

Institute of Software Engineering

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Masterarbeit

**Towards Empirical Evidence for  
the Impact of Microservice API  
Patterns on Software Quality:  
A Controlled Experiment**

Pawel Wójcik

**Course of Study:** Informatik

**Examiner:** Prof. Dr. Stefan Wagner

**Supervisor:** Dr. Justus Bogner

**Commenced:** September 22, 2022

**Completed:** March 22, 2023



## Abstract

Encapsulating parts of a software system into separate, small, independent modules, called microservices, is currently very popular in software development. Design patterns are templates that can help professionals to solve commonly occurring problems. Currently, there exists only a sparse amount of empirical evidence that supports the positive impact on the quality of a software, by incorporating the quite newly defined Microservice API patterns. Therefore, the purpose of this research is to examine the impact of 6 chosen Microservice API patterns on software quality, with a conducted controlled experiment with 65 participants. For each of the six design patterns, two tasks were created. Each task has a Pattern version P, where the scenario actively uses one of the six design patterns, and a Non-Pattern version N, where an almost identical scenario was constructed without the use of the respective pattern. For the web-based online survey, the participants were randomly put into one of two groups. Depending on the group, different task versions were shown to the participants. A crossover design with a fixed task order was used. The participants were asked to answer 12 comprehension questions. The correctness and the needed time for responding were tracked and aggregated for the analysis into one variable: Timed Actual Understandability (TAU). The results show that for five out of six Microservice patterns, the software quality was indeed increased significantly. Only the *Request Bundle* pattern did not increase the software quality significantly. The effect size for all significant findings was of either medium or small size. The results of this research provide empirical evidence for the improvement of two software quality attributes. Further research is needed if a generalization to a broader set of design patterns or to additional quality attributes is desired. Additionally, future research could focus on tasks in which several design patterns are combined, to get results for scenarios that are closer to environments in practice.



## Kurzfassung

Das Aufspalten von Softwaresystemen in kleine, unabhängige Module, sogenannte 'Microservices', ist aktuell sehr beliebt in der Softwareentwicklung. 'Design Patterns' dt. *Entwurfsmuster* sind Vorlagen, welche von Fachleuten eingesetzt werden können, um häufig auftretende Probleme einfacher und struktureller zu lösen. Aktuell existiert nur eine kleine Menge an empirischen Belegen dafür, dass die Softwarequalität durch den Einsatz von 'Microservice API Patterns (MAP)' verbessert wird. Deshalb ist das Ziel dieser Forschungsarbeit, den Einfluss von 6 ausgewählten Microservice API Patterns auf die Softwarequalität zu bestimmen. Diese Untersuchung wurde mit Hilfe eines kontrollierten Experiments mit 65 Teilnehmenden durchgeführt. Für jedes der sechs Patterns wurden jeweils zwei verschiedene Aufgaben erstellt. Jede Aufgabe besitzt eine Pattern-Version P, in der ein Pattern aktiv eingesetzt wird, sowie eine Non-Pattern Version N, wo ein ähnliches Szenario ohne den Einsatz eines Patterns vorgestellt wird. Für die Online-Umfrage wurden die Teilnehmenden zufällig einer von zwei Gruppen zugewiesen. Je nach Gruppe wurden den Teilnehmenden verschiedene Aufgabenversionen gezeigt. Das Experiment hat ein Crossover-Design mit festgelegter Aufgabenreihenfolge. Die Teilnehmenden sollten zwölf Verständnisfragen beantworten. Die Korrektheit und die benötigte Zeit zum Antworten wurden aufgezeichnet und zu einer Variable namens TAU zusammengefasst. Das Ergebnis zeigt, dass beim Einsatz von fünf der sechs Microservice Patterns, die festgestellte Softwarequalität signifikant gesteigert wurde. Nur das *Request Bundle* Pattern konnte keine signifikante Verbesserung der Softwarequalität erzielen. Die Effektgröße für die signifikanten Resultate war jeweils von kleinem oder mittlerem Ausmaß. Obwohl die Ergebnisse dieser Forschung empirische Belege für die Verbesserung von bestimmten Softwareattributen liefern, ist weitere Forschung unabdingbar, um eine Verallgemeinerung der Resultate auf eine größere Menge von Software Patterns zu ermöglichen. Außerdem könnte zusätzliche Forschung mit Aufgaben, in denen mehrere Patterns kombiniert eingesetzt werden, weitere wertvolle Ergebnisse für praxis-orientierte Szenarios liefern.



# Contents

<b>1</b>	<b>Introduction</b>	<b>15</b>
<b>2</b>	<b>Background</b>	<b>17</b>
2.1	Software Design Patterns . . . . .	17
2.2	Microservices . . . . .	20
2.3	Microservice API Patterns . . . . .	21
2.4	Software Quality . . . . .	23
2.5	Related Work . . . . .	24
<b>3</b>	<b>Experiment Planning</b>	<b>29</b>
3.1	Research Questions . . . . .	29
3.2	Dependent and Independent Variables . . . . .	30
3.3	Hypotheses . . . . .	30
3.4	Material . . . . .	31
3.5	Participants and Requirements . . . . .	31
3.6	Experiment Specifications . . . . .	32
3.7	Data Analysis . . . . .	41
<b>4</b>	<b>Results</b>	<b>43</b>
4.1	Participant Demographics . . . . .	43
4.2	Impact of Patterns (RQ1) . . . . .	46
4.3	Influence of Demographic Data (RQ2) . . . . .	51
<b>5</b>	<b>Discussion</b>	<b>53</b>
5.1	Interpretation of Results and Implications . . . . .	53
5.2	Threats to Validity . . . . .	55
<b>6</b>	<b>Conclusion and Outlook</b>	<b>57</b>
	<b>Bibliography</b>	<b>59</b>





## List of Figures

2.1	Cloud Computing Pattern <i>Transaction-Based Processor</i> [FLR+14]	17
2.2	Quantum Pattern <i>Quantum-Classic Split</i> [Ley19]	18
2.3	Architectural differences between a monolith and Microservice-based system	21
2.4	Three clients communicating via an <i>API Gateway</i> which is connected with four services (API Gateway pattern: Richardson [Ric18])	22
3.1	Overview of the crossover design, with the order of tasks for both groups	32
3.2	A single choice type tasks with the five answer options	33
3.3	A free-text type task with a text field	34
3.4	Pattern V1 (ER1P)	35
3.5	Non-Pattern V1 (ER1N)	35
3.6	Pattern V2 (PT2P)	36
3.7	Non-Pattern V2 (PT2N)	36
3.8	Pattern V2 (MD2P)	37
3.9	Non-Pattern V2 (MD2N)	38
3.10	Pattern and Non-Pattern V1 (WL1P/WL1N)	38
3.11	Pattern V2 (AG2P)	40
3.12	Non-Pattern V2 (AG2N)	40
4.1	Distribution of participants in regards to their years of experience in IT	43
4.2	Pie chart with the distribution of participants in regards of their professional background	44
4.3	Pie chart with the distribution of participants in regards of their main role	44
4.4	5-point Likert scale for three self-assessment topics	45
4.5	TAU values in contrast for the Error Report tasks	47
4.6	TAU values in contrast for the Parameter Tree tasks	48
4.7	TAU values in contrast for the Wish List tasks	48
4.8	TAU values in contrast for the Metadata tasks	49
4.9	TAU values in contrast for the API Gateway tasks	49
4.10	TAU values in contrast for the Request Bundle tasks	50



