP(n) = (I(n) + O(n)) * (3 + H(n)), n \in V$
Edge Priority: $EP((n_1, n_2)) = (NP(n_1) + NP(n_2)) * (1 + H(n_1) + H(n_2)), (n_1, n_2) \in E$

The configuration view is organized as a tab view and offers three different types of tabs. The first type of tab offers help texts to the users and assists them during the elicitation process. The second type of tab shows the architecture graph's configuration. Besides the already mentioned features, it is possible to lay out the nodes in fixed positions. This allows users to inspect nodes in isolation. Edges can also be rendered as curved lines instead of straight ones. Further details on the rendering of the graph are given in Section 5.3. The third type of tab presents a finished scenario, once it is saved by the chatbot as shown in Figure 5.3. The scenario tab summarizes the elicited parameters of a resilience scenario and provides an export button to the users. The export button lets the users download the scenario in a JSON format to their PC. The format of the downloaded scenario is described in Section 4.1.

The chatbot is divided into two subcomponents. The first subcomponent is the chat input, where users can write text to the chatbot. Once sent via the send button, the written text appears in the second subcomponent, the chat content. The chat content has three kinds of content types. On the right side of the chat content is the user content (blue). The left side of the chat content contains the chatbot responses (yellow). Rich content covers the middle of the chat content, which renders *Card Responses*, *Accordions*, and *Quick Replies* as explained in Section 5.4.2. The chat content follows

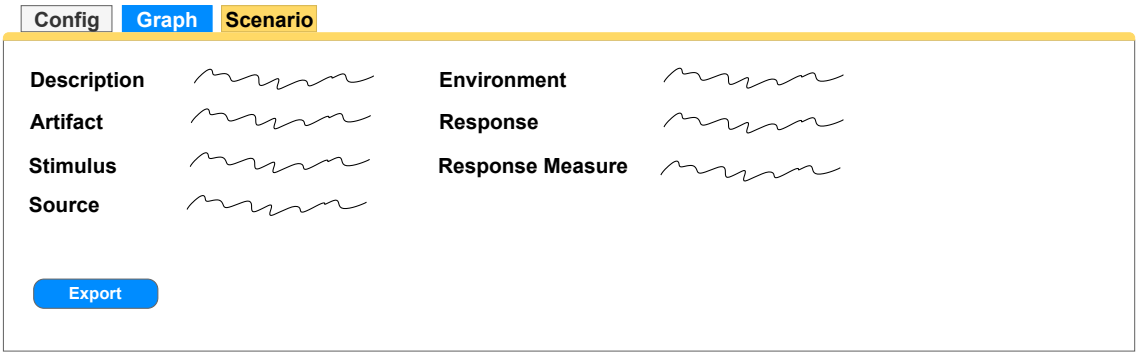


Figure 5.3: Sketch of the prototype's configuration view with a finished scenario.

the design of a typical messenger interface. Each message of the communicating partners is drawn on either side of the chat. Messages in the chat flow from the bottom to the top. This means that new messages appear at the bottom of the chat content.

5.3 Technologies

Since the prototype's frontend is implemented to work in the browser, the used frameworks are primarily web-based. Static elements are rendered in Hypertext Markup Language (HTML) and stylized with Cascading Style Sheets (CSS). In addition to the standard CSS, the open-source framework Bootstrap is used. Bootstrap is a CSS library initially developed by Twitter and allows frontend developers to create responsive HTML layouts. In addition it gives the option to extend default HTML widgets and renders more complex constructs (i.e. tab views, navigation bars) that are not available in standard HTML. Furthermore, Bootstrap comes with a built-in JavaScript (JS) extension that handles input events from the user in the UI and enables customization of event handlers.

The combination of the three technologies HTML, CSS, and JS builds the foundation for the page layout. As depicted in Section 5.2 the UI is designed to have three separate subcomponents on the *Client Interface*. To separate these three subcomponents while keeping them on the same layout, Split.js[2] is used. This JS framework allows a user to dynamically arrange the subcomponents by dragging panes and resizing the content. We decided to use Split.js because it is well-maintained and produces only low overhead. Its total size is 2KB.

For the basis of the backend, we decided to use Django[3] which is a web server framework for the Python programming language. Django comes with a multitude of libraries and extensions that can easily be configured. On the one hand, it provides configuration files that can be configured quickly by new users. On the other hand, the framework also enables more experienced Python developers to extend existing interfaces (i.e., Database Models or the REST-Framework). The use of Python in the backend also lead to the decision to implement the architecture extraction and

[2] https://split.js.org/
[3] https://www.djangoproject.com/

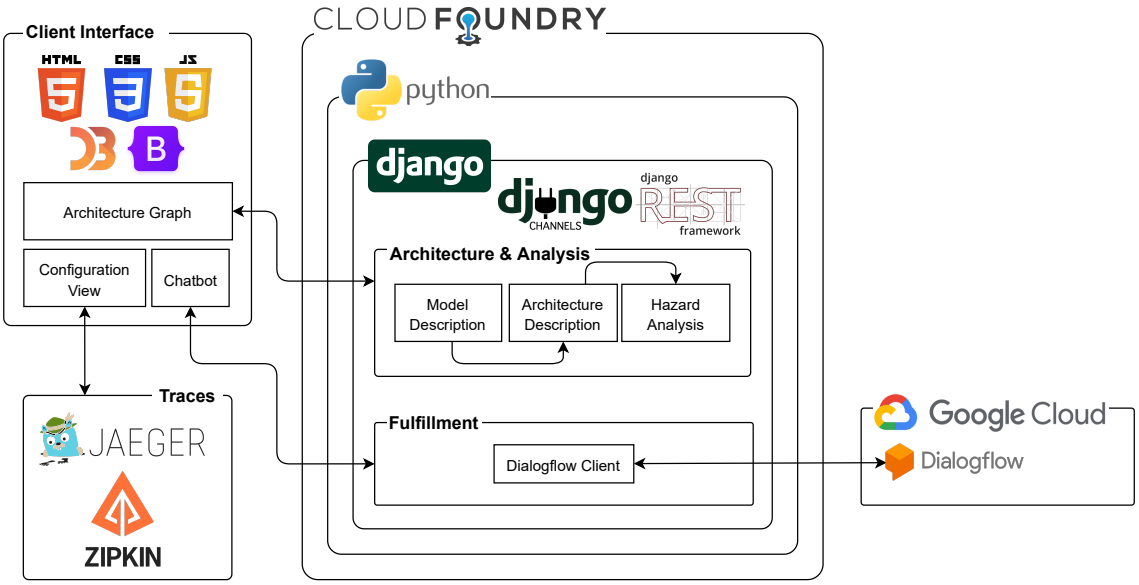


Figure 5.4: Overview of all technologies used in the prototype.

the DF client in Python, which is emphasized in Section 5.4.3 and Section 5.4.4 respectively. Our decision to use Django as a webserver and foundation for the prototype is based on the experiences of Okanović [OBM+20] and Beck [Bec20]. This is also emphasized later in this chapter.

Figure 5.4 shows the interplay of all the involved technologies of the implementation. The *Client Interface* contains the presentation layer, which is rendered on the clients' PC. The backend is deployed in a Cloud Foundry (CF)[4] environment. At the source of every interaction with the prototype is a tracing model from Zipkin or Jaeger. Traces can be uploaded via the configuration view in the *Client Interface* to the backend. Uploaded traces are stored in the Django REST framework extension, making it accessible in the UI or for other endpoints. From there, the trace is first transformed into the generic *Model Description*. Secondly, the trace model is used in the architecture extraction, which creates a DG. The resulting DG is stored in the *Architecture Description* format. If both *Model Description* and *Analysis Description* are present, they are analyzed for hazards with the CHAZOP-based approach. Once uploaded, the analyzed trace can be selected via the UI in the configuration view. The chatbot can also access available traces in the backend and make suggestions to the users based on the name of the trace.

The selection of a trace initiates the chatbot conversation. Every user communication between the *Client Interface* and the backend is realized via WebSockets. This means that input from the user, via text input or *Quick Replies*, is first sent to the backend. An example of the client to server WebSocket message is given in Listing 5.1. In the backend, the content of the WebSocket message is unpacked and given to the *Dialogflow Client*, which connects to the DF cloud service. Here the user input is analyzed, and matched against intents. If the user input does match an intent, the respective intent handler is called, which produces the response (fulfillment) to the user. The content of this

[4] https://www.cloudfoundry.org/

response is then sent via the backend to the client WebSocket. The use of WebSockets is described in more detail in Section 5.4.4. An example for a response message sent to the client is given in Listing 5.2.

In addition to the already mentioned technologies, the CF cloud delivery platform is used for the deployment of the backend. There exist several other cloud deployment platforms like Heroku[5] or AWS[6] that provide similar offerings. However the offerings provide only limited access for free resource usage. For example, Heroku uses so-called dyno hours that allow developers to run their software only for the time they paid. The free plan without paying any costs provides only 550 dyno-hours which is roughly 22 days. This duration is not suitable for the use of the developed prototype beyond the user study. The IBM cloud offering, which is built upon CF provides a free plan that allows developers to deploy their apps for any amount of time, however only with limited resources. The free plan limits resources to 512 MB of memory and 1 GB disk storage. CF has the additional advantage of vendor-free deployment infrastructure. This means that a CF application can be deployed on OpenStack, AWS, or other services that support the CF deployment model.

5.4 Implementation

As mentioned before, the *Client Interface* and the *Architecture & Analysis* operate on different layers of the architecture. In the following, we distinguish at least two different layers (frontend, backend) for *Client Interface* and the *Architecture & Analysis*. Section 5.4.1 and Section 5.4.2 focus on the visualization while Section 5.4.3 and Section 5.4.4 describe the corresponding backends.

5.4.1 Architecture Frontend

The visualization of the architecture, which is embedded in the *Client Interface*, is done by generating a DG from the *Model Description*. Nodes represent processes, and edges represent spans. This graph is constructed using the JS framework D3.js[7] which offers a wide range of data visualization. The force-directed network functionality of the D3.js library enables the creation of graphs. A default graph can only be configured to have non-directed edges. Since spans from a trace describe dependencies between services, it is necessary to have directed edges. This functionality does not exist in the given implementation, so we implemented it on top of the given framework. Furthermore, labels for nodes and edges are not part of the standard implementation. This functionality was added as well. In addition, the force-directed network's library was extended with a zooming function and the option to drag the graph around in the UI.

For further inspection of details of a trace, the graph is extended with tooltips and a context menu. It can be opened through a right-click event from a user on nodes or edges. Once it is opened, it creates a list of available meta-information. Nodes display names of processes and the endpoint information (hostname and port). Edges display the execution details of spans such as duration of a call, logs, and tags.

[5] https://www.heroku.com/
[6] https://aws.amazon.com/
[7] https://d3js.org/

For further customization, the configuration view contains the options to show or hide nodes, edges, and their associated labels. As graphs can take up a large amount of the available screen, users can decide how many details they want to see. The force simulation of the graph runs once the graph is loaded. It distributes the nodes equally on the available screen space. Since dragging single nodes on the screen affects the positioning of other nodes, a "sticky" option was added for nodes. The sticky option enables an arrangement of nodes in fixed positions and disables the force simulation. As a result, users can group nodes or inspect specific ones that are of particular interest.

5.4.2 Chatbot Frontend

Section 5.4.4 describes in detail why we used DF and how we implemented it in the backend to exchange messages between users and chatbot. This section focuses on the implementation of this work's custom integration for DF.

DF provides so called integrations that make a chatbot defined in DF available in either, voice-based or text-based applications (i.e. Telegram, Slack, Skype). Besides third-party providers, DF offers a Web-Demo and the DF Messenger in the beta version. The functionality of the beta versions is limited and supports only text-based user interactions with the bot.

In an early tests of the Web-Demo and DF Messenger integrations, we identified one major drawback. Both integrations do not allow for custom-made, dynamic responses via code generation. The only possibility to do this is by using webhooks. An integration using a webhook calls the corresponding URL and sends an HTTP request. A complete DF client response is encoded within this request and has to be parsed by the receiving endpoint. This induces further overhead and high response times to the chatbot interface. Another reason for not using the integrations is the absence of customization for the interface. The chatbot interfaces offer a fixed unresponsive chat window with small text input because they are meant to be used with mobile devices. Furthermore, the customization in the design is limited to the color of the chatbot window. For this thesis, the chatbot represents the central functionality of the system and therefore should have a prominent representation in the UI as outlined in Figure 5.2. As a result, we decided to create a custom and generic chatbot frontend that other developers can use. The only requirement this frontend implementation has is the need for WebSocket communication with the backend. The backend also has to use a predefined but extensible JSON-like response format. The backend implementation of this work provides an interface with generator methods to efficiently create the responses.

In the following, all available response types are listed and briefly explained:

- Empty: Contains payload that can be used for debugging or custom implementation of the UI.
- Action: Creates a client-side function call with the specified function name and parameters.
- Multi-action: Contains several *Action* responses. This prevents the overhead produced by sending multiple single *Action* responses.
- Formatting: A *Formatting* response steers the appearance of the chatbot conversation. The prototype presented in this thesis uses only dividers between specific messages to make changes of topics more distinctive.
- Text: Represents a message sent by either the user or the chatbot. The author of the text message is implicitly detected by either outgoing (user) or incoming (bot) messages.

- Quick-reply: Provides the user with possibilities to give a predefined answer instead of writing it in the chat. The developer has to make sure that the chatbot UI is not flooded with too many *Quick Replies*.
- Card: The *Card* response is neither part of the user nor the bot and is one of two rich responses that provide the user with additional information. Examples are, explanations for topics stored in the knowledge base, or headings for topic changes. The *Card* response can contain images, hyperlinks, and spoilers for additional texts.
- Accordion: The *Accordion* provides a list of collapsible sections. Each section can contain several text elements.

5.4.3 Architecture Backend

When creating the architecture from either the Zipkin or the Jaeger trace, two interfaces are given to the users. The first option is to use the CLI. The CLI is equipped with command line arguments. This way, users can tell the analysis program for which type of traces the analysis should be done. Furthermore, it can be defined to what degree the models should be analyzed. Finally, the export type of the model can be specified. The second option is to use the UI that comes with the prototype's web interfaces in the browser. However, the options of the UI are not configurable and are hidden to the users.

The architecture extraction is implemented generically. Every type of trace is first transformed into a *Model Description* that unifies the different formats of the traces. All tracing models inherit from a base class that forces the child classes to implement two parsing methods. One method parses a single trace, while the other one parses a collection of traces. During the parsing of the trace models, every process and the related spans are stored in the base class in a key-value lookup. This lookup table is later used to create the graph. Additionally, the trace models are checked for syntactic and semantic correctness. In case of missing process names, ill-configured process relations, or cyclic dependencies, the users are notified about the warning or error. Is the parsing done, the transformation from the model into the architecture is executed. Here, the previously mentioned lookup table is used to create a DG. For the lack of low-overhead libraries, a small DG library was implemented. Once the architecture graph is created, the meta-information is attached to every node and edge. Finally, the analysis is done by iterating over the graph and looking at available metrics. The analysis can detect two different types of hazards for operations (Response Time Deviation, Spike Response Times) and two types of hazards for processes (Service Failure, Degraded Service Performance). These four types of hazards are solely detected by the inspection of response times from spans.

In the web UI a drag and drop file upload is provided that takes an exported JSON file from either tracing tool. The import is executed automatically and notifies the users about success or failures. The endpoint offers the same functionality as the CLI however, it gives the users no custom options for analysis and export. If the web UI is chosen for the analysis, the architecture model of the trace is stored in a database. The architecture model is then available to the chatbot and in the configuration

Listing 5.1 Contents of the websocket communication from the frontend to the backend.

```
{
  "uuid":     "abcd-efgh-1234-5678",
  "type":     "event",
  "data":     "e-set-architecture",
  "contexts": [{
    "name": "architecture",
    "lifespan": 100,
    "parameters": {
      "architecture-name": "hotrod.json"
    }
  }]
}
```

view. Furthermore, a REST interface is provided to users to inspect the value of the analyzed and exported architecture. For a more detailed overview of either CLI or the UI take a look at the official GitHub repository[8] where further documentation exists.

5.4.4 Chatbot Backend

Okanović et al. [OBM+20] compared the capabilities of different chatbot frameworks for the application of interactive load test execution. They found out that DF provides an easy-to-use interface for developers and produces fast results. Based on Okanović et al. and early tests of DF we decided to adapt their approach with some modification. In contrast to their approach of using webhooks, we used a custom frontend with WebSockets. An introduction on how DF works is given in Section 2.6.2.

As mentioned in Section 5.4.2 the DF integrations induce overhead through webhooks. Therefore, we decided to use WebSockets in favor of performance and better response times. The Django framework provides the asynchronous WebSocket functionality through the channels[9] library. A WebSocket connection connects the server and the client with a single Transmission Control Protocol (TCP) connection over a whole session. Either server or client can send and receive TCP packets independently. This bidirectional communication is therefore not bound to the request-reply pattern used in HTTP.

Listing 5.1 shows the contents of a WebSocket that is sent from the client to the server. This message is sent to the backend when the user types in the message “Select architecture Hotrod” into the chat, or the architecture is selected in the configuration view. The content of the WebSocket message contains multiple fields. The Universal Unique Identifier (UUID) is used to uniquely identify the session of a user. The *type* helps to specify if the message is a text or an event. In the *data* field, the

[8] https://github.com/Cambio-Project/hazard-elicitation/
[9] https://github.com/django/channels

Listing 5.2 Contents of a websocket communication from the backend to the frontend.

```
{
  "type": "dialogflow_response",
  "data": [{
    "type": "text",
    "payload": {"text": "Please specify the name of the artifact."}
  }, {
    "type": "quick_reply",
    "payload": {
      "values": [
        {"text": "frontend", "action": "event", "values": ["set-artifact", "frontend"]},
        {"text": "backend", "action": "event", "values": ["set-artifact", "backend"]}
      ]}
  }]
}
```

content of the text or the name of the event is stored. Finally, in the optional field *contexts* a list of DF contexts is stored. DF contexts are used to steer the conversation between user and chatbot based on previous inputs.

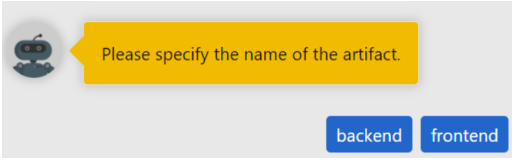


Figure 5.5: Visualization of the message send from the backend to the *Client Interface* in Listing 5.2.

In Listing 5.2 the contents of a response WebSocket message, from the server to the client, is given. The response message contains the *type* of the response that defines the name of the response handler (further explain in Section 5.4.4) and the *data* for the handler. In the example given in Listing 5.2, the user would be provided with a chatbot text and two *Quick Replies* as depicted in Figure 5.5. Examples of the response message data types are given in Section 5.4.2.

As we decided to produce chatbot responses in the backend, we had to use the DF client library[10]. It is available in different programming languages, i.e., JavaScript and Python. Since the architecture extraction backend is also implemented in Python, we decided to use the DF client library for Python. The library supports the same functionality used in the official DF UI, such as managing intents and training phrases by adding new ones or removing unused ones. Furthermore, the DF agent can be trained on-demand, and sessions of users can be managed. The library also contains all the beta features from which some are not present in the UI. We used only the intent detection, event detection, and context creation functionalities in the backend. On the one hand, this eases the creation of the response messages from the chatbot. On the other hand, additional contents such as

[10] https://googleapis.dev/python/dialogflow/0.7.0/index.html

properties of the architecture graph can be used in the responses. In theory, it is possible to create intents, managing entities, or add entities with the DF client. However, this would induce further management of modules. We think that this could be an area for future work.

One example of a beta feature is the knowledge base that is used in the chatbot backend. A knowledge base can be used as key-value storage of questions and resulting answers. The content of the knowledge base is stored in a comma-separated file, which makes it easy to update questions and provides the possibility to generate questions automatically. The ultimate purpose of the knowledge base is to reduce the effort to create new intents for every question there might be. However, using a knowledge base also comes with a trade-off. When the chatbot users ask a question that is part of a particular intent but might also be present in the knowledge base, it gets matched against the content in the knowledge base. One solution for this problem is to reduce the priority of the knowledge base questions in favor of the intent defined in the UI. The DF UI provides a setting for this that can be specified between -1.0 (prefer the intent) and 1.0 (prefer the knowledge base). Here, the chatbot developer must find a compromise between preciseness of intents and maintainability of the knowledge base.

As described before, DF is part of the Google Cloud platform and therefore requires a Google Cloud account. Just as the DF UI any library that uses the DF API is required to provide a JSON Web Token. The fastest way to use the JWT is to include it in the system environment. A program using DF can then load it at the startup procedure.

Conversation Flow

Resulting from the decision to create a custom integration for DF, it was necessary to generate responses in the backend as well. Although the DF UI allows for the creation of response texts, it is not possible to create meaningful phrases with variable parameters. Therefore, the DF agent is only used to detect the intent via the DF API. The API returns the name of the detected intent and the entities present in the input sentence. In the chatbot backend, the name of the intent is matched with the corresponding intent handler. An intent handler is a Python function that takes the result from the DF agent and uses it to extract intent and context parameters to create a response for the chatbot frontend.

In the conversation with the chatbot, users have a variety of possibilities to interact with the UI or the chatbot. In the following, the focus lies on the interaction of the user with the chatbot. The triggering of intents, events, and the involvement of the intent handlers that are part of eliciting a scenario, are highlighted. In Figure 5.6 an exemplary conversation between a user and the chatbot backend is given. Once the *Client Interface* is open, a WebSocket connection is established with the backend. After the first message to the backend, a DF client instance is created that manages the traffic between the backend and the DF agent. A conversation with the chatbot always starts with the welcome event triggered by the UI. The content of such an event trigger is shown in Listing 5.1. Every message processed by the DF client returns a response containing the name of the matched event. In the backend, the matching intent handler is chosen for the intent. Like event handling, text input detection also returns the name of the matched intent and delivers the parameters detected in the user text.