# List of Tables

- 4.1 Demographic data of the participants in regards to the two randomized groups  
(Mean experience: M.E.) . . . . . 45
- 4.2 Several statistical values for all pattern tasks and versions (N and P) . . . . . 46
- 4.3 Results of the hypothesis tests for all examined Microservice API patterns . . . . . 50



# Acronyms

- API** Application Programming Interface. 9, 15, 17, 20, 21, 22, 23, 24, 25, 27, 29, 30, 35, 37, 38, 39, 40, 44, 45, 49, 51, 54, 55, 57
- CSV** comma-separated values. 41
- GoF** Gang of Four. 17, 22, 26
- GUI** graphical user interface. 20, 24
- HCI** human-computer interaction. 24
- HTTP** Hypertext Transfer Protocol. 24, 29, 30, 34, 42, 44, 45, 51, 54, 55, 56, 57
- ID** identifier. 39, 40, 41
- IEC** International Electrotechnical Commission. 23
- IoT** Internet of Things. 20
- IP** Internet Protocol. 40
- ISO** International Organization for Standardization. 23
- IT** Information Technology. 9, 15, 21, 24, 29, 30, 31, 43, 45, 51, 55, 57
- M.E.** mean experience. 45
- M.Sc.** Master of Science. 31, 56
- MAP** Microservice API patterns. 3, 5, 11, 15, 17, 21, 22, 29, 30, 31, 35, 44, 45, 46, 50, 51, 53, 54, 55, 56, 57
- OOPSLA** Object-Oriented Programming, Systems, Languages and Applications. 17
- Ph.D.** Philosophiae Doctor. 31
- REST** Representational State Transfer. 25
- RQ** research question. 29, 30, 31, 42, 46, 51, 53
- SCS** safety-critical system. 23
- TAU** Timed Actual Understandability. 3, 5, 9, 30, 42, 46, 47, 48, 49, 50, 51, 53, 54, 55, 57
- V** version. 9, 35, 36, 37, 38, 40



# 1 Introduction

In the current Information Technology (IT) landscape, the encapsulation of parts of software systems into separate, small and independent software modules, called *microservices*, is very popular. To ease the work for developers when designing new or refining already existing microservices, *Microservice Application Programming Interface (API) design patterns* are being defined. These patterns are solution propositions for frequently occurring problems, when developing software. In 2019 the Microservice API patterns (MAP) were collected by Zimmermann et al. [ZSL+22] with the goal to bring the benefits that software design patterns promise, into the Microservice API environment. Richardson also conceptualized an assortment of patterns for the Microservice environment one year earlier, with examples in the *Java* programming language [Ric18]. While one of the main objectives of software design patterns, is to increase the software quality, there have not yet been much empirical evidence for the newly defined MAP. Due to the high popularity of Microservices in the industry, such evidence is crucial, to improve the development of software and potentially save a substantial amount of money. It has been also shown several times that the creation of a good API design is a nontrivial process. The use of design patterns might aid in these instances. Therefore, a controlled experiment with initially 98 participants from academia and industry was conducted. Out of two literature sources, six appropriate design patterns were chosen and two tasks per pattern were formulated. The experiment has a crossover design, in which the participants get randomly assigned to one of two groups and get to work on 12 out of 24 designed tasks. The participants had to answer comprehensions questions, for which the responses and the time that was needed per task, were recorded. The detailed design, analysis and results of this experiment will be presented in this Master's Thesis.

After introducing the core ideas of software design patterns, Microservices and MAP in Chapter 2, a small introduction to software quality and a presentation of related work will round off the background chapter. Chapter 3 will present the main stages of the preparation of the experiment, including the formulated research questions, defined variables and hypotheses. In Section 3.6, the details of the formulated survey tasks will be shown. Additionally, the chosen design patterns and the resulting tasks will be presented in detail, in the last part of this section. The processing and analysis of gathered participant data is then depicted in the last fragment of this chapter. Next, the results of the experiment are presented precisely with the aid of different statistical plots, tables and figures in Chapter 4. Subsequently, the just presented results will be interpreted and discussed in the following Chapter 5, which includes the presentation on possible threats to the validity of the findings. Finally, the conclusion in Chapter 6 represents the closing of this Master's Thesis, with a brief summary of the research objectives, the findings and the interpretation of the results. In addition, possible topics for future research are presented that could use the findings from this experiment as a basis.





## 2 Background

In this chapter basic information about the technical and historical background of software design patterns, the Microservice architecture and the Microservice API patterns is presented. This basic knowledge is recommended to fully understand the motivation, planning and analysis of the controlled experiment in the following chapters. Additionally, a compact segment on software quality covers the motivation and definitions of various concepts in this topic. At the end of this chapter, the related work of this research is laid out. It gives a solid overview on the status quo of the empirical findings that have already been conducted in regards to the usage of (API) software design patterns and the software quality that is influenced by the deployment of them.

### 2.1 Software Design Patterns

Originating in classical architecture, patterns are formulated templates that can help professionals of the field to solve commonly occurring obstacles or problems [AIS77]. They are not meant to solve every issue with a strict step-by-step instruction, but instead act as a guideline or collection of best-practices. Since the idea of design patterns can be useful in other engineering fields than classical architecture, the first ideas for design patterns in the context of software programming were presented by K. Beck and W. Cunningham at the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) conference in September 1987 [BC87]. The first big breakthrough for software design patterns occurred in 1994 after the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published by Gamma et al. and introduced many software design patterns that focus on problem solving in areas, like composing objects, the creation and management of objects or on the control flow [GHJV94]. The authors of this work are often referred to as the Gang of Four (GoF) in literature. While many defined patterns are applicable

#### Transaction-Based Processor

Components receive messages or read data and process the obtained information under a transactional context to ensure that all received messages are processed and all altered data is consistent after processing, respectively.

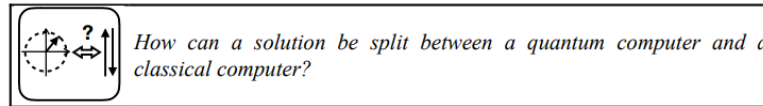


*How can an application component ensure that all messages it receives are processed successfully and altered data is persisted successfully after processing?*

**Figure 2.1:** Cloud Computing Pattern *Transaction-Based Processor* [FLR+14]

**2.10. Quantum-Classic Split**

**Intend:** The solution of a problem is often not achieved by only using a quantum computer. Thus, the solution is performed partially on a classical computer and partially on a quantum computer, and both parts of the solution interact.



**Context:** Some quantum algorithms inherently require pre- or post-processing on a classical device, resulting in a split of the solution into a classical part and a quantum part.

Also, if a quantum computer has a low number of qubits or its gates are noisy, a solution of a problem may have to be separated into a part executed on a quantum computer and a part executed on a classical computer [22].

**Solution:** The sheer fact that a split of the algorithms may be done is important. How such a split is applied is problem dependent.