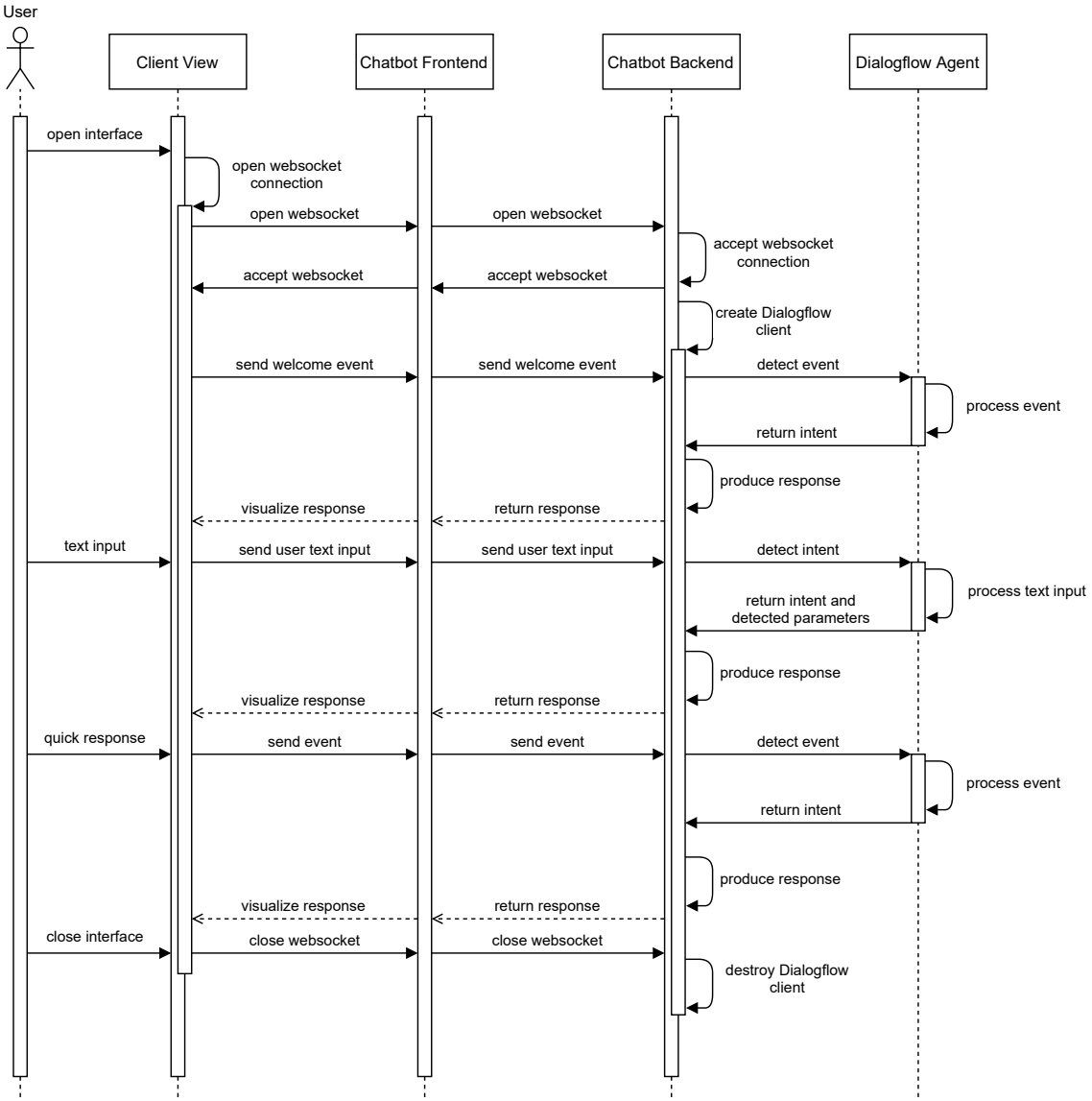


Figure 5.6: Conversation flow of intents between user and chatbot.

One drawback of the split design of client and agent is that the names of the intents have to be consistent. A change of an intent name in the DF UI breaks the functionality of the backend if it is not updated. It requires manual configuration and is error-prone. Future work for this prototype should shift the management of intents and training phrases from the DF UI into the backend.

Listing 5.3 shows the *guide_handler* that generates the visualization shown in Figure 5.7. The *guide_handler* is a supportive feature of the chatbot and provides explanations about a particular topic to the users. This feature uses the previously mentioned knowledge base to detect specific keywords. The input parameter *result* of the method contains the result of the API call. First, a list is created that contains all elements of the conversation. This list is ordered, and elements of it are rendered in the stored order. Secondly, a collection of *Quick Replies* is generated. The names of these options are stored in the backend. If a user clicks on it, the knowledge base detects

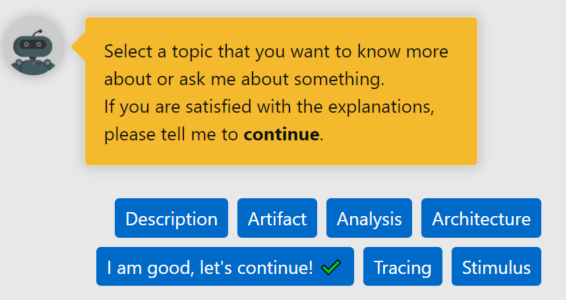


Figure 5.7: Guide handler visualization in the frontend.

the corresponding item and returns the explanation. Additionally, the collection of *Quick Replies* is extended with an extra button that leads the user back to the last unspecified parameter of the scenario. Thirdly, the context parameters of the intent are checked if the intent was previously called with a click on one of the *Quick Replies*. If this is the case, then the conversation list is extended with an *Card Response*, containing the explanation of the keyword. Finally, the conversation list is transformed into the JSON format of the chatbot frontend. This can be done implicitly by the call of the generator methods (*create()*) or explicitly with *__repr__()*. Is the conversation converted into the JSON format, it can be returned to the clients and is rendered on their screen.

Figure 5.9 gives an overview of all the intents that are part of the elicitation process and how they are connected. As described in Chapter 4, a resilience scenario consists of eight parameters. Every scenario parameter is mapped against one intent where the parameter's value is extracted from the user text. One exception is the response measure which is split into two intents for service artifacts and three for operation artifacts.

For every parameter of the scenario, the users are given the same chat visualization. At the top, the number of the parameter and a short summary of it is given as a *Card Response*. This is followed by an optional response of the chatbot and *Quick Replies*.

An example for the configuration of the artifact is given in Figure 5.8. The users have three different possibilities to configure the value. The first option is to use a *Quick Reply* which sends an event to the backend with the value of the presented text. The second option is to write the value in the chat where it is analyzed by the DF agent. The third option is to use a default which can be achieved by writing a specific text to the chatbot (i.e., "skip this", "use a default"). The advantage of choosing the first option is that the correct value is always set, and the following intent is triggered in a particular order. As shown in Figure 5.9, these three alternatives are available to any intent that specified a parameter of the scenario. Other intents, such as the *Guide*, *Save Scenario*, and *Next Step* intents do not provide a default value and require explicit input from the users. The *Save Scenario* intent also provides an additional functionality. This intent presents the users with a summary of the scenario and lets them choose to save the scenario or change any scenario parameter that does not match the users' preference. The *Next Step* intent lets the users choose between the completion of the elicitation process or the configuration of another scenario. Besides, every other intent except *Next Step* can be triggered by an external event. The trigger for such an event can be to write the exact name of the intent to the chatbot (i.e., "step 2" for selecting a different artifact) or calling an *event* trigger in the backend. A developer of the chatbot can use the WebSocket function *event* which requires the name of the event as a parameter. Once this function is called, the corresponding chatbot intent is called in the DF agent, and the intent handler is executed in the backend. One

Listing 5.3 The intent handler for the guide.

```
async def guide_handler(result) -> List[Dict]:
    """
    Presents the users with quick reply topics, from which they can get explanations.
    @param result: Contains the DF agent response.
    """
    # 1. The conversation contains the response types in the order they should appear.
    conversation = []

    # 2. Add quick reply options for the user to ask.
    quick_reply = QuickReplyResponse()
    for option in text(INTENT_GUIDE_OPTIONS):
        # If clicked by the user, the quick reply button calls this intent handler again
        # and adds the explanation for this option. The click also adds the 'quick-option'
        # parameter to the 'c-guide-option' context, which is checked further below.
        quick_reply.add_reply(option, 'event', ['e-guide', [{
            'name':       'c-guide-option',
            'lifespan':   1,
            'parameters': {'quick-option': option}
        }]])

    # Add option to allow the user to return to the last configuration step.
    # The 'next_event' function detects what parameters are missing in the scenario.
    continue_text = text(INTENT_GUIDE_CONTINUE_CONFIRM_TEXT)
    quick_reply.add_reply(continue_text, 'event', [next_event(result)])

    # 3. Check if an option was selected by the user.
    option_context = get_context('c-guide-option', result)
    if is_in_context('quick-option', option_context):
        option_value = option_context.parameters['quick-option']
    elif is_in_context('option', result.query_result):
        option_value = result.query_result.parameters['option']
    else:
        option_value = None

    conversation.append(FormattingResponse.create('divider'))

    if option_value:
        # If an option is given, explain it and add further quick reply options.
        explanation = text(INTENT_GUIDE_EXPLANATIONS)[option_value]
        guide_text = random_text(INTENT_GUIDE_CONTINUE_TEXT)
        conversation.append(CardResponse.create(**explanation))
        conversation.append(TextResponse.create(guide_text))
    else:
        # Otherwise tell the user to ask something.
        conversation.append(TextResponse.create(text(INTENT_GUIDE_TEXT)))

    # 4. The '__repr__' method converts the response type into a format that the chatbot
    # frontend can understand.
    conversation.append(quick_reply.__repr__())

    return conversation

```

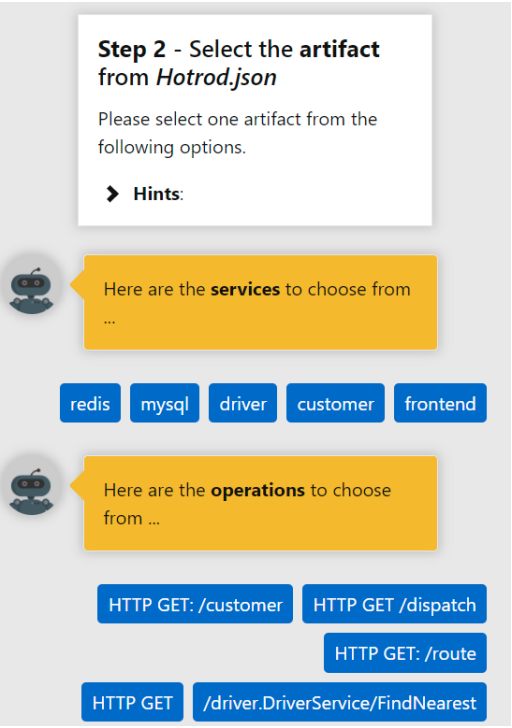


Figure 5.8: Intent for the artifact selection with *Quick Replies* given by the chatbot.

example for calling an event in the backend is the selection of an artifact in the architecture graph. If an artifact in the architecture graph is selected, its name and artifact type (node, or edge) is encoded in a DF context object and send to the intent handler in the backend. The intent handler responsible for the artifact selection, extracts the context and uses the artifact name and type in the response. The created response is sent back to the user who sees it in the chat window.

As mentioned before, the parameters of the scenarios are stored in the contexts of the chatbot. Additionally, more contexts exist to store information about the architecture graph and configurations during the elicitation process. In the following, the focus lies on the elicitation context. Figure 5.10 depicts the relationships of the different contexts. Every intent of the chatbot has an incoming and an outgoing context, which are both optional. An incoming context means that the user can only trigger the intent if the context was set before. An outgoing context means that the intent produces the context or modifies the contents of it. This way, the conversation flow can be steered implicitly without any intent handlers. If the intent contains entities that are recognized from the user input, the entity name and value are inserted into the context. In the intent handler, both entity name and entity value can be extracted. An example is given in Listing 5.3 at line 27. The context is extracted from the DF client result and queried for parameters. Since not all information might be present from the user input to create a scenario, further contexts can be injected artificially by calls from the WebSocket or the intent handlers. The previously given example utilizes this methodology. A click on a node in the architecture graph triggers an *event* in the backend and injects the name of the node as a context parameter. This name is then put into the elicitation context, where it is stored as the artifact.

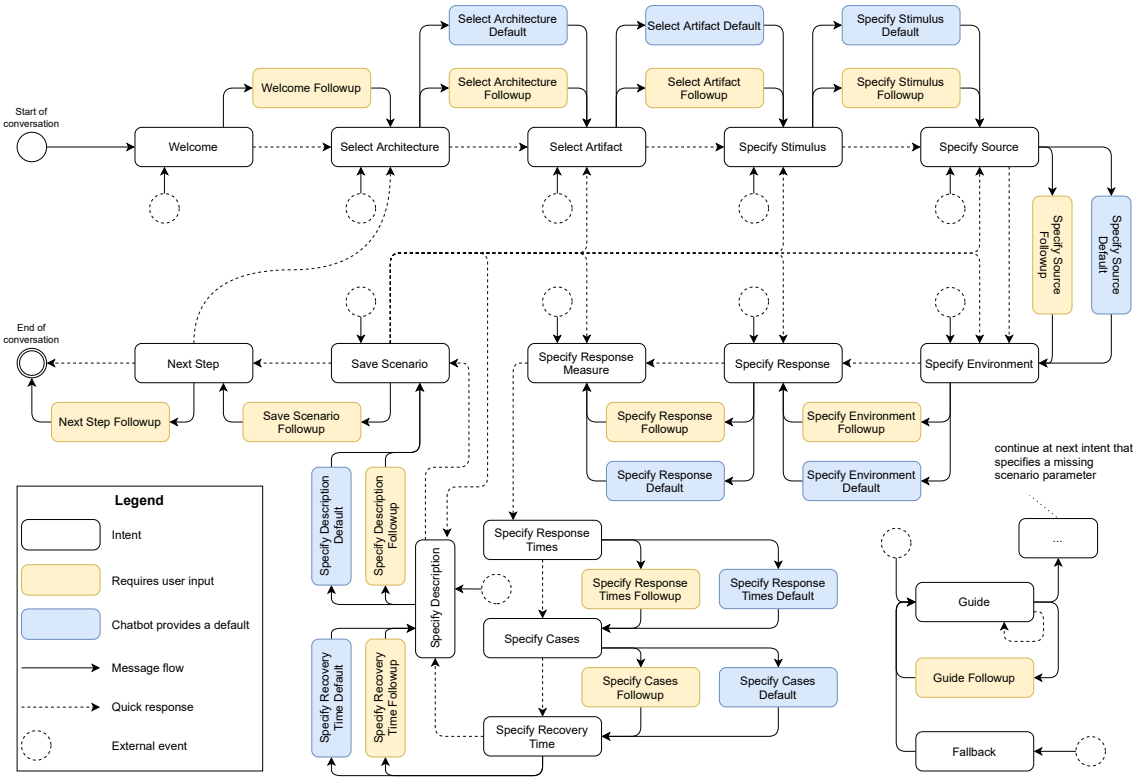


Figure 5.9: Conversation flow between user and chatbot.

Line 15 in Listing 5.3 demonstrates this behavior. As described before, this line of code creates a *Quick Reply* button that returns a *Card Response* with explanations and calls the same intent handler again after it is clicked. The last parameter in this call contains an array of a context object for the context *c-guide-option*. In the second call of the same intent handler, the *quick-option* parameter is extracted from the context and used for the response creation, as shown in Line 29.

One significant advantage of storing the scenario parameters in the chatbot context is that the chatbot developer always knows which intent should be called next inside an intent handler. For this reason, every intent handler is equipped with the *next_event* function that checks for missing scenario parameters and calls the intent handler for the next missing parameter. As a result, if a parameter configuration is missing before the scenario summary is shown, the users first have to configure the missing parameter. Furthermore, the values of the context parameters are also used to create valuable responses to the users. This is demonstrated in the *Card Response* that shows the current step number. As depicted in Figure 5.8, the users are given the name of the current architecture in the heading of the *Card Response*. This way, users can immediately recognize for which architecture, artifact, or stimulus they compose the scenario.

Besides the elicitation context, the prototype stores the architecture graph in a different context. On the one side, this helps to make suggestions to the users, based on the artifacts' priority. On the other side, the user input during the artifact specification can be checked against the artifacts in the architecture graph.

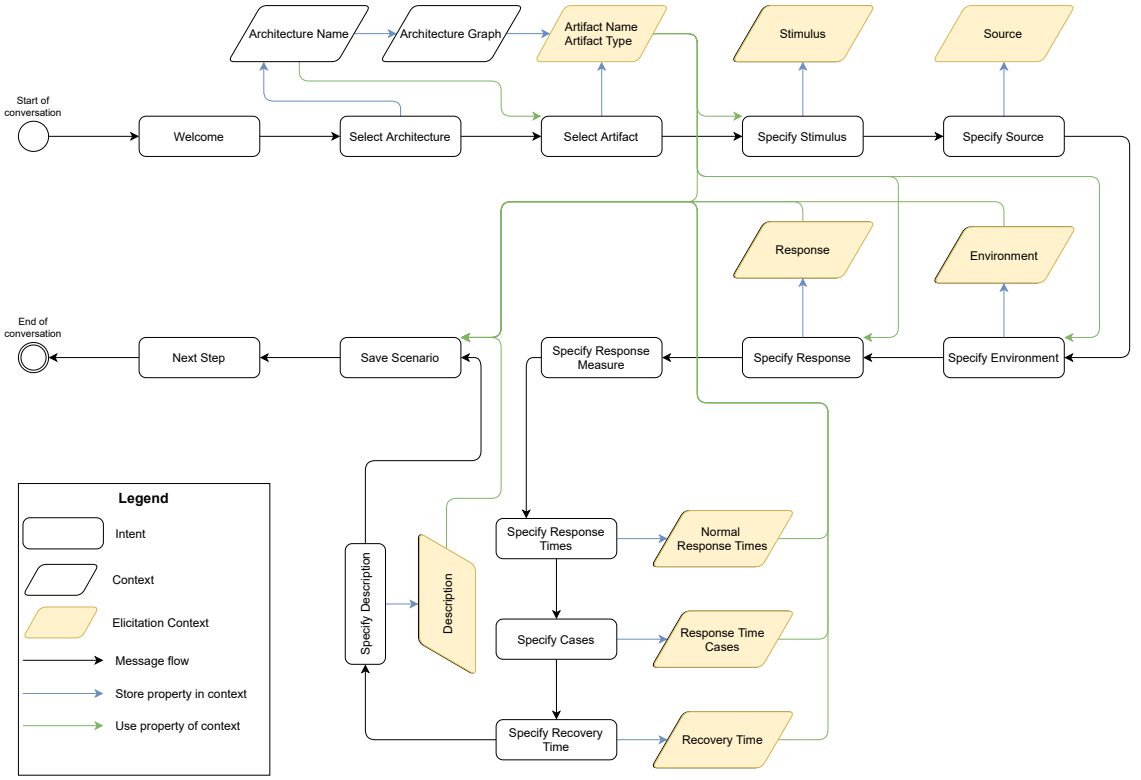


Figure 5.10: Context properties that are shared between intents.

5.5 Summary

Previous sections explained the two components, *Client Interface* and the *Architecture & Analysis* in detail. In the following, we give a running example of how the prototype is used. Screenshots of the prototype that are referenced in this section are given in Appendix B.

Users are first welcomed by the chatbot when they open the prototype, as depicted in Figure B.1. The chatbot gives a short introduction by explaining the 7-step procedure to create a scenario. The process can be started by writing a confirmation into the chat (i.e., “ok”, “go”, “yes”) or clicking on the *Quick Reply* (“Yes let’s go!”).

After the users have started the conversation, they are prompted to select a trace model. It is also possible to upload trace models via the upload view shown in Figure B.2. Is the trace model stored in the database, users can either load it with the combo box in the configuration view or tell the chatbot the name of the architecture.

Is a trace model chosen, its architecture is visualized in the architecture graph as shown in Figure B.4. From now on, the chatbot is steering the conversation. The users are prompted with *Card Responses* to specify the parameters of the resilience scenario. It is possible to specify the parameter from the suggestions of the chatbot through *Quick Replies* or text input that the users define. The chatbot provides hints to the users to give an impression of how parameters could look. Inexperienced users can inspire themselves through these hints. The following steps function the same way as the artifact selection works. The chatbot provides a short overview of the current parameters and a

limited amount of *Quick Replies*. The users can then choose a default provided by the chatbot, or provide a custom text response. Figure B.3a shows the configuration of the source, environment, and response. Figure B.3b prompts users with the response measure. In Figure B.3c the summary of the scenario is shown. If the scenario contains parameters the users did not intend to specify, they can use the *Quick Replies* for the appropriate action and change these parameters.

Is the scenario saved, it appears in the configuration view in a separate tab. Each parameter is visualized with the corresponding key-value pair in the tab. Furthermore, the scenario tab is equipped with an export button that downloads the scenario in the JSON format. The architecture graph also highlights artifacts that are part of a scenario. In the end, users can decide to create another scenario and repeat the process or end the conversation with the chatbot.

6 Evaluation

In this chapter, Resirio, introduced in Chapter 5, is evaluated. First, the objective of the evaluation is explained in Section 6.1. In Section 6.2 the methodologies of the evaluation are introduced. Section 6.3 focuses on the design of an expert user study that was used to collect metrics for evaluating the approach. Section 6.4 describes the execution of the study and Section 6.5 summarizes the results of the evaluation. In Section 6.6 the results of the evaluation are interpreted and discussed. Finally, in Section 6.7 the threats to validity are given.

6.1 Goals

To evaluate the developed prototype on usability and effectiveness, we designed an expert user study. The user study aimed to get feedback from inexperienced engineers and experts in the area of resilience engineering. For each aspect that the developed tool provides, a separate research question is proposed. The four research questions are:

- **RQ1** Are users of Resirio able to create resilience scenarios successfully?
- **RQ2** How effective is Resirio in contrast to the traditional elicitation process?
- **RQ3** How supportive is Resirio during the elicitation process?
- **RQ4** How usable are ATAM-based scenarios created with Resirio for resilience specifications?

RQ1 focuses on the overall functionality of Resirio. It is tested whether users of the prototype can create ATAM-based scenarios in a conversation with the chatbot. Resirio is capable of recording user input and elicited scenarios during the elicitation process. These two metrics help verify if the user study participants could create ATAM-based scenarios. In **RQ2** we emphasize how effective the presented prototype is in contrast to the traditional elicitation process of group meetings. The number of interactions during the conversation and the duration of the study is used to evaluate the effectiveness. **RQ3** covers the supportive features of the prototype. On the one hand, the use of the chatbot guide is reviewed. On the other hand, the use of the architecture graph and the configuration view is inspected. Furthermore, we use feedback collected from the System Usability Scale (SUS) [BKM08] to identify the prototype's usability. **RQ4** aims to evaluate the quality of the elicited ATAM-based scenarios.

We define the following hypotheses for each research question:

- **RQ1**
  - H_{10} Study participants can not complete the study tasks successfully.
  - H_{11} Study participants can complete the study tasks successfully.

Contents of scenarios created by the study participants can easily be verified on completeness and correctness. However, besides the successful creation of a ATAM-based scenario, it is interesting to see how study participants create scenarios. The number of created scenarios during the study can give insights on how many attempts a user needed to complete the given tasks.

- **RQ2**
  - H_{20} Study participants create ATAM-based scenarios faster with the traditional approach than with Resirio.
  - H_{21} Study participants create ATAM-based scenarios faster with Resirio than with the traditional approach.

  While this work does not evaluate the effectiveness of the traditional approach, we rely on the findings of other academic works [BCK03; HT11; KA00; KKB+98]. H_{21} is expected to prove that less effort is needed with the presented approach in contrast to the traditional approach.

- **RQ3**
  - H_{300} Resirio does not support non-experienced resilience engineers in the elicitation process.
  - H_{301} Resirio supports non-experienced resilience engineers in the elicitation process.
  - H_{310} Resirio does not support experienced resilience engineers in the elicitation process.
  - H_{311} Resirio supports experienced resilience engineers in the elicitation process.

  The content of **RQ3** can be split into two parts. Since knowledge about resilience engineering plays a crucial role for the interaction with the prototype, we distinct between novices and expert users. An expert has previously been involved in a requirements elicitation process and has experience with resilience testing. Novices, on the other hand, may be familiar with resilience testing but have limited experience with requirements elicitation. Inexperienced users may require a different level of assistance than expert users. Expert users are expected to want more insights into metrics of the inspected software system. It is assumed that novice users are most likely to require more assistance in using the prototype rather than getting system-level insights.

- **RQ4**
  - H_{40} ATAM-based scenarios produced by Resirio can not be used for system resilience specifications.
  - H_{41} ATAM-based scenarios produced by Resirio can be used for system resilience specifications.

  RQ4 provides the biggest challenge for the evaluation. While comparing the effectiveness between the traditional approach and Resirio relies on existing research, the quality of elicited scenarios is even harder to compare. We rely on feedback from the participants who have practiced resilience testing before. Furthermore, expert feedback helps us identifying how resilience attributes elicited during the study can be used to create resilience specifications.

6.2 Methodology

As mentioned before, an expert user study was designed to evaluate the developed prototype. A qualitative study provides the best possible evaluation methodology since there is a limited number of domain experts for scenario-based resilience engineering. In qualitative research studies, it is common practice to focus on a small number of participants and make no or only a few statements about the generalizability of the evaluated work [Max08]. An expert study provides an open setting and room for discussion between participants and the study conductor. The study was designed to have three different strategies to collect feedback. First, subjective feedback is collected through interview-style and open-ended questions. Secondly, the prototype will measure metrics like the duration of the study and the user input. Finally, feedback about usability is measured with the SUS. How these three strategies are applied is emphasized in Section 6.3.

As the developed prototype is based around a user interface, it is hard to evaluate the usability of measurements. Therefore, we used the System Usability Scale developed by Brooke in 1996. Usability can not be defined in a general context and measured in absolute values [Bro96]. It is perceived differently through factors such as subjective reactions. Brooke suggests that general questions about usability based on the Likert scale can provide values for effectiveness, efficiency, and satisfaction [Bro96]. However, results from these measurements can vary and are dependant on the given tasks for users. Nielsen et al. [NL94] claim that besides subjective feedback, usability parameters exist that can be measured. Furthermore, every interactive system implicitly provides objective measurements that can be used to correlate the subjective feedback.

During the practical part of the study, the prototype collects metrics from the participant's interaction with the chatbot. Collected metrics include the number and type of interactions, interactions with features besides the chatbot, and the number of elicited scenarios. To support the measurement of effectiveness and the quality of the prototype, we included close-ended questions. Some of them can be answered with either "Yes" or "No". Others provided specific fixed responses. Close-ended questions provide the possibility to find common ground between study participants in contrast to open-ended questions. Visualization of the collected metrics combined with the close-ended questions provide clear feedback, where participants put their focus.

In addition to the SUS, interview-style questions were asked to evaluate the support of Resirio. These interview-style questions were a mixture of questions asked after the practical part of the study. Some questions were asked in a questionnaire, while other open-ended questions were asked spontaneous by the study conductor. As the feedback discussion differed with every participant, the content and order of the spontaneous questions did not follow a previously defined structure. Furthermore, due to time constraints, it was sometimes not possible to ask all questions. As Peterson lays out, a direct personal discussion with participants in a small study setting can produces more valuable feedback than numerical data [Pet19]. If participants had a particular interest in a feature of the prototype or made suggestions for improvements, the study conductor made up followup questions to get more insights. Again, these questions differed between participants.

6.3 Design

This section describes the design of the study that was used for the evaluation. In Section 6.3.1 the description of the study tasks is given. Section 6.3.2 emphasizes the design of the questionnaire.

6.3.1 Study Tasks

Two tasks were created to lead participants through the use of the prototype. Both tasks differed in difficulty. Participants took the role of a software architect who was responsible to ensure the quality of a microservice-based software system. At the beginning of each task, the participants were given a general description of the inspected system and its capabilities. Each task description introduced a new failure to the system. The solution for every task was to create an ATAM-based scenario that could create a specification to fix the failure.

In the first task, which was considered beginner-friendly, we prepared a fixed solution for the scenario. Participants could see the solution. The intention behind this was for participants to familiarize themselves with the prototype. The only challenge in this task was to create the scenario through the use of the chatbot. However, in the first task, it was possible to create the resilience scenario only with the use of *Quick Replies* and without the use of text input. The second task was considered to be more challenging. On the one hand, the correct solution was not shown to the participants. On the other hand, the task description contained names of services and operations that were not available through *Quick Replies*. The absence of *Quick Replies* should encourage participants to use other features of the prototype, namely the architecture graph and the text input.

6.3.2 Questionnaire

The questionnaire for this evaluation was used to collect measurements for a variety of topics. Therefore, the questionnaire is split into four parts. The questionnaire can be found in Appendix A.4. The first part of the questionnaire was used to collect demographic information, such as the job description of the participants, their experience with resilience engineering, and background in computer science. On the one hand, this information was used to determine the experience level of the participant. On the other hand, we wanted to see how experienced participants were with the requirements elicitation process. The second part consisted of questions about the study tasks and the creation of resilience scenarios. Of particular interest was how participants would rate the difficulty of the study tasks and how an interactive solution could assist them. In the third part, the ten SUS questions were asked to evaluate the system's usability. These questions used the Likert scale from one to five to describe the usage aspects of the prototype. The last part contained questions about Resirio's features and future use of the prototype. Here we were interested in the participants' experience with the chatbot and how functional the prototype's features are. In a wrap-up section, we asked participants for input about other aspects that were not covered by the previous parts.

6.4 Execution

We invited software architects, requirements engineers, developers, and researchers from the industry and academia for the evaluation. Previous works by Kesim et al. [KWK+21] performed a workshop at an industry partner where the application of chaos experiments in an industrial setting was inspected. As the approach presented in this work is built on the experience collected during that project, we contacted the responsible people and established contact to the DATEV[1] company. After discussing potential participation in the evaluation of this work, the contact asked for volunteer participants in the company. One of the microservice-based systems tested in this evaluation was also inspected by Kesim et al. [KWK+21], as part of a case study. As the system contains sensitive data, details are not disclosed in this thesis. In the following, we call it the industry system. However, other works have used a Mockup of the industry system that shares some similarities with the original system [Bec20; KWK+21].

We made sure that volunteers of the DATEV company had at least some knowledge about the industry system and a background in resilience engineering. Furthermore, volunteers had to have taken part in a requirements process before this study. Five participants from the DATEV company did take part in the study, of which three people are considered experts and two as inexperienced. The experts were either responsible for the architectural design of the tested system or were directly involved in the development and testing of the system. The less experienced engineers were unfamiliar with the system or did not have any prior contact with the system.

The second tested system, TrainTicket, was selected based on current research from Zhou et al. [ZPX+19]. TrainTicket shares similarities with the industry system, such as the number of services and the number of operations. Therefore, it is suitable for comparison. Figure 6.1 depicts the architecture of the TrainTicket system and relationships between the services. Similar to our contact from the industry, we asked the research group responsible for creating the TrainTicket system for volunteer participation. As it was desirable to have the same amount of participants for each microservice-based system, we asked the research group to participate with five volunteers. The research group was happy to participate, and provided four volunteer participants. In addition to the four volunteers who were considered experts, three researchers from other academic research groups participated. These three participants had previously used the TrainTicket, but as they were not part of the development process, we considered them to be less experienced in the system.

After contact details were exchanged and suitable study dates were found, the study setting was prepared. We informed potential participants beforehand that the study would take approximately 45 minutes to one hour to complete. Choosing this amount of time for the study had several benefits. First, engineers from the industry could find a suitable timeslot in their schedules. Secondly, this amount allows participants to keep their concentration. Finally, it is assumed that it would be much harder to find participants for longer study durations.

Volunteers who agreed to participate were sent the consent form and official invitation, which contained the procedure plan. In the consent form, which can be found in Appendix A.2 the participants were informed about their rights before, during, and after the user study. Since the meeting sessions

[1] https://www.datev.de/

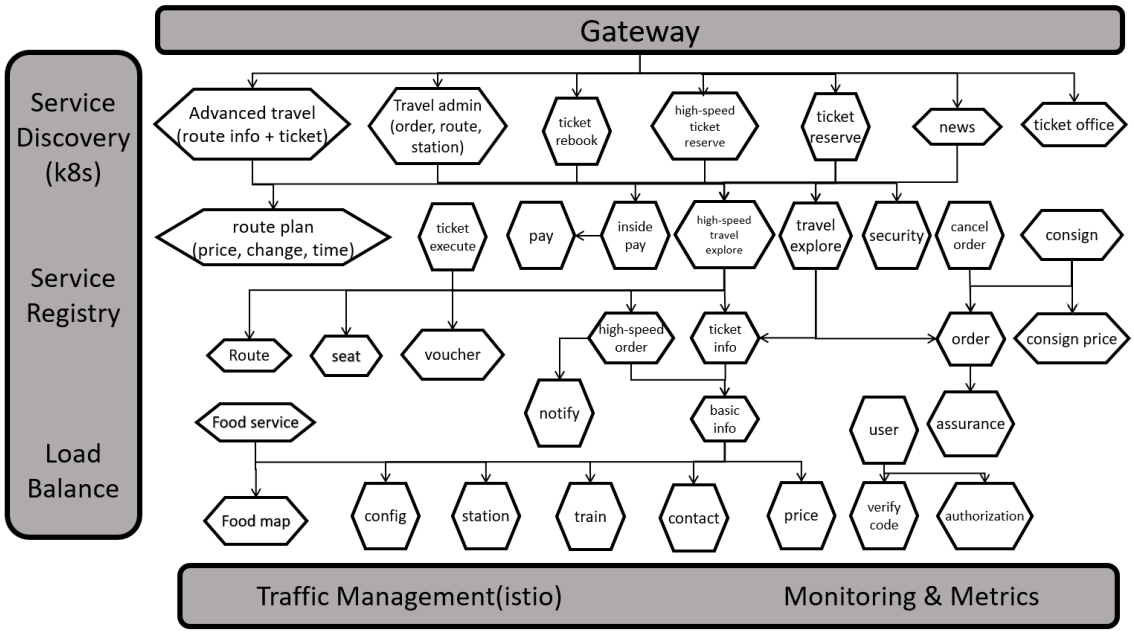


Figure 6.1: Architecture of the TrainTicket system [ZPX+19].

were recorded and interactions with the chatbot were stored in a database, participants had the chance to ask for the immediate deletion of both, after the study. Furthermore, if participants would feel sick or not agree to the given terms anymore, they could stop the user study.

The official invitation is given in Appendix A.1. It contains a short motivation about the topic and why the participation of the volunteers was meaningful. Furthermore, it gave the conditions necessary to participate in the study. These were a display size of at least 1600x900 and a modern browser such as Google Chrome, Mozilla Firefox, or Safari. The invitation also stated the use of the Cisco Webex[2] meeting software for the study meeting. Finally, the procedure plan was given that contained an overview of the three study parts.

The study meeting went the same way for every participant. The study conductor opened the meeting and waited for the individual participant to join. The study conductor used a webcam to create a more relaxed setting for the participant. Not every participant used a webcam, but we do not consider that this would create a disadvantage or influence the study's outcome. After questioning the participants' well-being and their verbal agreement to the consent form, they were sent the task description. Since two MSAs were used in the study, the task description was different for the respective participants.

The full task description can be found in Appendix A.3. It is separated into three parts. In the first part, the goal and purpose of the study are given. In the second part, the prototype is introduced. The third part contains the actual tasks. After the participants read the first two parts of the task description, they were given a link to the prototype, which initiated the practical part of the study. Participants had to share their screen, and the study conductor started recording. The study conductor

[2] https://www.webex.com

informed them that he would stop the time and give them a hint if they would exceed 15 minutes. The purpose of this was to keep the study duration within one hour. The consequences of introducing a time limit for the practical part are discussed in Section 6.7.2.

Communication during the practical part of the study was kept to a minimum to not influence the participants' actions. However, the participants were encouraged to describe their actions with the prototype, which was done in most cases. If confusion arose, the study conductor reminded the participants to carefully reread the task description, which resolved the problems. Furthermore, the participants were informed after each task if they had finished it.

The last part of the study was used for feedback. After completing the practical part, the study conductor started a verbal conversation with the participants and asked them about noticeable events during the practical part. Examples of noticeable events were not using the chatbot text interactions or creating a new scenario from the beginning. These questions usually led to a discussion with open-ended questions, which are emphasized in Section 6.2. If all noticeable events were covered, the study conductor asked the participants to fill out the questionnaire. Study participants could choose to fill out the questionnaire in private or if they wanted to do in in the presence of the study conductor. At the end of the study, the conductor thanked the participants and closed the meeting session. The study was conducted over the range of four weeks. During this time, neither the task description nor the prototype was changed to have equal conditions for every participant. Typographical or grammatical errors that were found during the study were corrected afterward.

6.5 Results

This section presents the result of the user study. First, we explain the used visualization techniques and the use of different types of diagrams in Section 6.5.1. Secondly, an overview of the participants' background and experience is provided in Section 6.5.2. After that, the results for each research question are given through visualization of collected metrics or qualitative feedback from participants.

6.5.1 Explanation of the Visualization

As described in Section 6.2, we used two different strategies to collect data from the study. On the one hand, the prototype collected measurements during the study, such as the number and duration of the interaction. On the other hand, the participants' feedback was collected through a questionnaire. Questionnaire results contain quantitative metrics, as well as qualitative. Questions or statements that produced the presented results are shown below the respective diagrams. For the quantitative results, we used different diagrams.

The first type of diagram is a boxplot, as shown in Figure 6.3a. An orange dot represents a sample of one participant. The lower end of the vertical line represents the minimum value, while the higher end of the line represents the maximum value. The lower end of the box represents the values of the first quartile, while the upper end represents the third quartile. The horizontal line represents the median.

The second type of diagram is a barchart as shown in Figure 6.2a, Figure 6.3c, Figure 6.3d, and Figure 6.7a. As the collected data varies in dimensions and categories, we used standard bar charts and stacked bar charts. Red lines were used to highlight a significant finding, while orange lines highlight the median. In cases of close-ended questions where participants did not choose an option, we omitted the visualization.

We use the non-suggestive color palette of LocusZoom[3]. This color palette is used in other scientific evaluations and is meant to avoid biased interpretation of results. Results from close-ended questions with two answer possibilities are visualized with green and blue. Questions with more than two answer possibilities use red, orange, green, cyan, blue, purple, and interpolated variations.

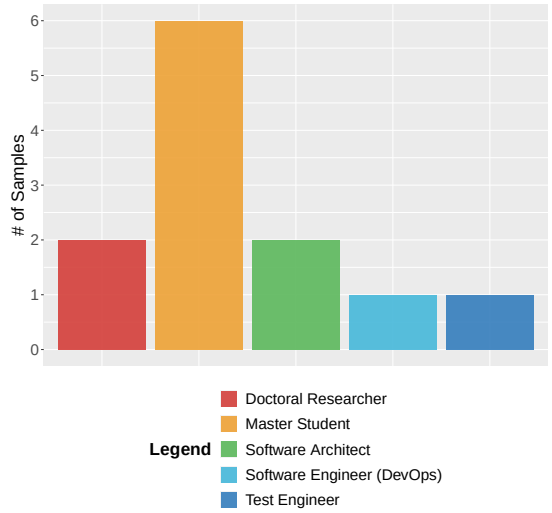
6.5.2 Participants' Background

In this section, the participants' background is described based on the information collected in the first part of the questionnaire.

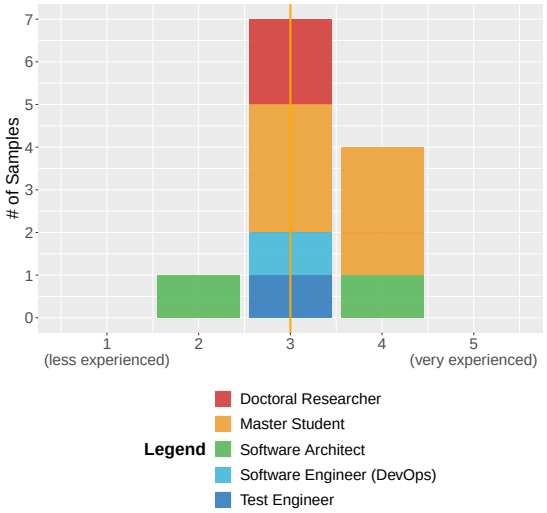
In total, twelve people participated in the user study. Five were considered to be experts while seven were considered inexperienced. One half of the participants were pursuing a master's degree in computer science or a related area. The other half was either employed in software engineering or pursuing a Ph.D. degree in computer science. Figure 6.2a shows the distribution of the participants' employment stats. Eight participants came from academia, while four were employed in the industry. Five people were inspecting the industry system, including two software architects, one DevOps engineer, one Test Engineer, and one doctoral researcher. Seven people inspected the TrainTicket system consisting of six master students and one doctoral researcher.

In Figure 6.2b the self-assigned experience levels of the participants are given. The Likert scale from one (less experienced) to five (very experienced) shows that the participants are decently experienced in resilience engineering. The median for the experience level is three. Eight participants had previous knowledge about resilience scenarios as depicted in Figure 6.2c. Four participants were already involved in an elicitation process that used resilience scenarios shown in Figure 6.2d.

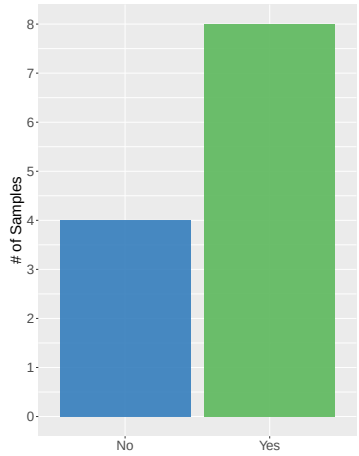
[3] http://locuszoom.org/



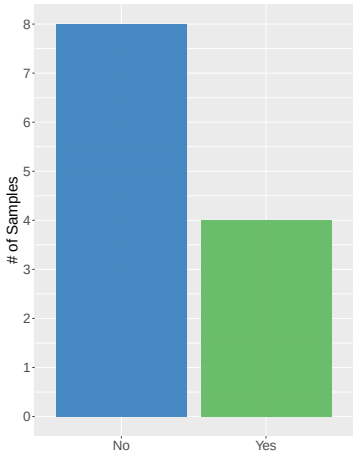
(a) What is your current role in your company or university?



(b) What is your level of confidence in resilience engineering?



(c) Did you know about resilience scenarios before this study?



(d) Have you ever been part of the elicitation process to create resilience scenarios before this study?

Figure 6.2: Background and experience of the participants.

6.5.3 RQ1: Are users of Resirio able to create resilience scenarios successfully?

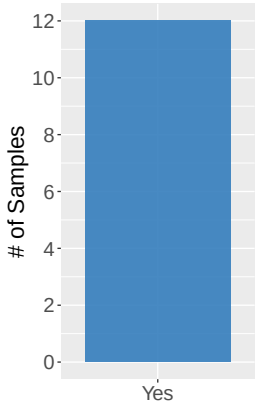
For the evaluation of **RQ1**, we use metrics from the prototype and questions of the questionnaire. Furthermore, the evaluation recordings are used to verify if participants created resilience scenarios based on the given tasks successfully. Figure 6.3 presents the metrics that were used for the evaluation.

Figure 6.3a shows that all participants believe that they completed their given tasks successfully. Since the first task was fixed for both evaluation settings, we can easily verify if their assumption is correct. The second task of the study was phrased openly and proposed no solution to the participants. However, we created a set of solutions for the second task before the study was executed. In the evaluation, we compared the elicited scenarios of the participants with the prepared solutions. For example, in the second task, we allowed the configuration of two different artifacts. In the case of the TrainTicket system either the service *ts-food-service* or the operation *createFoodOrder* were valid options for the artifact. For the industry system the service [redacted], or the operation [redacted] were allowed. In the second task, only one participant created a scenario with a configuration that was not valid.