**Know Uses:** Shor's algorithm or Simon's algorithm inherently make use of classical post-processing. The algorithm in [11] to solve combinatorial optimization problems uses classical pre-processing. The algorithm of [2] uses a split into a quantum part of the solution and a classical part to enable factorization on NISQ devices.

**Next:** Data is passed from the classical part of the solution to the quantum part by proper *initialization*.

**Figure 2.2:** Quantum Pattern *Quantum-Classic Split* [Ley19]

for a diverse set of software environments, some domains have unique common problems or constraints. Consequently, many domain-specific design patterns were introduced over the years by the respective communities. Examples are *user interface*, *business process*, *web design*, *quantum* and *cloud computing* patterns.

As an example, in Figure 2.1 the description of the *Transaction-Based Processor* pattern can be seen. The pattern is a solution template that helps to deal with data consistency in a transaction-based environment. From the quantum computing domain the *Quantum-Classic Split* pattern can be seen in Figure 2.2, which deals with the split of a solution between a quantum and a conventional machine.

Patterns can be classified in different ways, for example by differentiating in which way they want to solve a given problem or on which level of the system the problem typically occurs. Originally, the software design patterns were classified into three categories: **creational**, **behavioral** and **structural** patterns. Creational patterns depict templates with the goal to create an entity in a controlled way. An example is the *abstract factory* pattern which describes how to create a family of objects. Behavioral patterns are concerned with the communication between software entities and structural patterns present possible solution ideas for assembling different entities and providing new functionalities [GHJV94]. With time, the documentation of newly formulated patterns became more consistent, but there is not a single format for the definition of software design patterns. However, a scheme that is often used was provided by Gamma et al. [GHJV94] with the following structure:

**Pattern Name & Classification**

A distinctive name for this pattern and an assignment to a defined class of patterns

**Intent**

The purpose for using this pattern

**Context**

The environment in which this pattern should be used

**Structure**

A textual or graphical representation of the patterns composition

**Consequences**

A collection of effects and results of the usage of this pattern

**Implementation**

An elaboration of the implementation of the pattern

**Known Uses**

Usages of this pattern in a real environment

**Related Patterns**

Other patterns that have something in common with this pattern

Such descriptions should ideally reduce the needed time to understand the pattern, so it can be used in software systems by the responsible actors. However, a strong fixation on patterns is not always helpful. If the software system is not too large or complex, a simple implementation can be finalized quicker than learning, understanding and applying potentially more complicated design patterns [KG08]. Additionally, just understanding and deciding in favor of a design pattern, for example as a software architect or an engineer that is developing a new software system, is not the only aspect that has to be considered. Even if the pattern is used correctly and should thus improve the quality of the software in some way, this is not always guaranteed [KG08]. If software gets continuously developed over a long period of time, the size and complexity is usually increasing. New developers that have not worked with a deployed pattern might have difficulties understanding the source code. This results in additional time that is needed to learn about the pattern, before new features can be implemented or tests can be created. Another possible problem, especially with more complex or unintuitive software design patterns, is the inconsistency of implementations that can appear within one software system if some of the developers are familiar with certain patterns, while others cannot or do not want to work with these [Gef18]. Some criticism against design patterns come with the argumentation that they are needed, just because the used programming language or technology does not naively support the level of abstraction that is necessary to solve a problem [Nor96]. Another counterargument against the active use of design patterns is rooted in the common misunderstanding, mainly by beginners that just started to learn about them, that they are improving the quality of software systems universally and without exceptions. The consequence is that in some cases known patterns are tried to be used, while much simpler solutions would represent a better fit for the problem. The popular proverb *If all you have is a hammer, everything looks like a nail* that is most likely derived from Abraham Maslows work is frequently quoted when introducing or discussing software design patterns [Mas66].

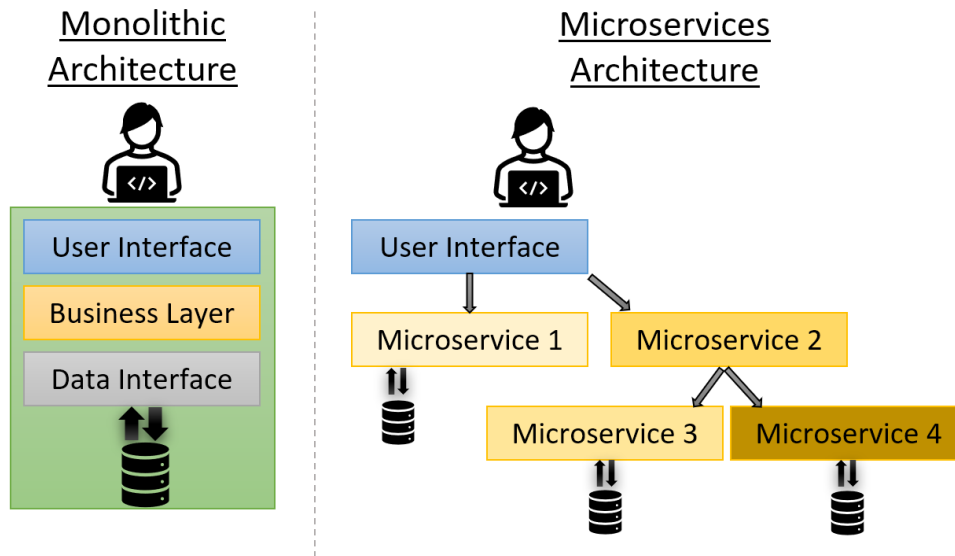
### 2.2 Microservices

The Microservice architecture or just *Microservices*, is an architectural design pattern. It describes a software system that consists out of several rather small, independent entities, such called *services*, that can communicate with each other over their defined API endpoints [BDD+19]. While this is a broad characterization for the concept of Microservices, there is not one precise definition for it. Presumably originating from the term *Micro-Web-Service* in the year of 2005 [Rod05], the idea of loosely coupled services that can interact with each other were officially named Microservices for the first time in 2011 at a software architect workshop in Venice, Italy.

The ongoing trend for Microservices can be reasoned mainly with the advantages that they promise, in contrast to a traditional monolithic architecture [BDD+19]. Further, the trend for small independent units with an external interface is present, because of the specifications in emerging areas like *cloud and mobile computing* or in the *Internet of Things* domain. Since microservices are mostly independent of each other, this results in a lot of additional *flexibility* on different levels. Two microservices that interact in some way with one other can use different technologies or can be implemented in distinctive programming languages. The machines where these services are deployed can be anywhere in the world and the position can be changed without investing a lot of resources, as long as these services can communicate over a common network, for example the internet. Since every microservice should be able to run independently, the *interchangeability* of them is possible without the need to adapt big parts of code in the rest of the software system. This also increases the *scalability*, since the respective microservice can be replaced with another that has more resources (vertical scaling). Alternatively, additional services can be added into the system which results in having several components that might possess the same set of functionalities (horizontal scaling). Another big advantage of Microservices is the high *reuseability*. Microservices usually fulfill a certain task and are loosely coupled. Therefore, they can sometimes be used in different software environments without too many necessary adjustments. Further, the *maintainability* is in general higher than in comparable monolithic systems, because it is easier to maintain and extend the independent, loosely coupled services. All of the just named advantages of Microservices enable additionally the continuous refactoring, integration and deployment of software parts and make programming of software in parallel more feasible [BHJ16].