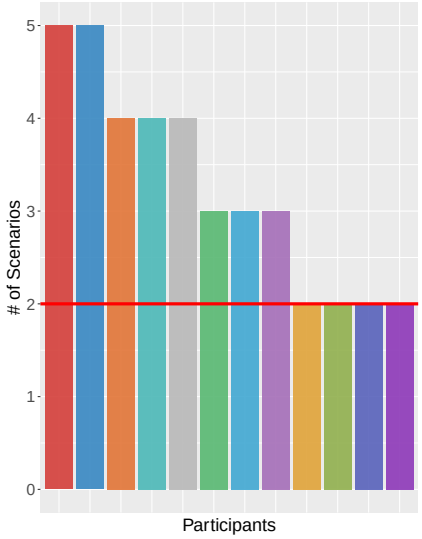
While all participants completed the given tasks, the study conductor noticed that one participant created a wrong scenario for the first task. Instead of the operation *getLeftTicketOfInterval* the service *ts-seat-service* was selected. The recording shows that this participant did complete the study the fastest. We assume that the participant only glanced over the task description of the first task without looking at the solution.

Figure 6.3b depicts the number of scenarios that each participant created. Every bar on the x-axis shows exactly one participant. The y-axis gives the number of scenarios that this participant did create. The red line highlights which participants created more than two scenarios. This is interesting because the maximum number of elicited scenarios should be two, one scenario for each task. A number higher than two allows the interpretation of two cases. The first interpretation is that the participant was not satisfied with the configured parameters. The second interpretation is that an error occurred during the creation of the scenario. Recordings of the scenario creation process show that five participants configured the wrong parameters for a scenario and changed them after seeing the summary. Three participants experienced problems while interacting with the chatbot or another feature of the prototype. Four participants created exactly two scenarios.

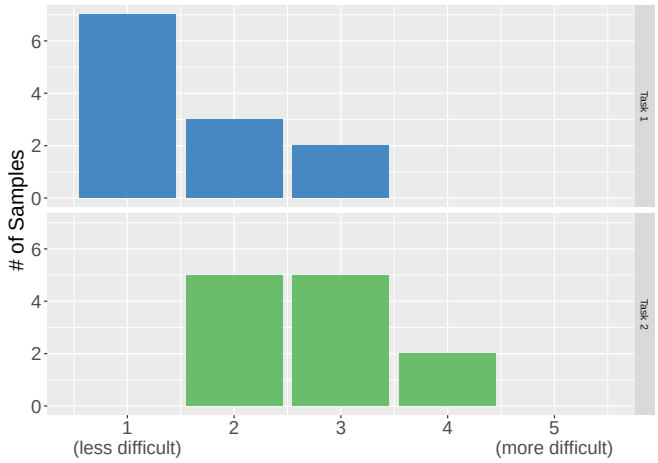
Figure 6.3c presents the participants' opinion on the complexity of the tasks. It confirms our intention behind the first task. The first task was meant to be beginner-friendly and easy to solve. Novice users had the chance to familiarize themselves with the chatbot and other features of the prototype instead of focusing on the contents of the task description. Although, as we mentioned before, it was crucial to read the task. Figure 6.3c shows the same data as depicted in Figure 6.3d but uses a different representation. In Figure 6.3d each sample is visible.



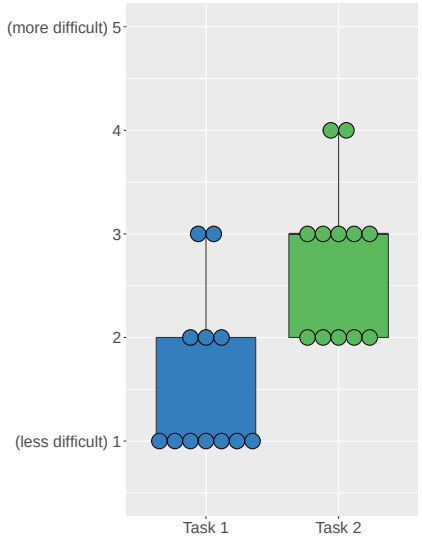
(a) Did you complete the given tasks successfully?



(b) Number of elicited scenarios during the study for each participant.



(c) How challenging was it to solve the tasks?



(d) How challenging was it to solve the tasks?

Figure 6.3: Measurements about the successful creation of a resilience scenario and the completion of the study tasks.

6.5.4 RQ2: How effective is Resirio in contrast to the traditional elicitation process?

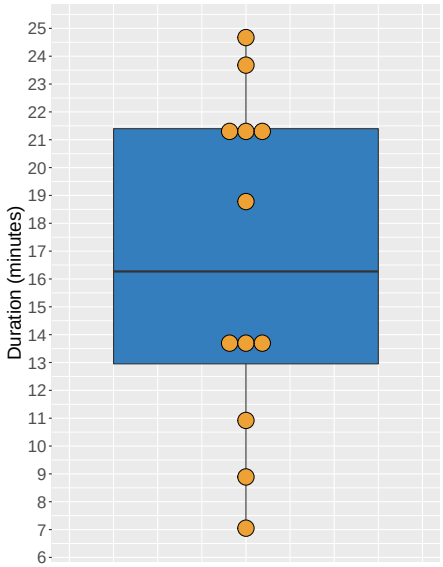
The comparison of the effectiveness of our approach in contrast to the traditional method of group meetings is limited to the available findings of other research as stated in Section 6.1. Furthermore, we rely on the expertise of the participants who took part in this study. Only four participants took part in an elicitation process that was using resilience scenarios. To measure the effectiveness of the scenario elicitation, we use the duration of creating a scenario and the number of interactions between participants and the chatbot.

Figure 6.4a shows the time it took participants to complete the practical part of the study. The fastest participant used seven minutes and three seconds to complete both tasks, while the slowest used 24 minutes and 38 seconds. The median lies at 16 minutes and 14 seconds.

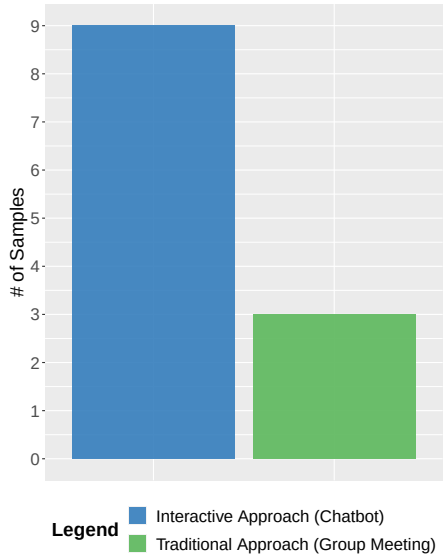
Figure 6.4b depicts the participants' opinion about the effectiveness of creating resilience scenarios with the traditional method of group meetings and the approach presented in this work. While we can not use the result of this question for the reasons mentioned before, it is interesting to see that most participants think that the interaction with the chatbot is faster than the traditional approach.

Figure 6.4c and Figure 6.4d show the same data in different representation. They show the number of interactions that were necessary to produce the solutions of both tasks. Since the measurements built into the prototype do not recognize for which task a solution was created, we can only make statements for the whole study. This does also hold for the duration of the study. Looking at the recordings, we can approximate both interactions and duration for either task. While participants reported that the second task was more complex than the first one (as depicted in Figure 6.3c and Figure 6.3d), they took less time than used in the first task. The median to solve the first task was approximately twelve minutes. The median to solve the second task was approximately five minutes. This can be ascribed to the learning effect that users had for solving the first task. As the solution for the first task was already given in the task description, the participants spent more time exploring the prototype and testing its features.

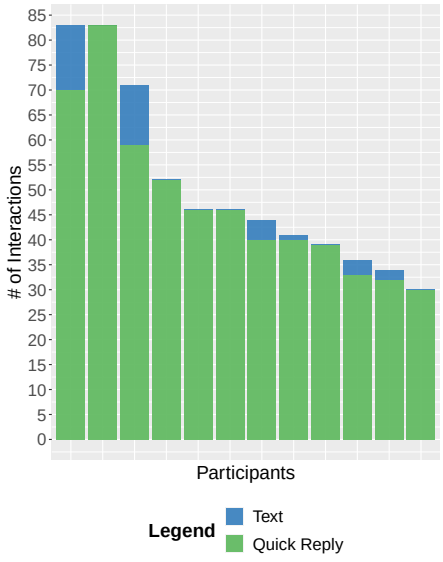
Another significant finding is the overall preference of using *Quick Replies* over the use of text interactions. While six participants did not use text interactions at all, the other six used only three to thirteen text interactions. The median of using *Quick Reply* lies at 43 interactions with a minimum of 32 and a maximum of 83. During the feedback round, the study conductor asked participants about the preferred use of *Quick Replies*. The arguments against using text interactions were "Text interactions take too long." and "The chatbot's text recognition is limited.".



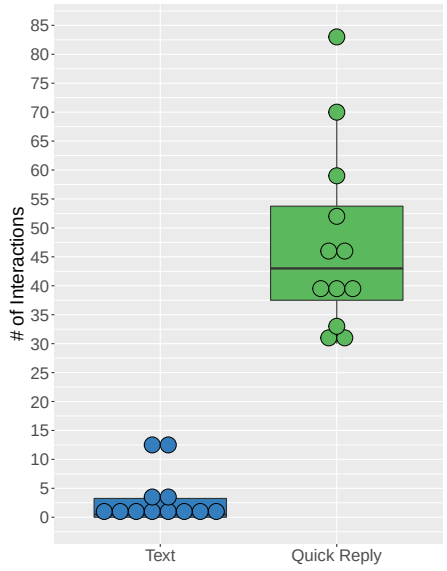
(a) Duration of the study.



(b) Which approach to elicit resilience scenarios is faster?



(c) Different type of interactions during the study.



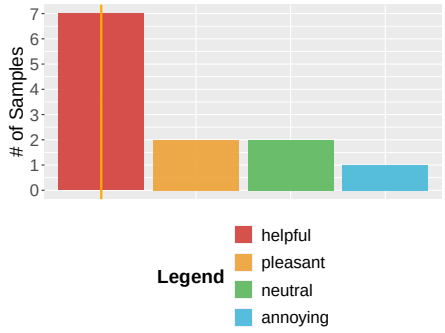
(d) Different type of interactions during the study.

Figure 6.4: Measurements about the effectiveness of the prototype.

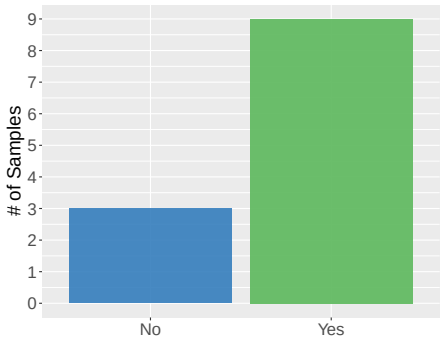
6.5.5 RQ3: How supportive is Resirio during the elicitation process?

For the evaluation of the chatbot's support, we used metrics collected from the third part of the questionnaire, qualitative responses from the feedback sessions, and the use of supportive features. The visualization of questions concerned with the prototype's support is given in Figure 6.5.

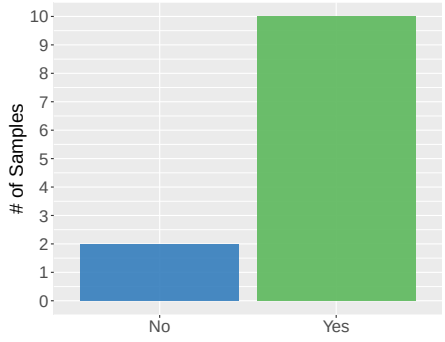
The first question in the fourth part of the questionnaire asked participants to describe the chatbot in one word. We provided two positive adjectives (helpful, pleasant), two negative adjectives (annoying, unpleasant), and a neutral option (neutral). Participants could also create a custom response. However, all participants used the provided options. Figure 6.5a shows the result of this question. Adjectives were ordered from positive, over neutral, to negative ones. The most used rating was *helpful* which also represents the median. Two ratings used *neutral* and *annoying* was used once.



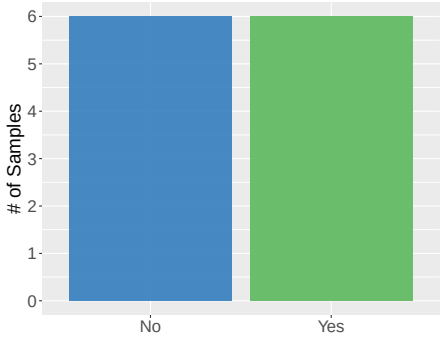
(a) Describe the chatbot or your interaction with the chatbot in one word.



(b) Does the chatbot provide enough assistance during the elicitation?



(c) Could you follow the flow of the conversation?



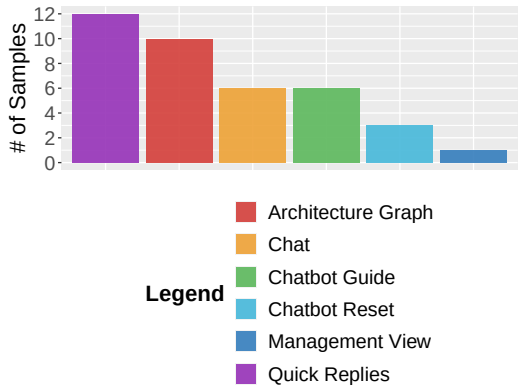
(d) Did you get stuck during any of the tasks?

Figure 6.5: Results from questions about the chatbot's support.

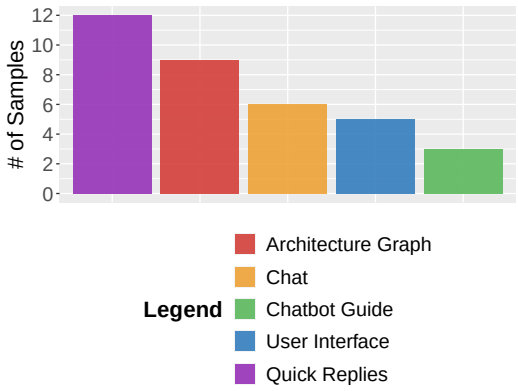
Figure 6.5b suggests that users of the chatbot were provided with enough assistance during the elicitation process. This either means that participants who answer the question with "Yes" did not need any help or that they were satisfied with the chatbot's supportive features. Figure 6.6a shows that only six participants used the supportive feature of the *Chatbot Guide*. We do not consider *Quick Replies* as a supportive feature, as they represent an essential part of the chatbot.

Figure 6.5c confirms that the conversation flow between the chatbot and a user was easy enough to follow. As presented in Figure 6.6a, three participants used the *Chatbot Reset* functionality, which suggests that they got stuck during the elicitation process. This also aligns with the results shown for **RQ1**, where three participants experienced problems during the creation of a scenario. This condition is also given by Figure 6.5d. It shows that every second participant got stuck during the elicitation process. Three participants of those who got stuck were able to return to the conversation without the help of the *Chatbot Reset* functionality.

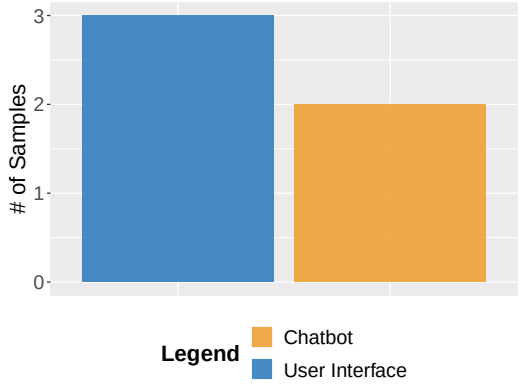
Figure 6.6 offers more insight on the used features of the prototype. *Quick Replies* were the most used feature of the prototype, followed by the *Architecture Graph* as presented by Figure 6.6a. Lesser used features were the *Management View* and *Chatbot Reset*. The frequent use of *Quick Replies* and *Architecture Graph* is also a result of participants liking these features as depicted in Figure 6.6b. Feedback about the disliked features is given in Figure 6.6c. From the qualitative feedback and the recordings, we noticed that three participants used a display resolution smaller than 1600x900 and therefore had an unpleasant interaction with the interface.



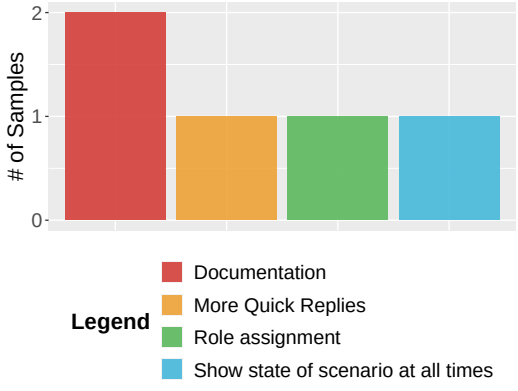
(a) Which features of the prototype did you use?



(b) Which feature(s) of the prototype did you like?



(c) Which feature(s) of the prototype did you dislike?



(d) What features are missing in the prototype?

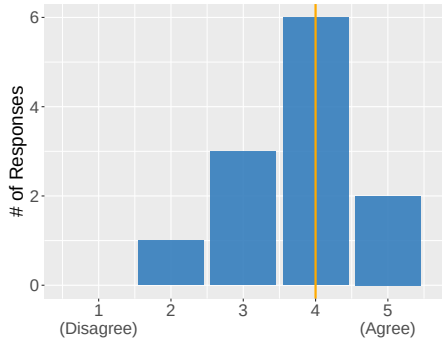
Figure 6.6: Results from questions about the chatbot's features.

Besides disliked features, participants also made suggestions for improvements to the prototype. In Figure 6.6d the participants suggestions are shown. Two people were asking for general documentation of the prototype and wanted to see all the available functionalities of the prototype. Other suggestions for improvements were more *Quick Replies*, role assignment (i.e., software architect, testing engineer), and the display of the scenario at all times. In the feedback interviews, different opinions about the prototype's appearance and documentation arose. Less experienced users were fine with the provided documentation. Furthermore, they preferred to see only the architecture graph and the chatbot. Experts, on the other hand, required more details. On the one hand, they wanted to see the scenario and its parameters during the configuration process and not just at the end. On the other hand, the experts were unaware of how the proposed suggestions from the chatbot were created.

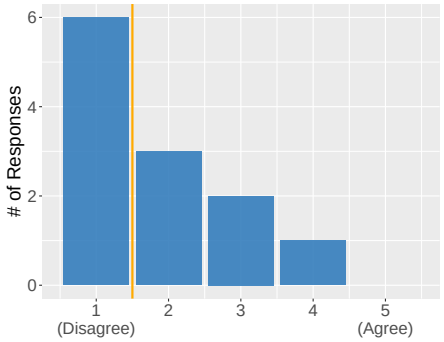
As described in the introduction of this chapter, we used the SUS to measure the usability of the system. These questions make up the third part of the questionnaire. Metrics collected from the questions of the SUS are depicted in Figure 6.7 and Figure 6.8. In the following summary of the results from the SUS, we group questions that can be categorized to a similar type. The four types of groups are (i) Ease of Use, (ii) Required Knowledge, (iii) Feature Integration, and (iv) Experience.

Ease of Use To this category we count the second, third, and eighth question of the SUS, that are visualized in Figure 6.7b, Figure 6.7c, and Figure 6.8b respectively. In both the second question (*I found the system unnecessarily complex.*) and the eighth question (*I found this system very cumbersome to use.*), the median is either below or equal to the value of two. Given answers tend to be on the lower end of the x-axis. A value smaller than three represents disagreement with the given questions. This suggests that most participants could use the prototype without problems. The third question (*I thought the system was easy to use.*) can not be interpreted so easily. Although the median of four suggests that the system was easy to use, the feedback from the users is more diverse. With two out of three questions answered in favor of the prototype's usability, we can confirm ease of use.

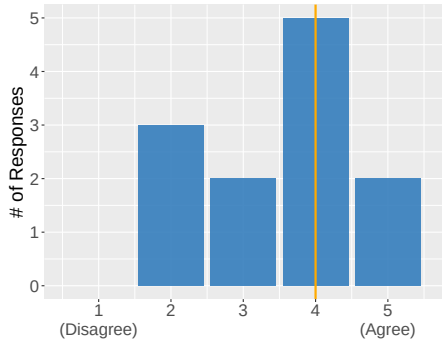
Required Knowledge To the second category we count the fourth and seventh question of the SUS which are visualized in Figure 6.7d and Figure 6.8a respectively. The fourth question (*I think that I would need the support of a technical person to be able to use this system.*) and the seventh question (*I would imagine that most people would learn to use this system very quickly.*) can be interpreted again in favor of the prototype's usability. Question four was answered with a median of two, with all answers except one on the lower end of the x-axis. A lower value suggests that there is no technical knowledge needed to use the system. Question seven also shows a median of two. Here, all answers without exception tend to agree that most people would learn the system quickly. Both questions in this category are answered with a positive tendency in favor of the prototype.



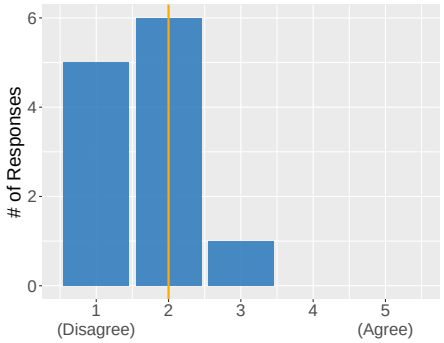
(a) Question 1: I think that I would like to use this system frequently.



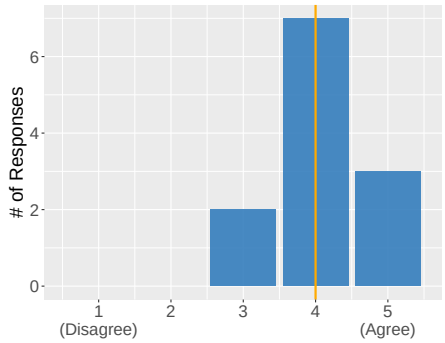
(b) Question 2: I found the system unnecessarily complex.



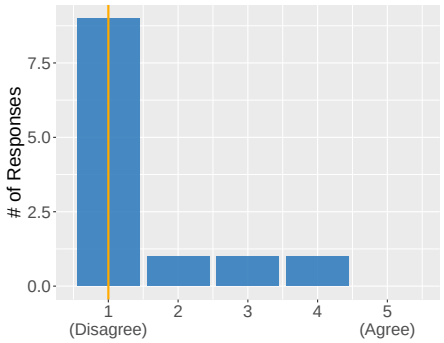
(c) Question 3: I thought the system was easy to use.



(d) Question 4: I think that I would need the support of a technical person to be able to use this system.



(e) Question 5: I found the various functions in this system were well integrated.

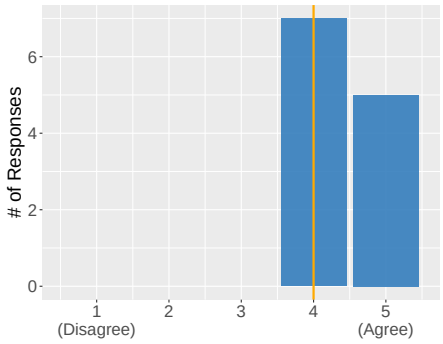


(f) Question 6: I thought there was too much inconsistency in this system.

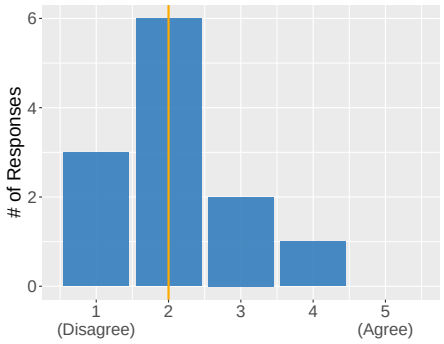
Figure 6.7: Results of the SUS for question 1–6.

Feature Integration Question five and six of the SUS are concerned with the integration of features. Results of both questions indicate that the features of the prototype were integrated well. The median for the fifth questions (*I found the various functions in this system were well integrated.*) is four, suggesting agreement that functions of the system were well integrated. Only two responses used the neutral value of three. This is also visualized in Figure 6.7e. In Figure 6.7f the results of the sixth question (*I thought there was too much inconsistency in this system.*) are shown. A median of one, with nine votes, indicates that there was no inconsistency in the prototype. We can confirm that all features of the prototype are well integrated.

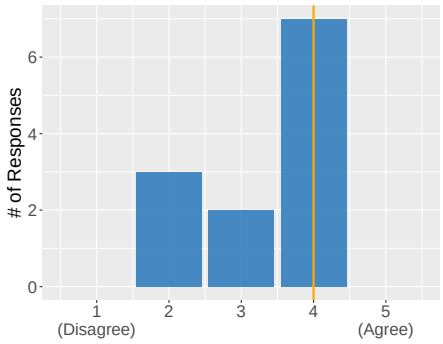
Experience Questions about the participants' experience are visualized in Figure 6.7a, Figure 6.8c, and Figure 6.8d. While the median of all three questions tends to give the impression that participants had a positive experience there is too much distribution amongst the answers to give a clear answer. Question one (*I think that I would like to use this system frequently.*), nine (*I felt very confident using this system.*), and ten (*I needed to learn a lot of things before I could get going with this system.*) have to be interpreted with as neutral results for the prototype.



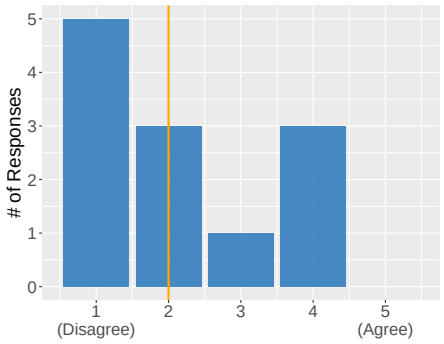
(a) Question 7: I would imagine that most people would learn to use this system very quickly.



(b) Question 8: I found this system very cumbersome to use.



(c) Question 9: I felt very confident using this system.



(d) Question 10: I needed to learn a lot of things before I could get going with this system.

Figure 6.8: Results of the SUS for question 7–10.

Based on the results of the number of used features, acceptance of supportive features, and metrics from the SUS we can make a statement about **RQ3**. Most study participants were happy about the integration of the supportive features. While precisely half of the participants got stuck during the elicitation process, they could resolve their issues with or without the help of the chatbot. *helpful* was the most rated adjective of the prototype. In combination with a rating of only one negative adjective, this seems to support the previous statement. Results from the SUS show a strong acceptance of the prototype, especially in the domain of *Ease of Use*, *Required Knowledge*, and *Feature Integration*. Overall, we can say that the prototype is able to support users with high certainty. This leads to the evaluation of qualitative statements to make the distinction between novice and expert users. In the following, we list statements from inexperienced, followed by a list of statements from experts. The listed statements are results from questions about the interaction with the prototype and general impressions during the interviews.

Feedback from inexperienced resilience engineers:

- "[…] I selected something in the graph before telling the chatbot to create a new scenario. The bot wanted to change the first scenario, and I found no way to cancel that."
- "The user interface was very well-designed, making it pleasant to use. It helped me to understand what is happening."
- "[The prototype is] nice to look at, the chatbot directs the flow of the conversation quite intuitively. I never felt like I was unsure how to proceed."
- "I think [the prototype] would be useful to learn how to describe resilience scenarios and quickly create a set of scenarios for a new system."
- "[The prototype] is nice to use and should provide architects and resilience engineers with the capability of quickly eliciting a set of base scenarios for a given microservice system."
- "The explanations helped to enter the correct information and somehow understand what the bot wanted without explaining too much."
- "[Quick Replies helped] me to quickly create a resilience scenario."

Feedback from experts:

- "[…] The conversation felt like a wizard, but I had to guess which turn the chatbot choose."
- "I had insufficient trust regarding the completeness of text suggestions."
- "I was confused that I could choose only a service or an operation for the artifact."
- "There are too few possible actions to warrant a chatbot. It is not apparent to the user what his potential actions are."
- "[More documentation] would have provided more clarity on what to do."
- "I think the prototype simplifies the elicitation process. However, in my opinion, the prototype can not replace traditional group meetings. I think [the prototype] would be a good addition to the traditional approach for preparing scenarios upfront."
- "[The prototype] provides a faster elicitation process. I am skeptical that it can replace group meetings. In any case, [the prototype] can be used to prepare a list of base scenarios that engineers can discuss in meetings."

- "The prototype provides a systematic process and stores the results in a structured manner. Requirements elicited in traditional approaches are documented rarely. Sometimes details are neglected or left out completely."
- "The chatbot is easy to use. It leads the user through an interactive process of creating quantifiable resilience scenarios."
- "The architecture graph helped to get a holistic view of the whole system."

Feedback from inexperienced users is focused more on the usability, appearance, and successful creation of a scenario. Overall, there are more positive comments than negative ones. Experts responded with more critical feedback. On the one hand, they seemed to be happy with the usability of the prototype. On the other hand, they felt that more features were missing to highlight details of a scenario and give more insights into the technical background of the analysis process. Furthermore, expert users reported that the available options for stimuli and responses were too limited to create usable scenarios in practice.

6.5.6 RQ4: How usable are ATAM-based scenarios created with Resirio for resilience specifications?

For the evaluation of **RQ4**, we use feedback from the participants given through quantitative and qualitative questions. Results rely on the expertise of both experts and inexperienced users with the respective systems.

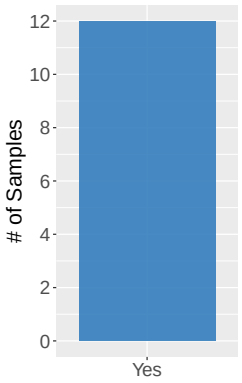
Figure 6.9a shows the acceptance of the provided *Quick Replies*. All participants feel like the provided *Quick Replies* are suitable to configure the parameters of a resilience scenario. In conversation with participants, especially the inexperienced engineers, the quick feedback loop is an immediate result of the preferred use of *Quick Replies* over the use of text responses.

In Figure 6.9b the results of the question *Does the chatbot provide enough Quick Replies?*. While experts also rated the *Quick Replies* as suitable to create resilience scenarios, they felt that the *Quick Replies* provided by the prototype could not cover all cases. Participants had the chance to configure the contents of the scenario parameters, yet they favored the use of the *Quick Replies* for reasons of performance and convenience.

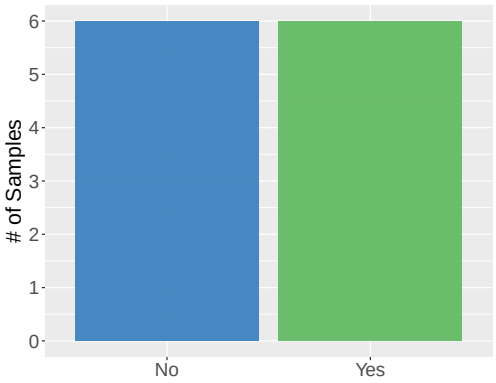
Regarding the usefulness of scenarios resulting from the interaction of the prototype, there seems to be no consent. There is no clear opinion if the interactive approach can produce more useful scenarios than the traditional approach. As depicted in Figure 6.9c, seven participants thought the interactive approach produces more useful scenarios. Five participants thought that the traditional approach produces more useful scenarios. In the interview-style feedback round, we found out that especially experts believed that the traditional approach could not be replaced with the proposed solution.

As mentioned before, participants had the impression that more *Quick Replies* were necessary to produce a better quality scenarios. The suggestions made by the chatbot did not cover all the cases that the participants would have liked to specify. However, Figure 6.9d shows that 75% of participants think that the scenarios created with the prototype can be used for the specification of

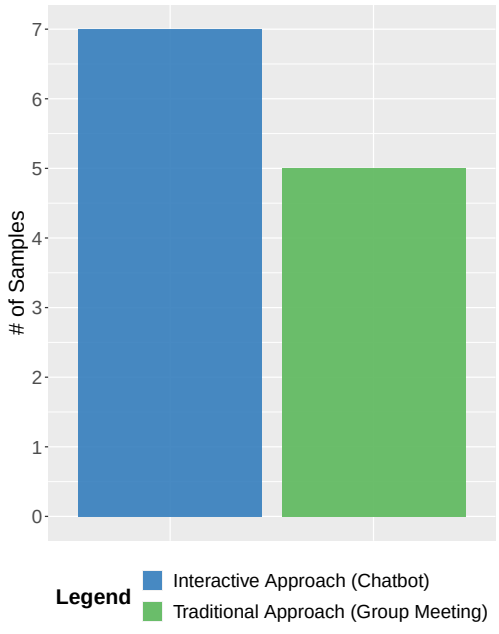
system components to test its resilience. Answers given during the feedback round pointed out that suggestions from the chatbot should support more types of stimuli and provide more specifications of metrics besides availability for services and response times of operations.



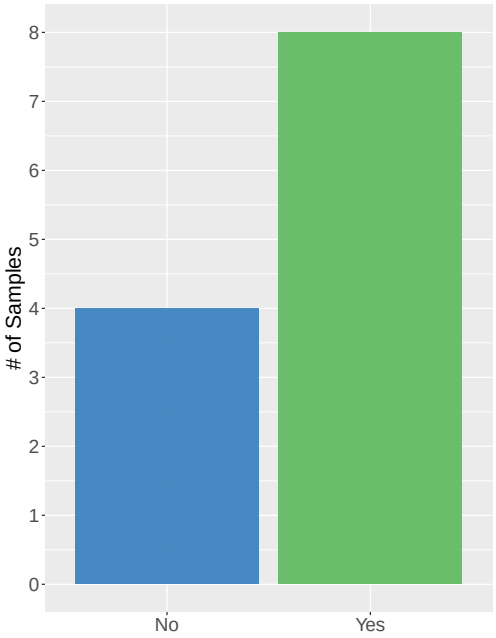
(a) Are the chatbot's quick replies suitable enough to elicit a useful resilience scenario?



(b) Does the chatbot provide enough *Quick Replies* (blue buttons)?



(c) Which approach to elicit resilience scenarios produces more useful scenarios for resilience specifications?



(d) Do you think that resilience scenarios created with the prototype provide enough information to test a systems' resilience?

Figure 6.9: Results from questions about the quality of elicited scenarios.

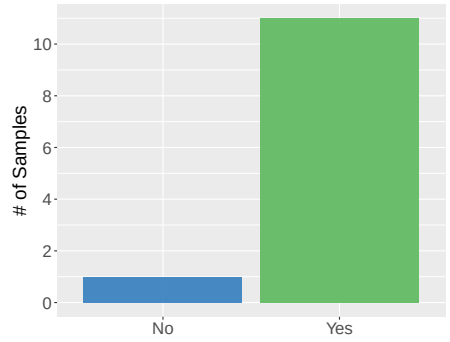


Figure 6.10: Would you use the prototype in the future or recommend it?

To conclude this section, we want to highlight the participants' experience with the prototype and give an impression of the potential future use. In Figure 6.10 the participants' feedback about recommending the prototype is visualized. All except one person had a pleasant experience with the prototype and would either recommend the prototype or use it in the future. While participants made many comments for improvements and suggestions for new features, the prototype was welcomed by the majority and seemed easy to use.

6.6 Discussion

In this section, we argue about the acceptance or rejection of the hypothesis phrased in Section 6.1 for every research question. In Section 6.6.1 the results overall applicability of the prototype is given, while Section 6.6.2 is concerned with the effectiveness. In Section 6.6.3 results about the supportive features are summarized. Section 6.6.4 concludes the discussion with the usability of the created resilience scenarios. The basis for the discussion are results formulated in Section 6.5.

6.6.1 RQ1: Are users of Resirio able to create resilience scenarios successfully?

All participants of the user study were able to complete the given tasks. The solution for each task was to create one resilience scenario. Six participants had to reconfigure the parameters of a scenario because they were not satisfied with the chosen parameters the first time, or the chatbot did not completely assign the specified parameters. With the help of supportive features and explanations given by the prototype, they could still complete their scenarios.

100% of participants were able to create scenarios successfully. Only three of all 24 final scenarios contained invalid parameters. This limits the portion of correct and successfully created scenarios to 87.5%. Four participants had difficulties understanding what they had to do in the first task. However, while using the chatbot and creating their first scenario, it became clear to them later during the elicitation process. While they created more scenarios than necessary, they could still create a correct scenario with valid parameters, except for one person. It is safe to say that H_{11} can be accepted and H_{10} is rejected.

6.6.2 RQ2: How effective is Resirio in contrast to the traditional elicitation process?

Participants were able to complete both tasks within a range of seven and 25 minutes. The median of the used time is 16 minutes and 14 seconds. As the prototype could not collect isolated results for either task, we rely on the results of the recordings. We see an apparent learning effect that participants experienced while solving the first task. In general, the first task took participants more than half of their used times. We collected the measurements from the recordings which indicates an approximated median of twelve minutes for the completion of the first task. The second task could be solved faster, while it was more difficult. Recordings indicate an approximated median of five minutes to complete the second task. This shows that the prototype is quick to learn and an apparent quick learning curve for inexperienced resilience engineers and software architects.

Discussion with the participants who had previously taken part in the traditional elicitation process confirmed our findings. In their opinion, the presented approach can be used to produce resilience scenarios faster than in the traditional approach. In addition, our solution seems to provide another benefit. In the traditional approach, the lack of consistent documentation seems to be a big problem. Results of the elicitation process with our are manifested in the reusable digital scenario format. It allows resilience engineers and software architects to concentrate on the details of the scenario without worrying about the documentation and formatting of the scenario, as would be the case in the traditional approach.

As we mentioned before, there is only limited research and measurements on the effectiveness of group meetings for the requirements process involving resilience scenarios. Other works [BCK03; HT11; KA00; KKB+98] discuss that group meetings can take a long time, yet details are not enclosed. While we think that the interactive approach presented in this work can produce resilience scenarios much faster than the traditional approach, we hesitate to accept either research hypothesis. Therefore we can not accept H_{21} or reject H_{20}.

6.6.3 RQ3: How supportive is Resirio during the elicitation process?

Results from Section 6.5 indicate that participants of the study were very pleased with the support that Resirio provided during the elicitation process. 50% of all participants got stuck during the elicitation process. On the one hand, the weak text recognition of the chatbot was responsible. On the other hand, users were initially not completely sure how to interact with the chatbot. Only 50% of the participants used the *Chatbot Guide* to let the chatbot explain an aspect of the research area or help with the configuration of a scenario parameter. While the usage of text responses was minimal, the usage of *Quick Replies* was very welcomed by users. Furthermore, results from the SUS categories (i) Ease of Use, (ii) Required Knowledge, (iii) Feature Integration, and (iv) Experience show a clear tendency that prototype is easy to use, requires only limited knowledge, and provides well-integrated features. However, not all participants were pleased with the experience they got from using the prototype. In the hypothesis defined for **RQ3**, we made a distinction between novice and expert users.

Novice users had a clear focus on the appearance and usability of the prototype. Statements collected during the feedback round at the end of the study resulted in positive feedback. Especially the architecture graph in combination with the *Quick Replies* were very welcomed and used often. The 85% of inexperienced users were satisfied with the provided supportive features. Therefore, we accept H_{311} and reject H_{310}.

Experienced users with a professional background in resilience engineering gave more critical statements about the prototype. Suggestions provided by the prototype were not extensive enough and focused only on limited metrics. Although the analysis depends on the data provided by traces, we see room for further collection of other metrics from different analysis strategies. Further comments complained about the fixed amount of available stimuli. While experts had the chance to configure those parameters through text responses, they argued that they would not want to use text responses. This was a result of weak text recognition by the chatbot. Besides critical comments, experts also gave positive comments about several features of the prototype. For example, the architecture was inspected for a longer time by experts. Other comments suggest that the systematic approach of storing requirements in scenarios could help to improve the traditional approach. Only 60% of expert users were satisfied with the provided supportive features. We can confirm H_{321} and reject H_{320} only by a small margin. The supportive features of the prototype can help expert users but leave room for improvement.

6.6.4 RQ4: How usable are ATAM-based scenarios created with Resirio for resilience specifications?

Feedback from the participants includes agreement and critical comments about the use of ATAM-based scenarios for specifications. 100% of participants think that the *Quick Replies* provided by the chatbot are suitable enough to configure resilience scenarios with useful parameters. Comments from participants included, that the hazard analysis provided too few suggestions for stimuli. Others commented that the proposed options for the response measures only cover availability and response time metrics. For some MSAs this is not enough to create a functional specification. Comments suggested that the hazard analysis should be extended with monitoring data besides information extracted from traces. 75% of all participants think that the resilience scenarios provide enough information to test a systems' resilience.

Further feedback collected during the interview-style questions revealed that experts especially doubt that the presented approach can produce resilience scenarios qualitatively equal to the traditional approach. In their opinion, the options provided in this approach can not achieve the same coverage of resilience attributes as in the traditional approach. Criticism includes that only one person is involved in a conversation with the chatbot in the presented approach. In the traditional approach, a group of resilience engineers discuss the appropriateness of resilience parameters. Furthermore, architects mentioned that the interactive approach should be used in combination with the traditional approach. Experts say that they see two benefits of combining the traditional approach with the presented approach. On the one hand, the usage of Resirio in a group meeting should accelerate the elicitation of resilience scenarios. On the other hand, Resirio provides a systematic and structured process to document requirements discussed in group meetings that is sometimes missing.

We accept hypothesis H_{41} and reject H_{40}, however, only in a limited context. Participants think that the created resilience scenarios can not be used effectively to create resilience specification in the real world because of more outside influences that are not covered by the chatbot suggestions and the ATAM-based scenarios.

6.7 Threats to Validity

We used a user study to evaluate the developed prototype. The study was conducted in a constrained context with a limited number of participants, of which several came from the same institutions. These circumstances and other decisions made in the studies' design and during the study execution limit the evaluation results. In the following sections, the four main threats to validity for the thesis evaluation are discussed. We use definitions of Wohlin et al. [WRH+12] to argue about the results. In Section 6.7.1 the conclusion validity is examined followed by Section 6.7.2 that is concerned with the internal validity. Section 6.7.3 discusses the construct validity while Section 6.7.4 is emphasizing on the external validity.

6.7.1 Conclusion Validity

Conclusion validity covers the aspect of drawing the correct conclusion from an experiment based on the statistical measurements [WRH+12]. Although this thesis is mainly concerned with qualitative evaluation, the conclusion validity can still be applied.

As most questions asked during the study were open-ended and of qualitative nature, it is hard to prove if we made the correct conclusion. Since we used only a limited amount of statistical measurements of the participants answers, we rely on our own classification of answers. Furthermore, the evaluation of this thesis was done by a single person who was also responsible for developing the prototype and the execution of the study. This implies that qualitative feedback might be interpreted based on subjective bias.

Besides the qualitative questions, the evaluation is also based on metrics collected during the study. As mentioned before, the number of participants was limited, resulting in ambiguous conclusions from measurements and general uncertainty in the evaluation. It is also not clear if participants of the study gave their complete feedback. Through time constraints, it is possible that some feedback could not be asked by the conductor or only arose after the study.

6.7.2 Internal Validity

The internal validity describes the extent to which the causality of independent variables is supported by the claims an experiment makes [WRH+12].

For the evaluation, potential participants were invited personally by e-mail and given a short introduction to the research interest and the goals of this thesis. Giving the additional information was necessary in some cases to convince them to participate. Since we could not rely on the

participation of volunteer people from the public, this step was needed to ensure that participants had a profound knowledge of the system they inspected. Providing the participants with additional information might have influenced their view and experience of the prototype.

The target group mainly consisted of participants with different levels of experience. This background knowledge of the participants is hard to compare. An uneven share of knowledge makes it difficult to compare the results of the created scenarios, especially with the limited number of participants.

Another aspect that might have influenced the results of this experiment was the existing knowledge about the prototype by some participants. Four participants that took part in the study had previously seen the prototype in a demonstration. Although they did not use the prototype, some transfer of knowledge might have happened. Especially the use of *Quick Replies* in the demonstrations provided a tendency for these participants to prefer *Quick Replies* in contrast to written texts.

As the experiment was executed remotely over the meeting software Webex, the experiment setup could not be enforced. Study participants had to use their personal equipment as no fixed hardware could be provided to them. Furthermore, study participants could use their preferred browser and access PCs with different specifications and performance. Some participants were using a second monitor, while others did not. Using a second monitor to read the task description does influence the speed of the task completion.