In Figure 2.3, the differences in the basic structure of a monolithic and a Microservice-based system can be seen. While the monolithic system contains all components within its system borders, the system with a Microservice architecture has loosely coupled services with their own separate databases, while still being accessible for the user over one single graphical user interface (GUI).

Although the benefits of Microservices are manifold, the drawbacks and the criticism that is based on these have also to be considered. The first problem leads back to the fact that there is no strict definition for the term microservice and especially no size limit is defined, where upon exceeding the service would not be considered to be micro anymore (Greek *mikrós*: small/narrow) [Löv19]. In practice the scope of Microservices is usually abstracted from the business context [Zim19]. This is not trivial because if the individual services are too big in regards to functionality, many benefits from the Microservices pattern are not valid. On the contrary if the scope is too fine-grained, this can result in a hard to understand software environment with many tiny microservices, API endpoints and communication connections that all have to be implemented correctly [Til14]. Another catch about microservices and the ability to use different technologies for their development within a system is that software experts cannot be moved comfortably from one team to another because there might be no strict standardization, like in an equivalent monolithic software system. While the ease



**Figure 2.3:** Architectural differences between a monolith and Microservice-based system

of testing individual microservices is in principal a benefit of the pattern, it can be harder to test a Microservice-based system as a whole than testing a more traditional and uniform system. Moreover, new challenges that arise from the introduction of internal communication and the connections between the microservices have to be tackled. These challenges might be new to some developers, since they might not exist in a software entity with a different architectural structure [Löw19]. Plus, load balancing, latency within the system and the concept of message-based communication have to be acknowledged in a Microservices project [PZA+17]. The chance of failure during communication between the services is also higher in general, than in communication between software parts within a monolith.

## 2.3 Microservice API Patterns

The benefits of standardized template solutions are applicable in many different domains of the IT environment. Since the communication between microservices via their APIs is message-based and the Microservices architecture pattern introduces a whole set of new challenges, the demand for sophisticated design patterns for this environment is present. Therefore, Zimmermann et al. conceptualized a set with over 30 different Microservice API patterns (MAP) on the *Patterns for API Design* web page<sup>1</sup> and released a book with additional information in 2022 [ZSL+22]. The main motivation includes the just mentioned challenges resulting for example from communication by messaging, heterogeneous data or technology, globally distributed clients and the diversity of client requirements. The defined patterns are classified in the literature in three different ways. One possibility is to classify the patterns by thematic categories, where each category tries to answer a subset of design issue questions. The categorization is presented in the following [ZSL+22]:

<sup>1</sup><https://www.Microservice-api-patterns.org/>

**Foundation Patterns**

What components and system types are integrated? From where is an API accessible?

**Responsibility Patterns**

What is the architectural role of an API endpoint? What are the responsibilities of their operation?

**Structure Patterns**

How many representation elements for request and response messages are appropriate?

**Quality Patterns**

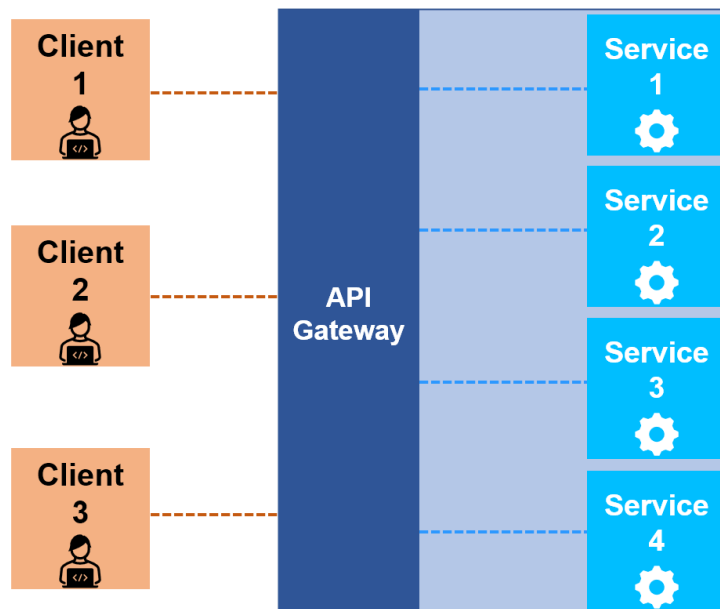
How can an API provider achieve a certain level of quality? How can the trade-offs be accounted for?

**Evolution Patterns**

How can backward compatibility be supported? How can life-cycle management concerns be handled?

The structure of the MAP descriptions are resembling the structure defined by the GoF that was described in Section 2.1. Many of these pattern descriptions include a graphical sketch, an example response and a request message in context of the pattern. Additionally, a *Discussion* section provides arguments for and against the use of the pattern. Like with design patterns in general, a user can choose between several options of MAPs to find a solution template for the encountered problem. Hence, in the *Related Patterns* section there are alternative patterns that can be used, including the respective drawbacks [ZSL+22]. The patterns focus on different areas and target different quality attributes. In the following Section 2.4, there are more details about software quality and software quality attributes.

A second source for a pattern language for the Microservices space is presented by Richardson



**Figure 2.4:** Three clients communicating via an *API Gateway* which is connected with four services (API Gateway pattern: Richardson [Ric18])

at [Microservices.io](https://www.Microservices.io/patterns)<sup>2</sup> and more detailed in his book [Ric18]. The design patterns are classified into three groups: *application patterns*, *application infrastructure patterns* and *infrastructure patterns*. The structure of the documentation is similar to the structures already presented before. For some patterns, possible variations are shown that might fit better into a use-case. In Figure 2.4, a sketch of a Microservices environment, using the *API Gateway Pattern* is shown. Instead of direct communication between each of the clients and the respective services, the clients instead communicate with the API Gateway, which is then responsible for the routing, monitoring, balancing and authorization of all requests [Ric18].

In Section 3.6.3, the chosen patterns for the conducted controlled experiment are presented thoroughly, including a pattern description and a short reasoning why these patterns were chosen as being adequate for this research on the impact on software quality.

## 2.4 Software Quality

From the very beginning of a basic computer science course, the importance of quality for software is being emphasized regularly. High software quality is so substantial because a lot of financial capital can be saved [Kra22], [Mat18]. The reasons for this fact are manifold. The time and thus the cost for bug fixing and additional maintenance steps is increased if the software quality of a system is neglected [Mat18]. Additionally, software where the quality is considered has in general a higher understandability and can thus also be adapted or tested quicker than low quality software [Kra22]. Also, there is a direct correlation between poor software quality and business damage, in form of performance problems, service downtime or security issues [EB18]. In addition to the loss of money, poor software quality can also be crucial in regards to the safety for humans in a safety-critical system [MP09]. To minimize the casualties and increase the minimum level for software quality in such systems, standards like *ISO 26262* were introduced to define a standard for functional safety in the area of road vehicles with electronic or electrical systems [Int11a].

While the consequences of poor quality is often noticeable in one way or another, the definition for software quality and how to measure it accurately is not trivial or uniform. *ISO/IEC 25010* for software engineering defines the *product quality model*, where the quality of a software system is a composition of the presence of eight main characteristics: *compatibility*, *maintainability*, *functional suitability*, *usability*, *performance efficiency*, *security*, *reliability* and *portability* [Int11b].