Since the number of participants was limited before the invitation was sent out, it was decided to keep the duration of the experiment to a minimum. Other studies have shown that the longer user studies take, the less reliable the results get [SEM12]. Furthermore, more extended studies have a psychological effect on participants that may lead to tiredness [WRH+12]. The decision to have a limited amount of time available for the study led to the judgment to use only two tasks with increasing difficulty. The effect of less exhaustion among the participants in contrast to the traditional elicitation approaches might also affect the resulting rating of the prototype. Using two scenarios can also be an influence for an undesired learning effect during the study. As measured in the results, the second task could be completed faster than the first task.

6.7.3 Construct Validity

The construct validity is concerned with the measurement of experiment metrics [WRH+12]. It is used to see if measurements do support the claim of the hypothesis.

In regards to **RQ1**, the correct measurement was used to answer this research question. The completeness of a scenario can be determined by the presence or absence of the scenario parameters. Although it is possible to create scenarios without meaningful parameters, the completeness is not influenced.

RQ2 uses the metric of duration to create a scenario as the measurement for the effectiveness. Therefore, the comparison to traditional elicitation processes is limited for the lack of existing measurements. We do not think that values extracted from experience reports and other literature reviews provide a good enough basis for comparison.

RQ3 is concerned with the support that the prototype provides. Since this quality is highly influenced by the experience and the subjectivity of the study, participant results must be considered with caution. Furthermore, besides the qualitative questions, there is no measurement used to validate

the results. Testing the usability of a system always offers some variation and depends on the participants' perception. As the SUS was used to measure the prototype's usability, we rely on the effectiveness of the SUS in academic research.

The research question with the highest amount of uncertainty is **RQ4** where the usability of scenarios was tested. We used qualitative and quantitative feedback from the users to evaluate this research question. The reliance on the participants' experience with their respective microservice systems might not yield accurate results. While we made sure that the participants had a good enough understanding of the microservices used in the study, it is not directly clear if the created scenarios can be used for a specification in the real world.

6.7.4 External Validity

External validity covers the aspect of generalization the results of an experiment and transferring it to the real world [WRH+12]. It should be noted that after Calder et al. [CPT82], not every research project has to have an emphasis on generalization. The results that were shown in this work should not be understood to be generally applicable to the domain of requirements elicitation without any restrictions. The study was conducted in a qualitative manner, which makes generalization even harder based on the subjectivity of the participants.

Since the study was conducted with volunteers who received an explicit invitation, it is very hard to assume that the results of this work can be applied in the real world. By the nature of this work's domain-specific topic, the number of potential study candidates is already limited. Furthermore, as we emphasized in the introduction, scenario elicitation is done only in a limited amount of environments. As a result, the participants were selected by contacts to the research and industry. Besides, participants of the study came from the same institutions and companies. This further complicates the generalization of results in surroundings that are different from those found in the participants' environments.

Finally, it should be noted that the author of this work created the task descriptions. Tasks were designed to be as close as possible to real-world scenarios. However, the complexity of the topic makes it very hard to compare them with each other. Results that were shown might not be transferable to real-world applications.

7 Conclusion

The following sections summarize and conclude this thesis. Section 7.1 summarizes the key aspects of this work. In Section 7.2 the benefits are highlighted while Section 7.3 describes the limitations. Lessons learned during the thesis are given in Section 7.4. Section 7.5 gives an outlook on potential extension on this work.

7.1 Summary

This work examined systematic hazard analysis techniques in the domain of software systems. We looked for ways to adapt one of these approaches for MSA. Results of the analysis should yield a starting point for requirements elicitation of resilience attributes. Other works about requirements elicitation have been proposed using interactive methodologies. Yet, they lack a systematic requirements approach or are missing a structured process to store results.

Based on findings in research about hazard analysis techniques and shortcomings of other works in the domain of interactive requirements elicitation, a prototype was designed and implemented. The prototype contains an architecture extraction that uses traces from Zipkin and Jaeger to describe the architecture of a microservice-based software system. A CHAZOP-based approach was designed to identify stimuli within the architecture based on available metrics in the trace. The prototype was extended with a chatbot as an interactive component to communicate with stakeholders of a microservice-based software system. This chatbot makes suggestions of resilience-related attributes to software architects and resilience engineers to support them in creating ATAM-based scenarios. Resulting scenarios can be used to create a specification for components of a MSA.

Through a user study, the developed approach was evaluated. Novice resilience engineers and expert software architects were invited to test the prototype with a MSA they had been involved with during development. Feedback from both user groups showed that there is great potential for interactive requirements elicitation. The prototype can support analysts in the creation of resilience scenarios and provides means to store them for future use. While the prototype satisfied the use cases of inexperienced resilience engineers, expert analysts think it can not replace the traditional approach of group meetings. However, experts think that a refined prototype could bring significant benefits in combination with the traditional approach. The traditional approach could be accelerated, and documentation of requirements could be done in a structured manner.

7.2 Benefits

The presented approach is able to support software developers and engineers that are concerned with the elicitation of resilience-related attributes. Our approach provides a structured approach for the creation of resilience scenarios. The addition of an interactive component lets even inexperienced resilience engineers create scenarios quickly. Based on the difficulty of a given task, resilience engineers can create ATAM-based scenarios within five and twelve minutes.

Traces from Zipkin and Jaeger can be uploaded and automatically analyzed by the prototype. The architecture extraction and inspection of traces provides a quick way to visualize the connectivity of services and operations. Created scenarios are directly visible to users of the system and provide further insights to trace details like logs and response times.

While we could not show that the presented solution is able to replace the traditional approach of group meetings, we see several benefits of using it in combination. On the one hand, the prototype provides a fixed documentation format that is sometimes missing in the traditional approach. On the other hand, analysts could produce a set of base scenarios with the prototype. Resilience engineers could then focus on the discussion of suitable resilience attributes for these scenarios.

7.3 Limitations

The developed solution is only applicable to microservice-based systems. Furthermore, the CHAZOP-based analysis can only be used with information from traces. The architecture extraction from traces does only work for traces of Zipkin and Jaeger. Therefore, the response time is the only metric used by the analysis.

The results of the conducted study have to be considered with care. We collected feedback from twelve participants, of which five had already seen the prototype. This limits the results and findings of our evaluation. To increase the reliability of our conclusions, more participants, both novices and experts, would need to take part in a similar study setting.

Feedback from study participants also showed that tasks used in the study are not feasible to replicate complex situations in the real world. On the one hand, metrics from traces are not enough to create a functional specification. On the other hand, there is a multitude of outside factors not considered in the study.

7.4 Lessons Learned

A large amount of time for this work was spent implementing the chatbot backend. This was a result of a lack of insight into the DF implementation and the dependence of the DF UI. The use of the DF client was one way to reduce the dependency to the DF UI. However, the definition of intents and configuration of the knowledge base still have to be done in the DF UI. Creating intents in the backend could have been realized but would have required more initial effort. We think that managing everything over the DF client could have saved time in the later stages of development and improved maintainability at the same time. Furthermore, some functionality provided by the

chatbot (i.e. choosing a default, skip a step, jump between steps, or showing the summary), was not used at all during the study. One reason, was the lack of trust given to the chatbot's text recognition and the preferred use of *Quick Replies*. Another reason was lacking documentation of the chatbot's features in the *Client Interface*.

During the execution of the user study, it became clear that some participants were not experienced enough to define resilience parameters of scenarios on their own. The second task of the study was intentionally phrased more openly and required participants to use their understanding of the inspected microservice-based system. All participants were able to complete the tasks. Yet, we think a more detailed task description with the incorporation of all needed resilience parameters would have benefited the result of the user study.

7.5 Future Work

As stated in Section 7.3, the presented approach is limited to the use of traces for the hazard analysis. Furthermore, the hazard analysis can only make suggestions through the chatbot for two metrics availability and response time. In a real-world application, this is not enough to inspect the complete resilience of a service. Further metrics from monitoring systems could be used to extend the capabilities of the hazard analysis and make more sophisticated suggestions to the user.

In its current design, the chatbot does not take previously created scenarios into account. In traditional requirements approaches, it is common practice to refine scenarios and create variations of those for the same artifacts. Suppose the chatbot backend is given the parameters of the elicited scenarios. In that case, it could make better recommendations to the users and produce multiple scenarios for the same artifact in one iteration of the conversation.

While the study showed that the prototype has a short learning curve, it was not intuitively clear to some users how the prototype works. Further documentation for available resilience parameters and the display of the current state of resilience scenarios at any time could bring more clarity. Visualization of runtime data for the inspected services could also improve the detection of weak components in the system.

One significant finding in this work was the hesitant use of text interactions with the chatbot. Participants of the study preferred the use of *Quick Replies* in favor of effectiveness and preciseness. Reasons for not using text interactions were previous skepticism about chatbots and weak text recognition. Voice commands could accelerate the elicitation process even further, but they might also be affected by imprecise intent detection. Extensions of this work should focus more on ways to provide direct user input rather than text interactions.

Bibliography

[AGL+15] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, A. Tang. "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar". In: *IEEE Transactions on Software Engineering* 41.7 (2015), pp. 620–638 (cit. on pp. 31, 45).

[AH91] R. Alur, T. A. Henzinger. "Logics and models of real time: A survey". In: *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. Springer. 1991, pp. 74–106 (cit. on p. 30).

[AMSW19] D. Arruda, M. Marinho, E. Souza, F. Wanderley. "A Chatbot for goal-oriented requirements modeling". In: *International Conference on Computational Science and Its Applications*. Springer. 2019, pp. 506–519 (cit. on pp. 2, 33, 37).

[AMW+03] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, A. Muthitacharoen. "Performance debugging for distributed systems of black boxes". In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 74–89 (cit. on p. 26).

[And91] P. Andow. *Guidance on HAZOP procedures for computer-controlled plants*. Great Britain, Health and Safety Executive, 1991 (cit. on p. 13).

[And98] J. Andrews. "Tutorial Fault Tree Analysis". In: *Proceeding of the 16th International System Safety Conference, Loughborough*. 1998 (cit. on p. 16).

[BAG17] P. Bailis, P. Alvaro, S. Gulwani. "Research for practice: Tracing and debugging distributed systems; programming by examples". In: *Communications of the ACM* 60.7 (2017), pp. 46–49 (cit. on p. 45).

[Bay15] P. Baybutt. "A critique of the Hazard and Operability (HAZOP) study". In: *Journal of Loss Prevention in the Process Industries* 33 (2015), pp. 52–58 (cit. on pp. 10, 13).

[BCD+06] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, J. Chau. "Framework for instruction-level tracing and analysis of program executions". In: *Proceedings of the 2nd international conference on Virtual execution environments*. 2006, pp. 154–163 (cit. on p. 26).

[BCK03] L. Bass, P. Clements, R. Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003 (cit. on pp. 2, 4, 10, 43, 44, 74, 95).

[BDH03] L. A. Barroso, J. Dean, U. Holzle. "Web search for a planet: The Google cluster architecture". In: *IEEE micro* 23.2 (2003), pp. 22–28 (cit. on p. 27).

[Bec20] S. Beck. "Evaluating Human-Computer Interfaces for Specification and Comprehension of Transient Behavior in Microservice-based Software Systems". In: 2020 (cit. on pp. 4, 58, 77).

[Ben09] M. Ben-Daya. "Failure mode and effect analysis". In: *Handbook of maintenance management and engineering*. Springer, 2009, pp. 75–90 (cit. on pp. 19, 20).

[BKM08] A. Bangor, P. T. Kortum, J. T. Miller. “An empirical evaluation of the system usability scale”. In: *Intl. Journal of Human–Computer Interaction* 24.6 (2008), pp. 574–594 (cit. on p. 73).

[BL94] T. Ball, J. R. Larus. “Optimally profiling and tracing programs”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.4 (1994), pp. 1319–1360 (cit. on p. 26).

[BM+01] L. Bass, G. Moreno, et al. *Applicability of general scenarios to the architecture tradeoff analysis method*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2001 (cit. on p. 9).

[BP93] D. Burns, R. Pitblado. “A modified HAZOP methodology for safety critical system assessment”. In: *Directions in safety-critical systems*. Springer, 1993, pp. 232–245 (cit. on p. 48).

[Bro96] J. Brooke. “Sus: a “quick and dirty’usability”. In: *Usability evaluation in industry* 189 (1996) (cit. on p. 75).

[BT12] C. Bennett, A. Tseitlin. “Chaos monkey released into the wild”. In: *Netflix Tech Blog* 30 (2012) (cit. on p. 1).

[CDG+08] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber. “Bigtable: A distributed storage system for structured data”. In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26 (cit. on p. 28).

[CE81] E. M. Clarke, E. A. Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Workshop on Logic of Programs*. Springer. 1981, pp. 52–71 (cit. on p. 30).

[CES86] E. M. Clarke, E. A. Emerson, A. P. Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), pp. 244–263 (cit. on p. 30).

[Cha05] R. N. Charette. “Why software fails [software failure]”. In: *IEEE spectrum* 42.9 (2005), pp. 42–49 (cit. on p. 1).

[CKRB19] J. Cerezo, J. Kubelka, R. Robbes, A. Bergel. “Building an expert recommender chatbot”. In: *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. IEEE. 2019, pp. 59–63 (cit. on p. 38).

[CM01] P. Conmy, J. A. McDermid. “High Level Failure Analysis for Integrated Modular Avionics.” In: *SCS*. 2001, pp. 13–22 (cit. on p. 22).

[CNYM12] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos. *Non-functional requirements in software engineering*. Vol. 5. Springer Science & Business Media, 2012 (cit. on p. 1).

[Com+90] I. E. Commission et al. *IEC 61025 Fault Tree Analysis*. 1990 (cit. on pp. 8, 15, 16, 19).

[CPT82] B. J. Calder, L. W. Phillips, A. M. Tybout. “The concept of external validity”. In: *Journal of consumer research* 9.3 (1982), pp. 240–244 (cit. on p. 99).

[DAC98] M. B. Dwyer, G. S. Avrunin, J. C. Corbett. “Property specification patterns for finite-state verification”. In: *Proceedings of the second workshop on Formal methods in software practice*. 1998, pp. 7–15 (cit. on p. 45).

[DAC99] M. B. Dwyer, G. S. Avrunin, J. C. Corbett. “Patterns in property specifications for finite-state verification”. In: *Proceedings of the 21st international conference on Software engineering*. 1999, pp. 411–420 (cit. on pp. 30, 45).

[DDML97] R. Darimont, E. Delor, P. Massonet, A. van Lamsweerde. “GRAIL/KAOS: an environment for goal-driven requirements engineering”. In: *Proceedings of the 19th international conference on Software engineering*. 1997, pp. 612–613 (cit. on p. 37).

[DGH+14] S. Dogramadzi, M. E. Giannaccini, C. Harper, M. Sobhani, R. Woodman, J. Choung. “Environmental hazard analysis-a variant of preliminary hazard analysis for autonomous mobile robots”. In: *Journal of Intelligent & Robotic Systems* 76.1 (2014), pp. 73–117 (cit. on p. 22).

[DJ10] O. Dieste, N. Juristo. “Systematic review and aggregation of empirical studies on elicitation techniques”. In: *IEEE Transactions on Software Engineering* 37.2 (2010), pp. 283–304 (cit. on p. 34).

[DNA+21] I. Dauda, B. Nuhu, J. Abubakar, I. Abdullahi, D. Maliki. “Software Failures: A Review of Causes and Solutions”. In: *ATBU Journal of Science, Technology and Education* 9.1 (2021), pp. 415–423 (cit. on p. 1).

[EGA11] P. Espada, M. Goulão, J. Araújo. “Measuring complexity and completeness of KAOS goal models”. In: *Workshop on Empirical Requirements Engineering (EmpiRE 2011)*. IEEE. 2011, pp. 29–32 (cit. on p. 37).

[EGA13] P. Espada, M. Goulão, J. Araújo. “A framework to evaluate complexity and completeness of KAOS goal models”. In: *International Conference on Advanced Information Systems Engineering*. Springer. 2013, pp. 562–577 (cit. on pp. 36, 37).

[FBG18] E. Friesen, F. S. Bäumer, M. Geierhos. “CORDULA: Software Requirements Extraction Utilizing Chatbot as Communication Interface.” In: *REFSQ Workshops*. 2018 (cit. on p. 35).

[FGGB17] J. L. Fuentes-Bargues, M. González-Cruz, C. González-Gaya, M. Baixauli-Pérez. “Risk analysis of a fuel storage terminal using HAZOP and FTA”. In: *International Journal of Environmental Research and Public Health* 14.7 (2017), p. 705 (cit. on p. 12).

[FH94] P. Fenelon, B. Hebbron. “Applying HAZOP to software engineering models”. In: *Risk management and critical protective systems: proceedings of SARSS*. 1994, pp. 11–116 (cit. on pp. 11, 13, 48).

[FL14] C. H. Fleming, N. G. Leveson. “Improving hazard analysis and certification of integrated modular avionics”. In: *Journal of Aerospace Information Systems* 11.6 (2014), pp. 397–411 (cit. on p. 48).

[FPKS07] R. Fonseca, G. Porter, R. H. Katz, S. Shenker. “X-trace: A pervasive network tracing framework”. In: *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*. 2007 (cit. on p. 26).

[God00] P. L. Goddard. “Software FMEA techniques”. In: *Annual Reliability and Maintainability Symposium. 2000 Proceedings. International Symposium on Product Quality and Integrity (Cat. No. 00CH37055)*. IEEE. 2000, pp. 118–123 (cit. on p. 19).

[Gol16] Y. Goldberg. “A primer on neural network models for natural language processing”. In: *Journal of Artificial Intelligence Research* 57 (2016), pp. 345–420 (cit. on p. 31).

[Gol17] Y. Goldberg. “Neural network methods for natural language processing”. In: *Synthesis lectures on human language technologies* 10.1 (2017), pp. 1–309 (cit. on p. 31).

[Goo16] Google. https://cloud.google.com/dialogflow/es/docs/concepts. Accessed: 2020-12-9. 2016 (cit. on p. 32).

[HBB+11] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell. “A systematic literature review on fault prediction performance in software engineering”. In: *IEEE Transactions on Software Engineering* 38.6 (2011), pp. 1276–1304 (cit. on p. 1).

[Hor01] H. Horacek. *Building natural language generation systems*. 2001 (cit. on p. 31).

[HRJ+16] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar. “Gremlin: Systematic resilience testing of microservices”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 57–66 (cit. on p. 39).

[HT11] B. Hulin, R. Tschachtli. “Identifying software hazards with a modified CHAZOP”. In: *PESARO 2011 First Int. Conf. Performance, Saf. Robustness Complex Syst. Appl.* Vol. 12. 2011, p. 7 (cit. on pp. 1, 13, 48, 74, 95).

[HWL06] E. Hollnagel, D. D. Woods, N. Leveson. *Resilience engineering: Concepts and precepts*. Ashgate Publishing, Ltd., 2006 (cit. on pp. 1, 11).

[Jae15] Jaeger. https://www.jaegertracing.io/. Accessed: 2020-12-9. 2015 (cit. on p. 29).

[JH19] P. Johnston, R. Harris. “The Boeing 737 MAX saga: lessons for software organizations”. In: *Software Quality Professional* 21.3 (2019), pp. 4–12 (cit. on p. 1).

[KA00] F. I. Khan, S. Abbasi. “Towards automation of HAZOP with a new tool EXPERTOP”. In: *Environmental Modelling & Software* 15.1 (2000), pp. 67–77 (cit. on pp. 1, 13, 74, 95).

[Kab17] S. Kabir. “An overview of fault tree analysis and its application in model based dependability analysis”. In: *Expert Systems with Applications* 77 (2017), pp. 114–135 (cit. on p. 14).

[KBK+99] R. Kazman, M. Barbacci, M. Klein, S. J. Carrière, S. G. Woods. “Experience with performing architecture tradeoff analysis”. In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*. IEEE. 1999, pp. 54–63 (cit. on p. 9).

[KDK18] J. von Kistowski, M. Deffner, S. Kounev. “Run-time Prediction of Power Consumption for Component Deployments”. In: *Proceedings of the 15th IEEE International Conference on Autonomic Computing (ICAC 2018)*. Trento, Italy, 2018 (cit. on p. 39).

[KEM20] K. R. KALANTARI, A. Ebrahimnejad, H. Motameni. “Dynamic software rejuvenation in web services: a whale optimization algorithm-based approach”. In: *Turkish Journal of Electrical Engineering & Computer Sciences* 28.2 (2020), pp. 890–903 (cit. on p. 1).

[KHFH20] D. Kesim, A. van Hoorn, S. Frank, M. Häussler. “Identifying and prioritizing chaos experiments by using established risk analysis techniques”. In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2020, pp. 229–240 (cit. on p. 39).

[KKB+98] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere. “The architecture tradeoff analysis method”. In: *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No. 98EX193)*. IEEE. 1998, pp. 68–78 (cit. on pp. 2, 9, 74, 95).

[KKC00] R. Kazman, M. Klein, P. Clements. *ATAM: Method for architecture evaluation*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2000 (cit. on p. 9).

[Kle99] T. A. Kletz. *HAZOP and HAZAN: identifying and assessing process industry hazards*. IChemE, 1999 (cit. on pp. 10, 11).

[KMB+17] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O’Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, et al. “Canopy: An end-to-end performance tracing and analysis system”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 34–50 (cit. on p. 45).

[Koy90] R. Koymans. “Specifying real-time properties with metric temporal logic”. In: *Real-time systems* 2.4 (1990), pp. 255–299 (cit. on p. 30).

[KW20] D. Kesim, L. Wager. “Scenario-based Resilience Evaluation and Improvement of Microservice Architectures by Applying Automated Chaos Experimentation”. In: 2020 (cit. on pp. 4, 10).

[KWK+21] D. Kesim, L. Wager, J. von Kistowski, S. Frank, A. Hakamian, A. van Hoorn. “Scenario-based Resilience Evaluation and Improvement of Microservice Architectures: An Experience Report”. In: (2021). under review (cit. on pp. 8, 39, 44, 77).

[Lar90] J. R. Larus. “Abstract execution: A technique for efficiently tracing programs”. In: *Software: Practice and Experience* 20.12 (1990), pp. 1241–1258 (cit. on p. 26).

[Lee18] H. Lee. *Voice User Interface Projects: Build voice-enabled applications using Dialogflow for Google Home and Alexa Skills Kit for Amazon Echo*. Packt Publishing Ltd, 2018 (cit. on p. 32).

[Lev04] N. Leveson. “A new accident model for engineering safer systems”. In: *Safety science* 42.4 (2004), pp. 237–270 (cit. on p. 20).

[Lev11] N. G. Leveson. “Engineering a safer world: systems thinking applied to safety (engineering systems)”. In: *MIT Press Cambridge* (2011) (cit. on p. 20).

[Lev16] N. G. Leveson. *Engineering a safer world: Systems thinking applied to safety*. The MIT Press, 2016 (cit. on p. 20).

[Lev95] N. G. Leveson. *Safeware: system safety and computers*. Addison-Wesley, 1995 (cit. on p. 1).

[LGTL85] W.-S. Lee, D. L. Grosh, F. A. Tillman, C. H. Lie. “Fault Tree Analysis, Methods, and Applications A Review”. In: *IEEE transactions on reliability* 34.3 (1985), pp. 194–203 (cit. on p. 8).

[Mac17] J. Mace. “End-to-End Tracing: Adoption and Use Cases”. In: *Survey, Brown University* (2017) (cit. on p. 45).

[Max08] J. A. Maxwell. “Designing a qualitative study”. In: *The SAGE handbook of applied social research methods* 2 (2008), pp. 214–253 (cit. on p. 75).

[MNPF95] J. A. McDermid, M. Nicholson, D. J. Pumfrey, P. Fenelon. “Experience with the application of HAZOP to computer-based systems”. In: *COMPASS’95 Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security’*. IEEE. 1995, pp. 37–48 (cit. on pp. 21, 22).

[MP00] J. A. McDermid, D. J. Pumfrey. “Assessing the safety of integrity level partitioning in software”. In: *Proceedings of the Eighth Safety-critical Systems Symposium, Southampton UK*. Citeseer. 2000, pp. 134–152 (cit. on pp. 23, 24).

[MP94] J. McDermid, D. Pumfrey. “A development of hazard analysis to aid software design”. In: *Proceedings of COMPASS’94-1994 IEEE 9th Annual Conference on Computer Assurance*. IEEE. 1994, pp. 17–25 (cit. on p. 21).

[NE00] B. Nuseibeh, S. Easterbrook. “Requirements engineering: a roadmap”. In: *Proceedings of the Conference on the Future of Software Engineering*. 2000, pp. 35–46 (cit. on p. 1).

[Nea62] R. Neal. “Modes of Failure Analysis Summary for the Nerva B-2 Reactor”. In: *Westinghouse Electric Corporation Astronuclear Laboratory* (1962) (cit. on p. 19).

[New15] S. Newman. *Building microservices: designing fine-grained systems*. Ö’Reilly Media, Inc.”, 2015 (cit. on p. 11).

[NL03] C. J. Neill, P. A. Laplante. “Requirements engineering: the state of the practice”. In: *IEEE software* 20.6 (2003), pp. 40–45 (cit. on p. 1).

[NL94] J. Nielsen, J. Levy. “Measuring usability: preference vs. performance”. In: *Communications of the ACM* 37.4 (1994), pp. 66–75 (cit. on p. 75).

[NOC11] P. M. Nadkarni, L. Ohno-Machado, W. W. Chapman. “Natural language processing: an introduction”. In: *Journal of the American Medical Informatics Association* 18.5 (2011), pp. 544–551 (cit. on p. 31).

[OBM+20] D. Okanović, S. Beck, L. Merz, C. Zorn, L. Merino, A. van Hoorn, F. Beck. “Can a chatbot support software engineers with load testing? Approach and experiences”. In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 120–129 (cit. on pp. 58, 62).

[OGX11] A. Oliner, A. Ganapathi, W. Xu. “Advances and challenges in log analysis”. In: *Queue* 9.12 (2011), pp. 30–40 (cit. on p. 27).

[ONK17] A. Ora, A. Nandan, A. Kumar. “Hazard Identification of Chemical Mixing Plant through Hazop Study”. In: *International Journal for Advance Research and Development* 2.3 (2017) (cit. on p. 13).

[PEM03] F. Paetsch, A. Eberlein, F. Maurer. “Requirements engineering and agile software development”. In: *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.* IEEE. 2003, pp. 308–313 (cit. on p. 1).

[Pet19] J. S. Peterson. “Presenting a qualitative study: A reviewer’s perspective”. In: *Gifted Child Quarterly* 63.3 (2019), pp. 147–158 (cit. on p. 75).

[PGL17] S. Pérez-Soler, E. Guerra, J. de Lara. “Assisted Modelling Over Social Networks with SOCIO.” In: *MODELS (Satellite Events)*. 2017, pp. 561–565 (cit. on pp. 2, 37).

[Pnu77] A. Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE. 1977, pp. 46–57 (cit. on p. 30).

[Poh10] K. Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010 (cit. on p. 8).

[PSR10] D. Pandey, U. Suman, A. Ramani. “Social-organizational participation difficulties in requirement engineering process: A study”. In: *Software Engineering* 1.1 (2010), p. 1 (cit. on p. 1).

[Pum99] D. J. Pumfrey. “The principled design of computer system safety analyses.” PhD thesis. University of York, 1999 (cit. on pp. 21–24).

[RD97] E. Reiter, R. Dale. “Building applied natural language generation systems”. In: *Natural Language Engineering* 3.1 (1997), pp. 57–87 (cit. on p. 31).

[Rie19] T. Rietz. “Designing a conversational requirements elicitation system for end-users”. In: *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE. 2019, pp. 452–457 (cit. on pp. 2, 33, 34).

[RM95] G. Rugg, P. McGeorge. “Laddering”. In: *Expert Systems* 12.4 (1995), pp. 339–346 (cit. on p. 34).

[RS15] E. Ruijters, M. Stoelinga. “Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools”. In: *Computer science review* 15 (2015), pp. 29–62 (cit. on pp. 14, 16).

[SA20] N. Sabharwal, A. Agrawal. “Introduction to Google dialogflow”. In: *Cognitive virtual assistants using Google Dialogflow*. Springer, 2020, pp. 13–54 (cit. on p. 32).

[SBB+10] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag. “Dapper, a large-scale distributed systems tracing infrastructure”. In: (2010) (cit. on pp. 27, 28).

[SEM12] R. Schatz, S. Egger, K. Masuch. “The impact of test duration on user fatigue and reliability of subjective quality ratings”. In: *Journal of the Audio Engineering Society* 60.1/2 (2012), pp. 63–73 (cit. on p. 98).

[SFSG14] R. R. Sambasivan, R. Fonseca, I. Shafer, G. R. Ganger. “So, you want to trace your distributed system? Key design insights from years of practical experience”. In: *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL-14* (2014) (cit. on pp. 26–28, 45).

[SGS+19] C. S. R. K. Surana, D. B. Gupta, S. P. Shankar, et al. “Intelligent chatbot for requirements elicitation and classification”. In: *2019 4th International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT)*. IEEE. 2019, pp. 866–870 (cit. on pp. 2, 33, 36).

[Shk15] Y. Shkuro. https://eng.uber.com/distributed-tracing/. Accessed: 2020-12-9. 2015 (cit. on p. 29).

[Shk19] Y. Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd, 2019 (cit. on p. 45).

[SI01] B. Standard, B. IEC61882. “Hazard and Operability Studies (HAZOP studies)-Application Guide”. In: *BS IEC* 61882 (2001) (cit. on p. 11).

[SNZP16] N. Sehatbakhsh, A. Nazari, A. Zajic, M. Prvulovic. “Spectral profiling: Observer-effect-free profiling by monitoring EM emanations”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–11 (cit. on p. 26).

[Som17] J. Somers. “The coming software apocalypse”. In: *The Atlantic* 26 (2017) (cit. on p. 1).

[Son12] Y. Song. “Applying system-theoretic accident model and processes (STAMP) to hazard analysis”. PhD thesis. 2012 (cit. on p. 20).

[Sta80] U. M. Standard. “Mil-std-1629a”. In: *Procedures for Performing a Failure Mode, Effect and Criticality Analysis, Department of Defense, USA* (1980) (cit. on p. 19).

[Twi12] Twitter. https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin.html. Accessed: 2020-12-9. 2012 (cit. on p. 28).

[VGRH81] W. E. Vesely, F. F. Goldberg, N. H. Roberts, D. F. Haasl. *Fault tree handbook*. Tech. rep. Nuclear Regulatory Commission Washington DC, 1981 (cit. on pp. 8, 14–18).

[VIC06] T. M. Veludo-de-Oliveira, A. A. Ikeda, M. C. Campomar. “Discussing laddering application by the means-end chain theory”. In: *The Qualitative Report* 11.4 (2006), pp. 626–642 (cit. on p. 34).

[Wat+61] H. Watson et al. “Launch control safety study”. In: *Bell labs* (1961) (cit. on pp. 8, 14).

[WAVV12] K. Wolter, A. Avritzer, M. Vieira, A. Van Moorsel. *Resilience assessment and evaluation of computing systems*. Springer, 2012 (cit. on p. 1).

[WJG01] R. Winther, O.-A. Johnsen, B. A. Gran. “Security assessments of safety critical systems using HAZOPs”. In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2001, pp. 14–24 (cit. on pp. 11, 23).

[WOP09] V. M. B. Werneck, A. d. P. A. Oliveira, J. C. S. do Prado Leite. “Comparing GORE Frameworks: i-star and KAOS.” In: *WER*. 2009 (cit. on p. 36).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012 (cit. on pp. 97–99).

[YDW+19] K. Yin, Q. Du, W. Wang, J. Qiu, J. Xu. “On representing and eliciting resilience requirements of microservice architecture systems”. In: *arXiv preprint arXiv:1909.13096* (2019) (cit. on pp. 11, 39–41).

[Zip12] Zipkin. https://zipkin.io/. Accessed: 2020-12-9. 2012 (cit. on p. 28).

[Zor21a] C. Zorn. https://github.com/Cambio-Project/hazard-elicitation. Accessed: 2021-05-17. 2021 (cit. on pp. 4, 5).

[Zor21b] C. Zorn. https://zenodo.org/record/4781368. Accessed: 2021-05-22. 2021 (cit. on p. 5).

[ZPX+19] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, C. He. “Latent error prediction and fault localization for microservice applications by learning from system trace logs”. In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by M. Dumas, D. Pfahl, S. Apel, A. Russo. ACM, 2019, pp. 683–694. DOI: 10.1145/3338906.3338961. URL: https://doi.org/10.1145/3338906.3338961 (cit. on pp. 77, 78).

All links were last followed on May 20, 2021.

A Study Documents

A.1 Study Invitation

Interactive Elicitation of Resilience Scenarios in Microservice Architectures — Study Invitation

Hello,

In one of our students' thesis, which is part of the Cambio research project, we would like to evaluate the prototype that was developed during the thesis. We invite you to this study because your background in resilience engineering is of great relevance. The purpose of this study is to find out how beneficial an interactive elicitation of resilience scenarios can be.

The study will be conducted remotely, with the meeting software Cisco Webex which is available as a desktop application or via a browser. We would like to record the meeting to have a more reliable way of evaluating the study session.

In order to inform you about your rights and data collection during the study, we created a consent form. Please read this form before the study if you decide to participate.

Besides the meeting software, the only thing you need to participate in is a modern browser and a monitor resolution equal to or bigger than 1600x900. A higher resolution is preferable but not necessary.

The study will take approximately 45-60 minutes and is divided into three parts:

1. Introduction and familiarization (~15 minutes):
   You are given a high-level overview of the project and the goals that we aim to achieve. Furthermore, you will be introduced to the prototype that is used in the study. Afterward, you will be given task descriptions that define what you have to do in the study.

2. Task execution (~15 minutes):
   In the second part, you will work on your own to complete the given tasks.

3. Questionnaire and feedback (~15-30 minutes):
   Once you are finished with the task, you will be asked to answer questions about your experience with the prototype.

If you want to know more about the study or the project, write a mail to the study conductor (Christoph Zorn) st115403@stud.uni-stuttgart.de.

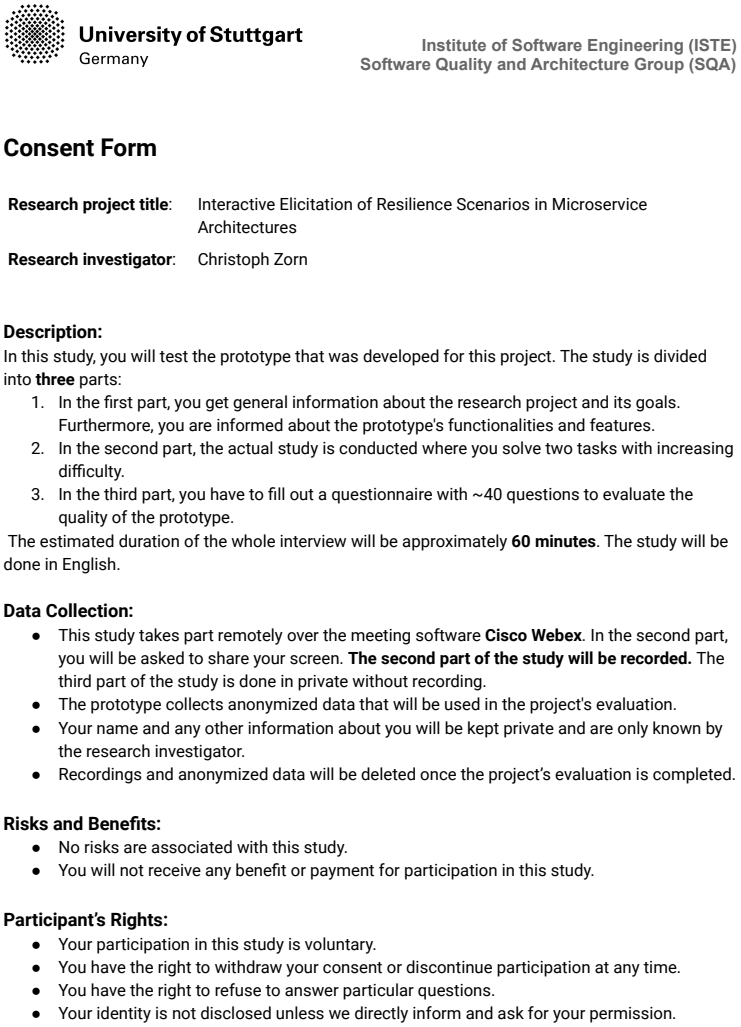
Thank you very much for your interest. We are looking forward to your participation.

With best regards,

The Cambio team

Figure A.1: The study invitation that every potential participant received.

A.2 Study Consent Form



University of Stuttgart
Germany

Institute of Software Engineering (ISTE)
Software Quality and Architecture Group (SQA)

Consent Form

Research project title:	Interactive Elicitation of Resilience Scenarios in Microservice Architectures
Research investigator:	Christoph Zorn

Description:
In this study, you will test the prototype that was developed for this project. The study is divided into **three** parts:

1. In the first part, you get general information about the research project and its goals. Furthermore, you are informed about the prototype's functionalities and features.
2. In the second part, the actual study is conducted where you solve two tasks with increasing difficulty.
3. In the third part, you have to fill out a questionnaire with ~40 questions to evaluate the quality of the prototype.

The estimated duration of the whole interview will be approximately **60 minutes**. The study will be done in English.

Data Collection:

- This study takes part remotely over the meeting software **Cisco Webex**. In the second part, you will be asked to share your screen. **The second part of the study will be recorded.** The third part of the study is done in private without recording.
- The prototype collects anonymized data that will be used in the project's evaluation.
- Your name and any other information about you will be kept private and are only known by the research investigator.
- Recordings and anonymized data will be deleted once the project's evaluation is completed.

Risks and Benefits:

- No risks are associated with this study.
- You will not receive any benefit or payment for participation in this study.

Participant's Rights:

- Your participation in this study is voluntary.
- You have the right to withdraw your consent or discontinue participation at any time.
- You have the right to refuse to answer particular questions.
- Your identity is not disclosed unless we directly inform and ask for your permission.

Figure A.2: The study consent from that every potential participant recieved with the invitation.

A.3 Study Task Description

All potential participants that agreed to the study meeting received the following seven-page task description after the familiarization. Since two separate systems were used in the study based on the participants, the description of the respective system is different. For this purpose, only the seventh page of the task description is different.

I. Purpose and Goal

Requirements elicitation is used to identify concerns of stakeholders that share an interest in a system. In traditional elicitation approaches the stakeholders would meet in-person and discuss threats and hazards of the system at design-time and create scenarios for the vulnerabilities of the system.
Typical scenarios would cover the use cases of failing components or deviations from the normal operation of the system. This requires a division of the system into important components. Therefore, the quality and completeness of the scenarios depend on the people involved and their knowledge of the system.

One drawback of this approach is the consumption of the stakeholders' working time. The elicitation process usually takes at least half a day with around five experts to complete. Another potential drawback is the incompleteness of the scenarios. As the traditional approach is done during design time it is not guaranteed that all components and threats are covered by the scenarios.

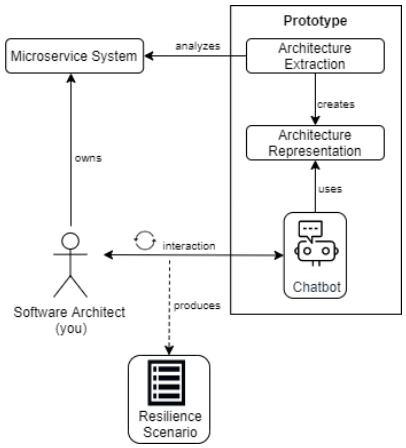


Figure 1: Processes and actors in the study.

With the approach presented in this study, we want to accelerate this process while maintaining the quality offered by the traditional approach.
Figure 1 shows the concept of this study.

In contrast to the traditional approach, which is usually top-down, we use the bottom-up perspective with an existing microservice software system.

Figure A.3: The first page of the task description.

- In the first step, the system is analyzed with a tracing tool to create an architecture description and collect available metrics from traces.
- In the second step, the architecture description is used to create an architecture graph and highlight potential sources of failure to the user.
- In the third step, the actual elicitation takes place where the users can define the resilience scenario with the help of a chatbot that provides them with options from the analysis. However, if the users are not satisfied with the presented options they can use their own knowledge to complete the resilience scenario.

The goal of this approach is to use the properties defined in the scenarios to configure a simulation (which is part of another thesis) to test the hypothesis elicited in the scenarios.
The approach is meant to have multiple iterations to continuously improve the weak components of the system.

Figure A.4: The second page of the task description.

II. Explanation of the prototype - Resirio

In the study, you will use a prototype developed to ease the elicitation of resilience scenarios.
The user interface of the prototype, depicted in Figure 2, consists of one window that is divided into three components.

1. Chatbot (Figure 2):

   In the study, you will have to interact with the chatbot to create scenarios. The chatbot will give you instructions on what you have to do and ask you to fill in missing parts of the scenario. It is also able to answer questions on the topic of resilience and tracing.
   The idea behind the chatbot is to elicit scenarios faster than with traditional methods. Therefore, the chatbot provides you with quick responses that contain preconfigured answers. However, to customize an answer to your liking, you can still write to the chatbot via the text input.

2. Architecture graph (Figure 3):

   In the architecture visualization, you will see a graph of architecture components (services and operations). You can change the layout of the graph by zooming or dragging the graph. The graph was created with a tracing tool and provides you with information that was saved in the trace. This information is accessible with a context menu. Right-click on one of the components and you will see a dialogue to inspect it.
   The graph also provides a legend on the top left to show the priority of the components of the graph. The priority (shown in Figure 4.) describes the importance of this component within the architecture.

3. Management view (Figure 5, Figure 6):

   The management view is located at the bottom and used to configure the behavior of the graph and the appearance of the prototype. It also shows information about the elicited scenarios once they are saved.

Figure A.5: The third page of the task description.

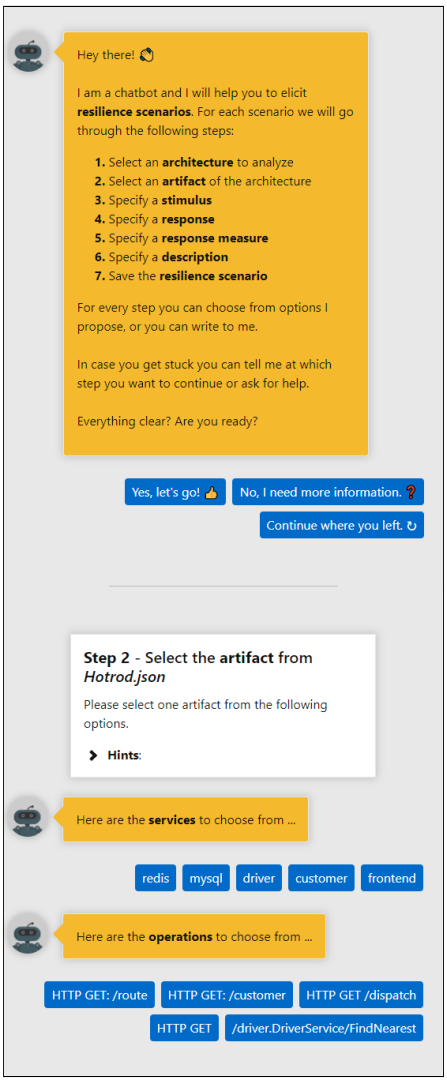


Figure 2: Screenshot of the chatbot component.

Figure A.6: The fourth page of the task description.

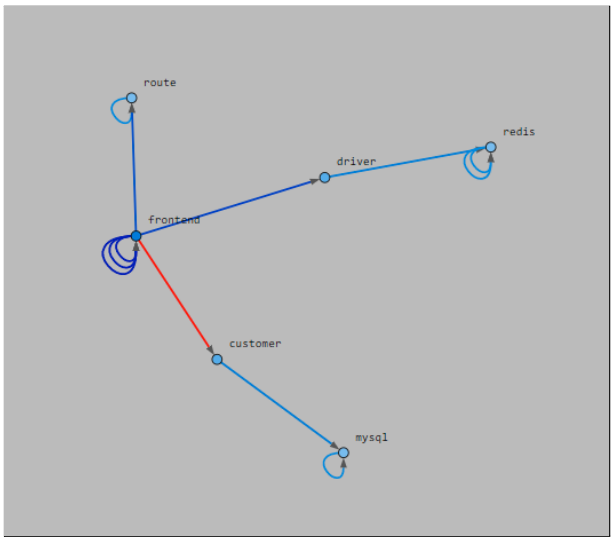


Figure 3: Screenshot of the architecture graph component.