*ISO 9126* which was superseded in 2001 by *ISO/IEC 25010*, provides the following definition for software quality:

The quality of a system is defined as a set of attributes of a software product by which its quality is described and evaluated [Int01].

The quality of a system can be represented by the aggregation of such called **quality attributes**, which are similar to the main characteristics of the *ISO/IEC 25010* standard. Every quality attribute can be measured and evaluated, in the best case objectively, with the aid of **quality metrics**. These metrics are similar to the sub characteristics of the *ISO/IEC 25010* standard. Every software attribute can be depicted by several quality metrics which act as tools for the parameterization of the quality attribute. One can argue that some quality attributes represent a subset of another

---

<sup>2</sup><https://www.Microservices.io/patterns>

quality attribute. For example, *modifiability* and *supportability* can be seen as a subset of the attribute *maintainability*. Since focusing on all quality attributes with the same force is not feasible in practice, trade-offs have to be accepted. Hence, an important task for software engineers is to decide on primary attributes and accept certain drawbacks in other areas [BPS+12]. For example, a higher modifiability can decrease the security, while a strong focus on performance can make it hard to also achieve a satisfactory level in terms of supportability, without additional investment of money and time.

The focus for this controlled experiment will lie on the two quality attributes *understandability* and *usability*. The definitions are as follows:

### **Understandability**

The understandability of a software entity describes how much effort has to be invested by a software professional to fully comprehend the functionality of it. The higher the understandability the less effort is needed by a professional to adapt or test the software entity [BBK+78]. Understandability can refer to different levels of a software system. Examples are the source code, components, data, documentation or the overall structure of the system [LW06].

### **Usability**

The usability describes the possibility for a user of a software system, to perform the desired task effectively, efficiently, safely and with satisfaction [Int20]. Since the environment and the user are important for determining these properties, it is impossible to determine the usability of a software system in total isolation [Avo01]. Usability is a frequent topic in the human-computer interaction (HCI) field and usually used in the context of end users that interact with the software, for example via a *GUI* [Bad02]. However, for this research the focus lies on the IT professionals that act as participants in our experiment and for example interact with an API endpoint with the support of HTTP requests.

## **2.5 Related Work**

The previous sections of this work introduced some of the basic concepts that form the basis for the controlled experiment. In this section, the work is presented which is related to this research in terms of similar goals, for example the examination of the impact on software quality by design patterns, or comparable methodological methods like the execution of a controlled experiment. Some of the literature is part of the motivation for this research, while the remaining research was conducted in areas that are closely related to the influence of design patterns on software quality.

Akbulut et al. [AP19] carried out an analysis on the system performance impact of various Microservice patterns. One of the examined patterns is the *API Gateway Pattern* that is also part of the conducted experiment of our work. The test environment consisted of three main components. These created user requests which were afterwards processed within the Microservice-based test environment. Afterwards, the time was measured that the application needed to process the requests under full load. The research focuses on the comparison between the examined Microservice design patterns and uses them to reinforce the hypothesis that the Microservice architecture provides a qualitative environment, in regards to performance. The results of this study show that there is no single examined pattern that performed significantly better than the rest. The authors highlight that use-case determines which pattern is most suitable.



Pfaff [Pfa21] examined the effect of design rules for Representational State Transfer (REST) APIs, on their perceived understandability. Similar to our research, a controlled experiment via an online survey with over 100 participants was conducted. Their assignment was to solve comprehension tasks for different REST API snippets and rate them by their difficulty. The results display that in all but one case, the tasks that implemented design rules, performed significantly better than the tasks that did not incorporate them. Additionally, for the same tasks the version that included the design rules, were seen as significantly easier than the tasks that did not adhere to the rules. The researcher could not confirm a significant influence on the understandability, by the type of profession or the years of experience of the participants. The author recommends in general, to actively comply to the improving design rules when constructing REST APIs.

Bogner et al. [BWZ19] examined four Service-based design patterns in regards to their impact on evolvability. The motivation behind the focus on evolvability is that it is an important quality attribute in fast changing environments, with frequently adapting requirements for the software system. The authors address the lack of empirical evidence on the impact by design patterns and use this fact as a core motivation for their research. A controlled experiment with almost 70 Bachelor students was conducted. The task for the participants was to extend a web application. A control and a treatment group were formed. The treatment group got presented with the application using the design patterns, while the control group got a version presented that has these patterns not incorporated. Although the mean efficiency was 12% higher in the treatment group, only for one 1 of 3 tasks a significant statistical support in favor of the use of design patterns could be observed. Additionally to the controlled experiment, both web systems were compared in regards to several maintainability metrics. While some metrics had better values for the system that used patterns, for other metrics the contrary was the case. Some metrics, like cohesion and complexity, were not impacted at all. The conclusion states that there is no clear empirical evidence about improved evolvability, by the incorporation of the examined Service-based design patterns.

Khomh et al. [KG08] provide evidence that software design patterns can impact certain quality attributes negatively and thus potentially decrease the overall quality of a software system. With the reasoning that the development of a software system is mainly a *manual task*, a questionnaire with software engineers was conducted. The participants of the questionnaire should evaluate different design patterns with regards to the impact on ten quality attributes that were chosen by the authors. After aggregating all responses for each quality attribute into the groups *positive*, *neutral* and *negative*, the results indicate a diversified perception between the examined design patterns. While for most of the examined patterns the perceived quality attributes were rated as positive or neutral, a few patterns were indeed seen as a cause for reducing the quality of the system. As an example, a large number of participants think that some of the patterns significantly reduce the understandability of a system. The research is concluded with the comment that software design patterns indeed do not always increase the quality of a system and thus patterns should be used cautiously. The benefits that some patterns provide have to be weighted against the general reduction in simplicity.

Wedyan et al. [WA20] executed a systematic literature review on 50 primary studies that were published between 2000 and 2018. They examined the impact of software design patterns on the quality of software. The authors state that many existing studies have different objectives, quality metrics or attributes and thus lead to opposed results or interpretations. One main finding of this research is that the examined controlled experiments contain severe differences in their general

designs. Furthermore, it was shown that certain attributes, like the size of pattern classes or the documentation of patterns, also impact the quality of the examined software system. This is one reason why the research team calls for a standardized set of quality metrics.

Ali et al. [AE13] carried out a comparative literature survey to collect the existing evidence for software quality attribute improvement with the use of GoF software design patterns. A summary of the findings was conducted and a generalization for design patterns and the resulting change in quality was presented. The found studies were classified by quality attributes and by level of design. Only four quality attributes were specified in all of the studies: "*maintainability, fault-proneness, evolution & change proneness and performance*". The impact of design patterns on the maintainability of the software was most often examined in the studies. One of the key findings of this research is that only a handful of the GoF design patterns were actively examined in the literature, while others were not inspected at all. While in regards to the change of maintainability, 5 design patterns were examined by numerous studies in detail, 16 other patterns were never considered in this context. In regards to performance, merely 3 patterns were examined in total, by 2 out of the 17 studies from the literature survey. In conclusion, there is no agreement regarding the impact on quality by software design patterns in general. Aggregating the findings of the studies, the majority of the researchers conducted that in regards to the evolution & change proneness and maintainability, the impact from the use of design patterns was negative. While for the other two considered attributes, the results were too dissimilar or the number of studies was too low, to attempt a generalization.

Vale et al. [VCG+22] organized semi-structured interviews with nine industry experts that have at least five years of experience in software development. The goal of these interviews was to identify the software quality trade-offs that are present in practice when using design patterns for Microservices. The interview is separated into three parts. First, the participants are asked about their experience with Microservices, including technical details about previous Microservices projects. Second, selected patterns were shown to the participants. The participants were asked if they already have some experience with the respective pattern. If the answer was negative, the interview continued with the next pattern. If the answer was positive, open questions about the motivation for using the pattern were asked. Questions about the measurement and importance of software quality attributes were also inquired in a third step. The results show that on average the experts had experience with 9 out of 14 patterns. While the pattern documentations already included some of the problems and trade-offs, new drawbacks for some of the patterns were derived from the interviews. Regarding the trade-offs reported by the interviewees, 3 out of the 14 examined patterns displayed more negative than positive characteristics, 2 showed an equal amount of positive and negative points and 9 patterns have significantly more gains than drawbacks. Overall, the majority of patterns can increase the software quality, especially if used correctly. Tracing software quality is a goal for the experts, but many do not actively use quality attributes to determine the overall quality of a system.

In summary, there is a decent amount of conducted research about the impact of design patterns on software quality. Most scientists focused on the impact on only one or two quality attributes, while others examined the impact on a greater scope. Although many studies came to the conclusion that the use of design patterns is beneficial, there is enough evidence that design patterns are not guaranteed to improve the quality of a system in all circumstances. Various investigators suggest that the benefits have to be weighed against possible drawbacks, in all situations where design patterns could be used. On the contrary, the research on design patterns for Microservices and their

impact of software quality is quite rare. While Akbulut et al. [AP19] examined the impact of some Microservice patterns on the performance, there seems to be almost no research focusing on the impact of Microservice patterns on other quality attributes, yet. Additionally, there seems to be little to no research on the impact of Microservice **API** Patterns. This controlled experiment can thus potentially deliver interesting insights of design patterns for this environment.



## 3 Experiment Planning

The following chapter will focus on the methodological details. This includes the two formulated research questions, the definition of the dependent and independent variables and the hypotheses of this research. Afterwards, some information about the subjects and the requirements for participation will be shared. The examined patterns and their respective tasks will be presented next. Finally, the data analysis section will describe in-depth the steps that were taken, to extract, clean, normalize and statistically test the gathered data.

### 3.1 Research Questions

Two main research questions (RQs) were investigated in this controlled experiment:

**RQ1:** Which Microservice API patterns have a significant impact on software quality?

This primary RQ represents the main hypothesis of this research, and claims that the usage of the examined Microservice API patterns increases software quality in general, mainly by improving the monitored *understandability* or *usability* of the test objects. More details on the software quality attributes can be found in Chapter 3.6.3. The patterns differ in complexity, functionality and in the manner in which they can be integrated into a system. This is why it is likely that different patterns show varying sizes of effect on software quality. While the active usage of certain patterns might increase the correctness for the tasks of the controlled experiment, other patterns do not necessarily increase the correctness but can lead to a faster understanding for a user and thus reduce the needed time to comprehend and work with the provided software object.

**RQ2:** How does the demographic background of participants influence the effectiveness of selected patterns?

This secondary RQ investigates an exploratory part about the examined data based on the different demographic backgrounds. The participants were asked how many years of experience in IT they have gained, if they come from an academic or industrial background and what their main professional role is. Finally, a self-evaluation was conducted where the participants should rate their experience with HTTP APIs, software design patterns and Microservice API patterns, on a scale of 1 to 5. One hypothesis might be that students have in general a better understanding of theoretical questions, because not much time has passed since they learned about these topics in their studies. On the contrary, a lot of experience as a software engineer or developer could decrease the needed time to understand and scan through a provided HTTP request within a task. This can lead to a higher probability of answering tasks correctly, in comparison to students or graduates with no or very little practical experience.

## 3.2 Dependent and Independent Variables

The **dependent variable** in this controlled experiment is an aggregation of two measures: the first of the two measures is the *correctness* that can have a binary value of  $1$ , thus the recorded response being correct, or  $0$ , the response being incorrect. The second measure is the *time* in seconds that was needed to understand a scenario and finally solve the related task. This aggregation is named *Timed Actual Understandability (TAU)* and was adapted from Scalabrino et al. [SBV+21].

For this experiment the calculation of TAU looks as follows:

$$TAU = correctness_{p,t,v} \times \left(1 - \frac{time_{p,t,v}}{\max(time_t)}\right)$$

For a given participant  $p$  and a task  $t$  with one of the two versions  $v$ , either the *Pattern version P* or *Non-Pattern version N*, the *correctness* is either  $1$  if the task was answered correctly, or  $0$  otherwise. The result is then being multiplied with the inverted relation of the time that was needed to solve the specific task version to the longest time for a given task, including the P and N versions for this task, over all participants.

If a participant answered incorrectly, the value of TAU becomes  $0$ , no matter how fast the response was provided. The closer TAU is to the value of  $1$ , the higher is the recorded understandability in general.

The **independent variables** in this experiment are the *task versions* that were used in the TAU equation and the *demographic attributes* of the participants. The demographic attributes that were monitored for this experiment are the *years of experience in the IT area*, the *main IT background*, the *main professional role* and the *experience levels* in the areas of HTTP APIs, software design patterns and Microservice API patterns.

Regarding the task versions, each of the provided tasks consists of the two task versions P and N. While the P version of a task uses one of the selected Microservice API patterns, the N version of this task does not use the pattern. Each participant was randomly placed into one of two groups at the start of the experiment. Depending on the group, one of two survey versions was presented to the participant. This *crossover design* [VAJ16] results in the fact that all participants see exactly one Pattern version and one Non-Pattern version for each of the six patterns.

## 3.3 Hypotheses

The expectation in regards to RQ1 is that for most of the 12 tasks, the calculated Timed Actual Understandability (TAU) is significantly higher for the Pattern task version P than the TAU for the Non-Pattern task version N. For RQ2 there are no strict hypotheses, since it is of an exploratory type. Using the just defined dependent and independent variables in Chapter 3.2 as a support for the definition of the research hypotheses, the following is defined for RQ1:

### **Null Hypothesis ( $H_0$ ):**

The task versions using Microservice API patterns P result in an overall lower or equal software quality than the opposing task versions not using Microservice API patterns N.

**Alternative Hypothesis ( $H_1$ ):**

The task versions using Microservice API patterns P result in an overall higher software quality than the opposing task versions not using Microservice API patterns N.

While only these two hypotheses are listed for RQ1 explicitly, it has to be noted that for the statistical tests the 2 tasks per pattern are combined. Therefore 6 statistical tests will be conducted in total, to clearly identify which patterns have a significant impact on software quality.

### 3.4 Material

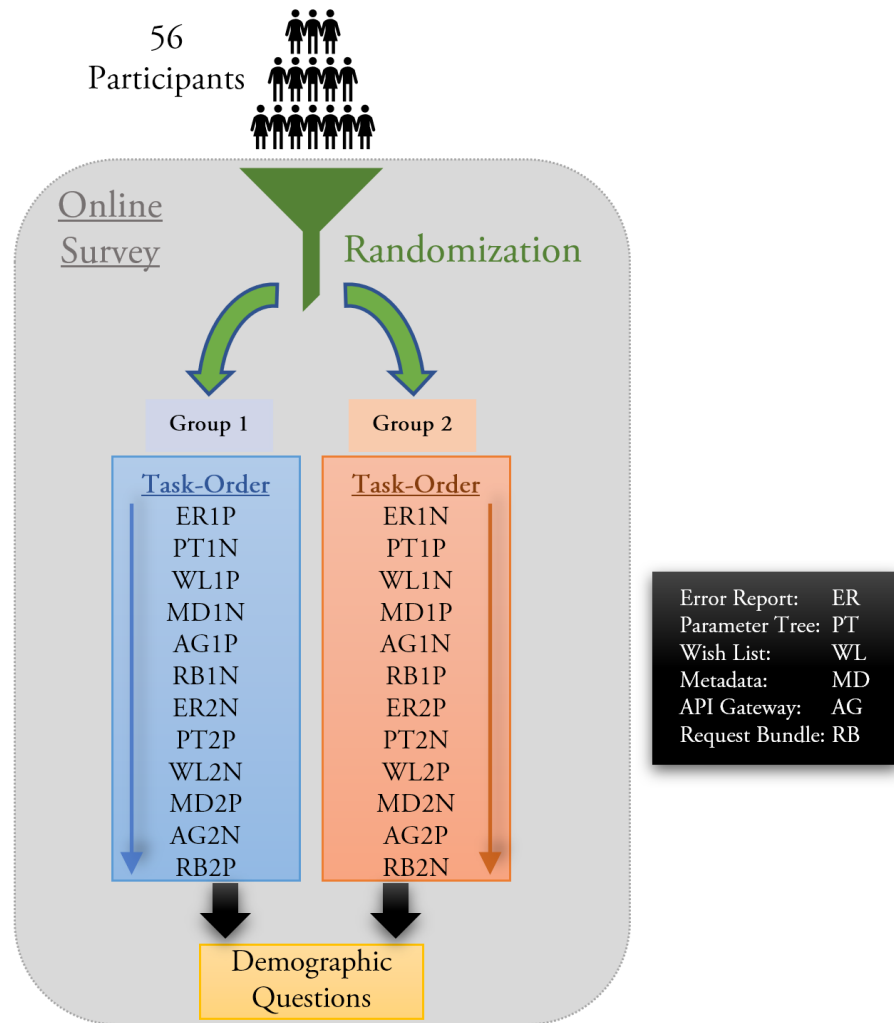
To increase the potential number of found participants, an online survey was chosen. The online survey tool *LimeSurvey*<sup>1</sup> was used, since it is easy to create new surveys, configure them, customize the structure and the content of tasks, measure the time that was needed per question and assign the participants randomly to different groups. Additionally, in contrast to some free alternatives, the settings for storing of identifiable data from the participants is variable. Throughout the whole duration of the survey the participants only used this survey tool, accessing it through a web browser of choice, without using any additional material.

### 3.5 Participants and Requirements

The goal was to attract many potential participants but also restrict the participation, so that the obtained data and results are representative of the true effects. After some reflection and discussions with colleagues, the requirements for participants were defined as follows:

- having the ability to understand English
- being enrolled as a M.Sc. or Ph.D. student in a computer science related study program  
or working as a software or IT professional in the industry or in academia

While the online survey was active, the link together with a summary of basic information about the motivation and data privacy was shared with students via mailing lists of the university. Furthermore, the online survey was shared over several social media services, like LinkedIn<sup>2</sup> or Twitter<sup>3</sup>. It was also sent directly to eligible friends and coworkers and shared inside populated Microservices and IT-focused Microsoft Teams Channels<sup>4</sup>. In addition, the donation of one euro for each of the first 200 fully completed surveys to a charity of the authors choice, was advertised to boost the motivation.



**Figure 3.1:** Overview of the crossover design, with the order of tasks for both groups

### 3.6 Experiment Specifications

This section goes into detail of the structure and content of the controlled experiment. Especially the chosen design patterns, as well as the structure and content of the formulated tasks will be shown in detail. In Figure 3.1, an overview of the seeding for the experiment can be seen. Each participant will be randomly put into one of two groups. Depending on the group, a different set of questions will be presented to that individual. Since the controlled experiment has a crossover design, for every selected pattern, each participant gets to see exactly one Pattern version P of a task, where the pattern is used, and one Non-Pattern version N of the other task for the respective pattern, where the

<sup>1</sup><https://www.LimeSurvey.org>

<sup>2</sup><https://www.Linkedin.com>

<sup>3</sup><https://www.Twitter.com>

<sup>4</sup><https://www.Microsoft.com/de-de/microsoft-teams>



pattern is not used. The order of task versions was chosen, so that several long exercises do not get shown after another. Additionally, the distance between the Pattern version P of a pattern and the Non-Pattern version N has to be maximized on average overall shown tasks. As seen in Figure 3.1, for example a participant in group 1 sees task number 1 for the *Error Report* pattern in version P on first and the inversed N version of task number 2, for the same pattern on position 7. A participant in group 2 sees the N version of the same pattern on the first position and the P version of the pattern on position 7. More details on the 24 task versions for the 6 patterns can be seen in Section 3.6.3.

### 3.6.1 Task Types

\*You want to log in to an API with the goal to retrieve an access token for it.  
As a result, you get back the following HTTP response:

```
HTTP/1.1 401 Unauthorized
Date: Wed, 20 Jan 2023 08:29:09 GMT

{
  "timestamp" : "2023-01-20T08:29:09.452+0000",
  "status" : 401,
  "error" : "Unauthorized",
  "message" : "Login Failed"
}
```

What does this response mean?

Choose one of the following answers

- The payload of the request is too large.
- The request took too long to complete.
- Your user does not have the necessary privileges for this resource.
- You used invalid authentication credentials to access the resource.
- The resource could not be found.

**Figure 3.2:** A single choice type tasks with the five answer options

The provided tasks are either comprehension questions with single choice answers or questions with a text field where the participants answer with words or sentences. For some text field tasks, an *expected syntax* is provided within the description. A big advantage of the single choice answer tasks is that these answers are easier to evaluate objectively, in comparison to multiple choice or free-text answers. Additionally, the extraction of the experiment responses is straightforward and can be automated without big challenges. Data cleaning and normalization is not needed, since only one of the five responses can be chosen without any customized input from the participant. The measured time component is in general also more meaningful when comparing single choice answers from different participants. When comparing the time needed for free-text answers, shorter answers often result in a lower response time, even though the actual time that was needed to

### 3 Experiment Planning

---

\*You are responsible for the management of articles at a grocery warehouse.

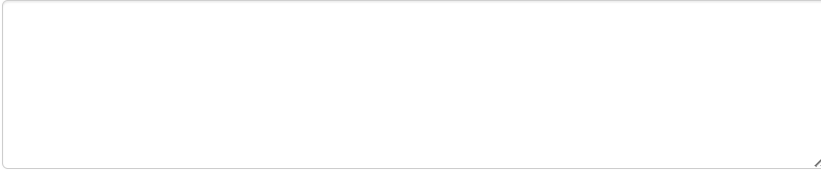
The warehouse API offers the following endpoint to delete an article with the respective item ID:

```
DELETE https://my-grocery.com/products/$id
```

Additionally, the API allows you to delete multiple items at once by using comma-separated IDs:

```
DELETE https://my-grocery.com/products/$id1,$id2,...
```

Using the given example syntax, formulate how you would delete the four articles with the IDs 12, 16, 27, and 35.



**Figure 3.3:** A free-text type task with a text field

understand the task, could be identical. Having these advantages, single choice type tasks are not always the best choice. The probability for a participant to randomly guess the correct answer or get an implicit hint by looking at the options, are two issues that are present for single choice tasks. Therefore, in this controlled experiment some tasks instead can only be answered by text input via a free-text field. For tasks where the participants have to name several objects, enter a number or have to answer with an own Hypertext Transfer Protocol (HTTP) request, free-text type tasks were chosen. The cleaning and normalization of these responses will take additional time and automation of these tasks might not be possible or restricted, as described before. An example for a single choice task can be seen in Figure 3.2. It contains some text introducing the task, a screenshot displaying an HTTP response and below the task question and the five possible answer options that are displayed to the participant in random order. Only one of these answers is correct and the interface only allows the participant to pick exactly one of these options before submitting the answer. In contrast, an example for a free-text type task can be seen in Figure 3.3. In this case, it contains some introductory text, some example syntax and below the question of the task. The participant can then enter the solution in the text field below.

#### 3.6.2 Pilot Experiment

Before the controlled experiment started, the tasks were edited and refined in many iterations. After the research group was satisfied with the overall pattern choice and the formulated tasks for the controlled experiment, a pilot experiment [TMCC10] was created that was shown to a closed group of six people with a computer science background. This group was asked to provide general feedback about the structure, visual presentation of the controlled experiment, as well as a more detailed evaluation of the content of the tasks. Most of the feedback was confirming, encouraging or included only some minor suggestions for improvement, like using a slightly different formulation for the tasks, adding or deleting some metadata in the screenshots or criticizing the color and size of a text font. However, some of the feedback from the pilot was more focused on the content and

discussed what possible conclusions could be drawn by the gathered data of the experiment. Some examples were criticized as being too special or artificial. In direct consequence, a generalization for a pattern from the chosen examples might not be feasible, or only with some disclaimers. A more detailed discussion about the generalization of the found results can be seen in Chapter 5.2.

### 3.6.3 Experiment Objects and Tasks

Picking adequate objects for the controlled experiment is important and was not trivial, since the design patterns are quite dissimilar in regards to the type of problem they are solving and in which way there are implemented. Hence, different software quality attributes are the main targets of these patterns. Since a controlled experiment with human participants was chosen as method of gathering the data, only a subset of these quality attributes is relevant for this thesis. The attributes that can be measured or experienced directly by the developer or end-user are relevant. Consequently, the main focus for this research lies on the attributes **understandability** and **usability**. The definitions of these attributes and general information about software quality can be found in Section 2.4. After a detailed examination of all Microservice API patterns in regards on the impact of quality attributes and on whether feasible examples for a survey could be formulated, the following design patterns were chosen for the controlled experiment from the main sources [ZSL+22], [Ric18]:

#### Error Report Pattern

**Problem:** How can an API provider inform its clients about communication and processing faults? How can this information be made independent of the underlying communication technologies and platforms?

**Solution:** Reply with error codes in response messages that indicate and classify the faults in a simple, machine-readable way. Additionally, add textual descriptions of the errors for the API client stakeholders, including developers and/or end users such as administrators.

**Task:** You want to log in to an API with the goal to retrieve an access token for it. As a result, you get back the following HTTP response:

```
HTTP/1.1 401 Unauthorized
Date: Wed, 20 Jan 2023 08:29:09 GMT

{
  "timestamp" : "2023-01-20T08:29:09.452+0000",
  "status" : 401,
  "error" : "Unauthorized",
  "message" : "Login Failed"
}
```

**Figure 3.4:** Pattern V1 (ER1P)

```
HTTP/1.1 401 Unauthorized
Date: Wed, 20 Jan 2023 08:29:09 GMT

{
  "timestamp" : "2023-01-20T08:29:09.452+0000",
  "status" : 401
}
```

**Figure 3.5:** Non-Pattern V1 (ER1N)

**Question:** What does this response mean?

- The resource could not be found.
- You used invalid authentication credentials to access the resource. **(Correct)**
- The payload of the request is too large.
- The request took too long to complete.
- Your user does not have the necessary privileges for this resource.

### Parameter Tree Pattern

**Problem:** How can containment relationships be expressed when defining complex representation elements and exchanging such related elements at runtime?

**Solution:** Define a Parameter Tree as a hierarchical structure with a dedicated root node that has one or more child nodes.

**Task:** You use a cinema API to retrieve all movies that will be screened in your hometown tomorrow. The response looks as follows:

```
{
  "cinema": {
    "date": "20.10.2022",
    "filtering": [],
    "published": true,
    "movies": [
      {
        "name": "Dune",
        "minAge": "12",
        "length": "155 minutes",
        "genre": "science fiction",
        "time": [
          "1:50pm",
          "4:30pm",
          "8:45pm"
        ],
        "reservation": "28/70"
      },
      {
        "name": "Pathaan",
        "minAge": "16",
        "length": "146 minutes",
        "genre": "thriller",
        "time": [
          "11:30am",
          "3:30pm"
        ],
        "reservation": "43/70"
      },
      {
        "name": "Avatar: The Way of Water",
        "minAge": "12",
        "length": "193 minutes",
        "genre": "science fiction",
        "time": [
          "11:00am",
          "3:30pm",
          "7:50pm"
        ],
        "reservation": "167/200"
      },
      {
        "name": "Alice, Darling",
        "minAge": "12",
        "length": "90 minutes",
        "genre": "thriller",
        "time": [
          "1:15pm",
          "3:40pm"
        ],
        "reservation": "13/70"
      }
    ]
  }
}
```

Figure 3.6: Pattern V2 (PT2P)

```
{
  "date": "20.10.2022",
  "filtering": [],
  "published": true,
  "movie1_name": "Dune",
  "movie1_minAge": "12",
  "movie1_length": "155 minutes",
  "movie1_genre": "science fiction",
  "movie1_time": [
    "1:50pm",
    "4:30pm",
    "8:45pm"
  ],
  "movie1_reservation": "28/70",
  "movie2_name": "Pathaan",
  "movie2_minAge": "16",
  "movie2_length": "146 minutes",
  "movie2_genre": "thriller",
  "movie2_time": [
    "11:30am",
    "3:30pm"
  ],
  "movie2_reservation": "43/70",
  "movie3_name": "Avatar: The Way of Water",
  "movie3_minAge": "12",
  "movie3_length": "193 minutes",
  "movie3_genre": "science fiction",
  "movie3_time": [
    "11:00am",
    "3:30pm",
    "7:50pm"
  ],
  "movie3_reservation": "167/200",
  