Figure 4: Screenshot of the architecture graph legend.

Figure A.7: The fifth page of the task description.

Figure 5: Screenshot of the management component showing the configuration panel.

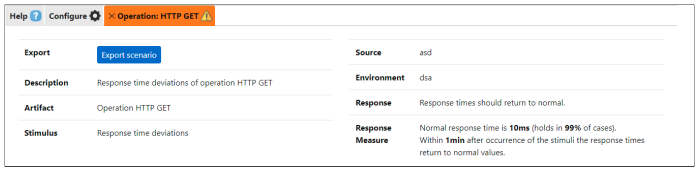


Figure 6: Screenshot of the management component showing a finished scenario.

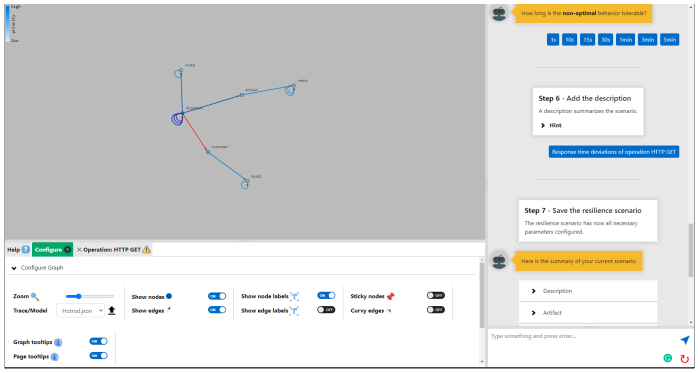


Figure 7: Screenshot of the all prototype components.

Figure A.8: The sixth page of the task description.

III. Study Description

You are a software architect and responsible for the resilience of the **TrainTicket** system. Your task is to elicit resilience scenarios. The scenario should provide the basis for testing the system's resilience against potential hazards. The specification that you define in the scenarios will be tested later (not part of the study).

You are given access to a newly developed prototype (Resirio) that is capable of eliciting resilience scenarios in an interactive manner. Solve the following two tasks with Resirio.

Task 1

In the last few days, the seat service (ts-seat-service) has had some problems. Customers complained that the user interface was responding slowly. From your experience, you know that these problems usually happen, once you deploy the latest version of the application.

Create and save (as shown in Figure 6) a resilience scenario with the following parameters:

Description	Response time deviations of operation getLeftTicketOfInterval
Artifact	getLeftTicketOfInterval
Stimulus	response time deviations
Source	a deployment error
Environment	during deployment
Response	Response times should return to normal values.
Response Measure	
• **optimal response time**	100 milliseconds
• **should hold in**	95% of all cases
• **recovery time**	10 seconds

Task 2

The specifications that you defined in task 1, were tested in a simulation experiment.
The result of the experiment showed that only one instance of the seat service was available at any time. The problem in the deployment file was fixed. However, customers still complain about another problem:

During the configuration of food for a trip, the user interface did not respond and showed an error. When users tried to access the same service again after some time, the service was still not available. A day later the customers could access the service again, yet the problem of crashing still remains.

Create and save a second scenario with a different artifact. Keep in mind the description above during the specification of the scenario. The scenario should give the simulation experiment enough information to identify the problem.
Once you are finished, tell the chatbot that you don't want to create another scenario.

Figure A.9: The seventh page of the task description for participants that were inspecting the TrainTicket system.

III. Study Description

You are a software architect and responsible for the resilience of the [redacted] system. Your task is to elicit resilience scenarios. The scenario should provide the basis for testing the system's resilience against potential hazards. The specification that you define in the scenarios will be tested later (not part of the study).

You are given access to a newly developed prototype (Resirio) that is capable of eliciting resilience scenarios in an interactive manner. Solve the following two tasks with Resirio.

Task 1

In the last few days, the [redacted] service ([redacted]) has had some problems. The monitoring system showed that response times vary from time to time. From your experience, you know that this problem usually occurs, once you deploy the latest version of the application.

Create and save (as shown in Figure 6) a resilience scenario with the following parameters:

Description	Response time deviations of operation [redacted]
Artifact	[redacted]
Stimulus	response time deviations
Source	a deployment error
Environment	during deployment
Response	Response times should return to normal values.
Response Measure	
• **optimal response time**	100 milliseconds
• **should hold in**	95% of all cases
• **recovery time**	10 seconds

Task 2

The specifications that you defined in task 1, were tested in a simulation experiment.
The result of the experiment showed that only one instance of the [redacted] service was available at any time. The problem in the deployment file was fixed. However, the monitoring system shows another problem:

In the last week, the [redacted] service ([redacted]) regularly returned error codes when the [redacted] endpoint was accessed. The errors occurred mostly during nightly batch processing tasks and non-office hours. After the occurrence of every error, the monitoring system showed that the [redacted] service was not available for a short time.

Create and save a second scenario with a different artifact. Keep in mind the description above during the specification of the scenario. The scenario should give the simulation experiment enough information to identify the problem.
Once you are finished, tell the chatbot that you don't want to create another scenario.

Figure A.10: The seventh page of the task description for participants that were inspecting the industry system (we excluded confidential names of services and operations).

A.4 Study Questionnaire

In the following, the questionnaire used in the study is given. Circles (∘) imply that it is possible to give only one answer to this question. Boxes (□) imply that it is possible to give multiple answers. Empty texts (…) imply that custom full-text answers are allowed.

Your Background

1. How would you describe your current role in your company/university?
   - ∘ Software Architect
   - ∘ Resilience Engineer
   - ∘ Doctoral Researcher
   - ∘ …

2. How would you describe your confidence in regards to resilience engineering?
   - ∘ 1 (less confident)
   - ∘ 2
   - ∘ 3
   - ∘ 4
   - ∘ 5 (very confident)

3. Did you know about resilience scenarios before this study?
   - ∘ Yes
   - ∘ No

4. Have you ever been part of the elicitation process to create resilience scenarios before this study?
   - ∘ Yes
   - ∘ No

Questions about the task & scenario

1. How challenging was it to solve the task 1?
   - ∘ 1 (very easy)
   - ∘ 2
   - ∘ 3
   - ∘ 4
   - ∘ 5 (very hard)

2. How challenging was it to solve the task 2?
   - 1 (very easy)
   - 2
   - 3
   - 4
   - 5 (very hard)

3. Did you complete the given tasks successfully?
   - Yes
   - No

4. Could you have solved the tasks without the help of the prototype?
   - Yes
   - No

5. Which approach to elicit resilience scenarios is faster?
   - Interactive Approach (Chatbot)
   - Traditional Approach (Group Meeting)

6. Which approach to elicit resilience scenarios produces more useful scenarios for resilience specifications?
   - Interactive Approach (Chatbot)
   - Traditional Approach (Group Meeting)

7. Do you think that resilience scenarios created with the prototype provide enough information to test a systems' resilience?
   - Yes
   - No

Statements about the prototype's usability

1. I think that I would like to use this system frequently.
   - 1 (strongly disagree)
   - 2
   - 3
   - 4
   - 5 (strongly agree)

2. I found the system unnecessarily complex.
   - 1 (strongly disagree)
   - 2
   - 3

- 4
- 5 (strongly agree)

3. I thought the system was easy to use.
   - 1 (strongly disagree)
   - 2
   - 3
   - 4
   - 5 (strongly agree)

4. I think that I would need the support of a technical person to be able to use this system.
   - 1 (strongly disagree)
   - 2
   - 3
   - 4
   - 5 (strongly agree)

5. I found the various functions in this system were well integrated.
   - 1 (strongly disagree)
   - 2
   - 3
   - 4
   - 5 (strongly agree)

6. I thought there was too much inconsistency in this system.
   - 1 (strongly disagree)
   - 2
   - 3
   - 4
   - 5 (strongly agree)

7. I would imagine that most people would learn to use this system very quickly.
   - 1 (strongly disagree)
   - 2
   - 3
   - 4
   - 5 (strongly agree)

8. I found this system very cumbersome to use.
   - 1 (strongly disagree)
   - 2
   - 3

- 4
- 5 (strongly agree)

9. I felt very confident using this system.
   - 1 (strongly disagree)
   - 2
   - 3
   - 4
   - 5 (strongly agree)

10. I needed to learn a lot of things before I could get going with this system.
    - 1 (strongly disagree)
    - 2
    - 3
    - 4
    - 5 (strongly agree)

Questions about the prototype's features

1. Describe the chatbot or your interaction with the chatbot in one word.
   - pleasant
   - unpleasant
   - neutral
   - helpful
   - annoying
   - …

2. Does the chatbot provide enough assistance during the elicitation?
   - Yes
   - No

3. Does the chatbot provide enough quick replies (blue buttons)?
   - Yes
   - No

4. Are the chatbot's quick replies suitable enough to elicit a useful resilience scenario?
   - Yes
   - No

5. Could you follow the flow of the conversation?
   - Yes
   - No

6. Did you get stuck during any of the tasks?
   - ○ Yes
   - ○ No

7. If you did get stuck, where did you get stuck, and what was the cause?
   - ○ …

8. Which features of the prototype did you use?
   - □ Chat (text input)
   - □ Quick Responses (blue buttons)
   - □ Chatbot Guide (explanations about a topic)
   - □ Chatbot Reset (start a conversation from the beginning)
   - □ Architecture Graph
   - □ Management View (configurations for the prototype)

9. Which feature(s) of the prototype did you like?
   - □ Architecture Graph
   - □ Chatbot
   - □ Chatbot Explanations
   - □ Chatbot Quick Responses
   - □ User Interface
   - □ Nothing

10. Why did you like these feature(s)?
    …

11. Which feature(s) of the prototype did you dislike?
    - □ Architecture Graph
    - □ Chatbot
    - □ Chatbot Explanations
    - □ Chatbot Quick Responses
    - □ User Interface
    - □ Nothing

12. Why did you dislike these feature(s)?
    …

13. What features are missing in the prototype?
    …

14. How would these features have helped you?
    …

15. Would you use the prototype in the future or recommend it?
    …

16. For what reason would you use or recommend the prototype?
    …

Wrap-up

1. If you have more thoughts about the prototype or if you feel there is something that was not talked about in the questionnaire, you can expand here:
   …

B Screenshots of the Prototype

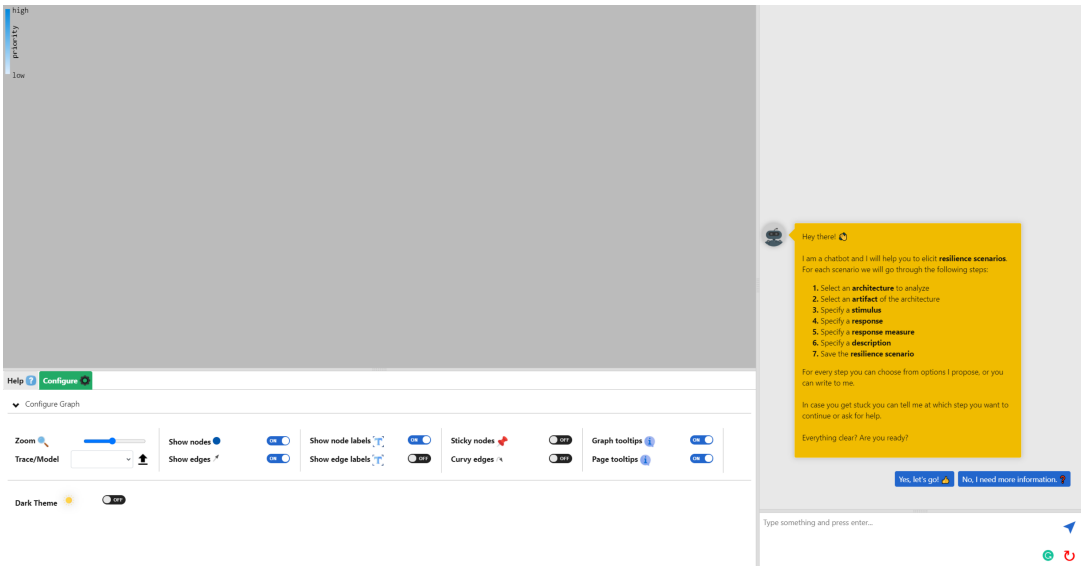


Figure B.1: The chatbot welcomes users see when they first access the prototype.

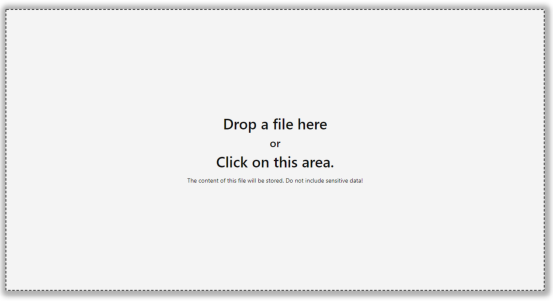
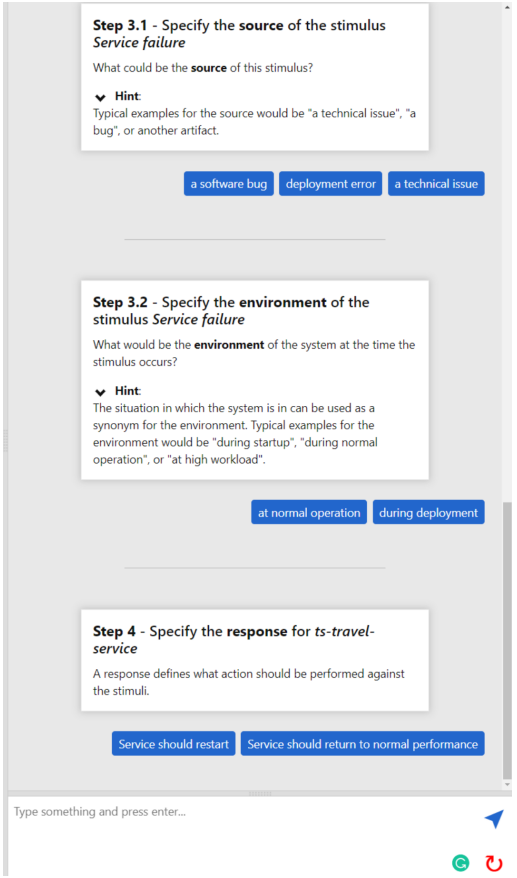
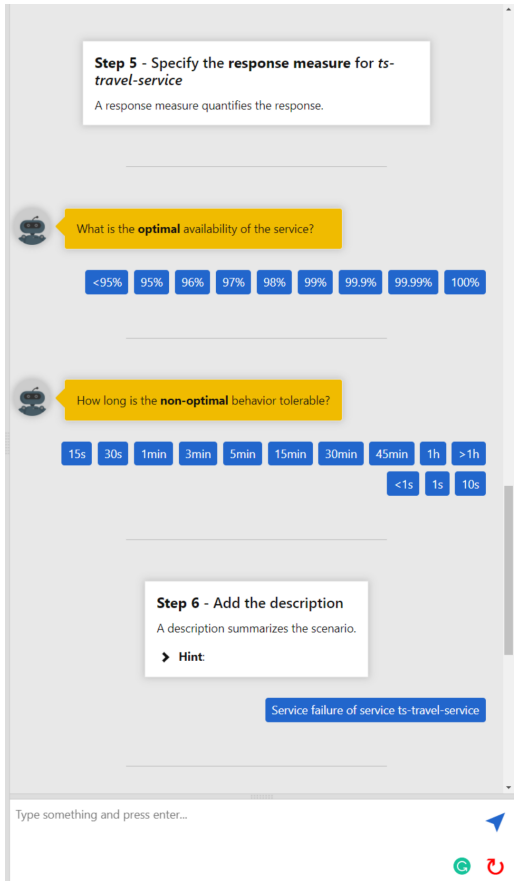


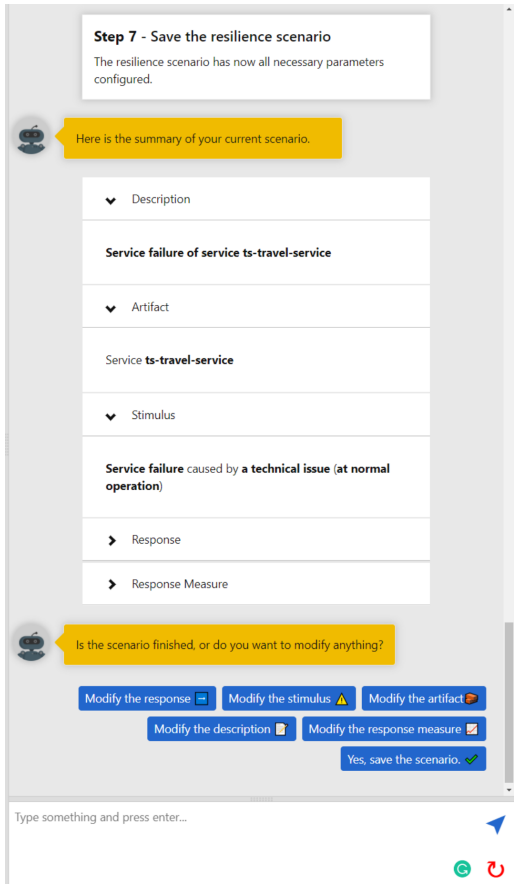
Figure B.2: The upload view where users can upload their trace models. The trace model is analyzed automatically. After analyzing, the users are redirected to the main page.



(a) The chatbot tells users for which step of the prototype they specify a parameter. In addition it provides hint for custom inputs.



(b) The response measure consists of two values for services and three values of operations.



(c) In the summary, an overview of configured parameters is shown. Users have the chance to change specific parameters or save the scenario.

Figure B.3: Excerpts of a conversation with the chatbot.

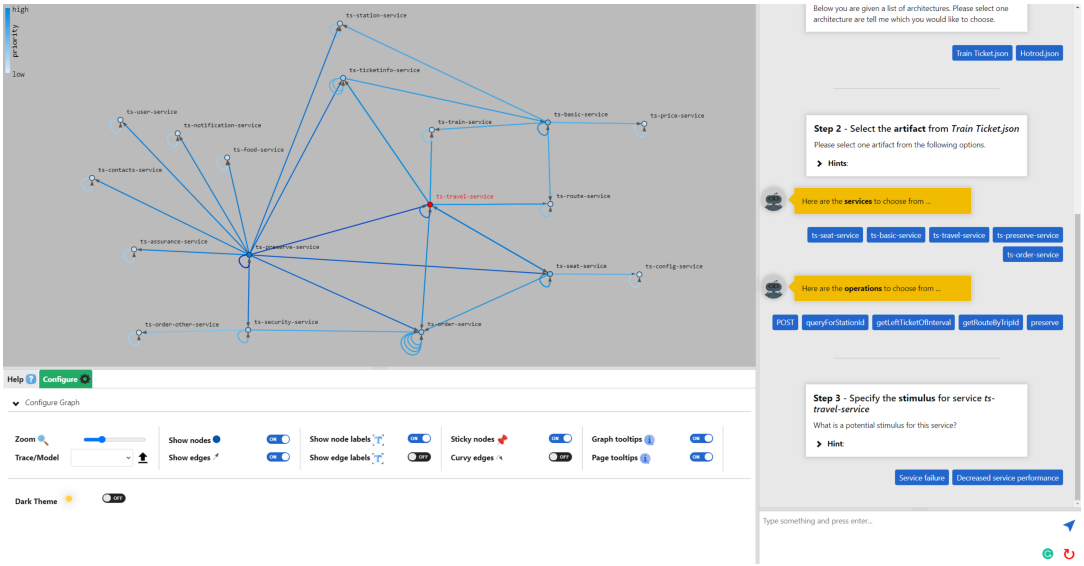


Figure B.4: The architecture graph is visualized after the selection of a trace model. The chatbot provides the five most important services and operations to users. A selected component of the graph is highlighted with red color.

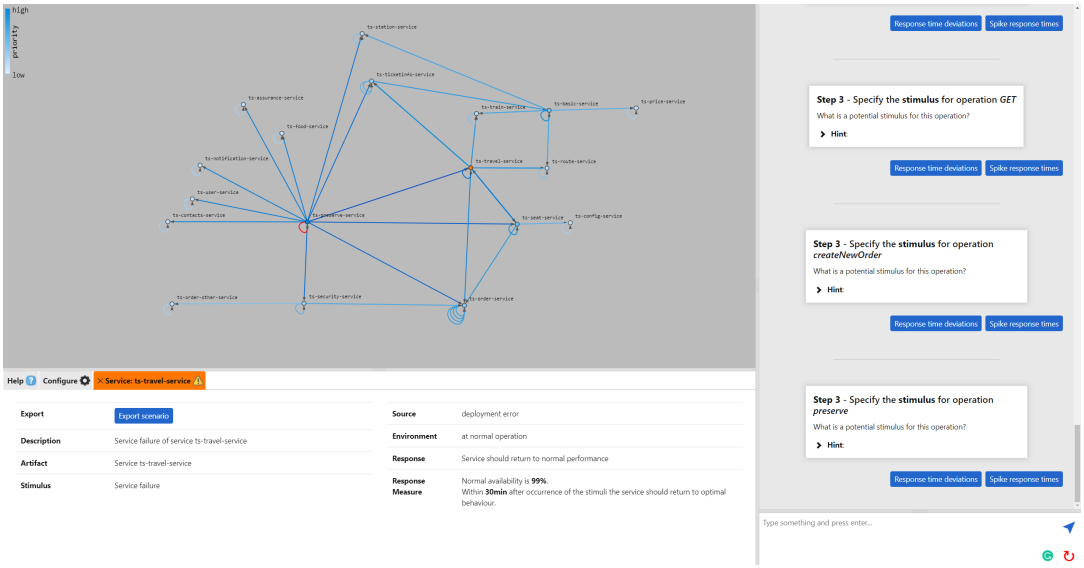


Figure B.5: After a scenario is saved, it is visible in the configuration view with all the specified parameters. A scenario is also highlighted in the architecture graph with an orange color.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature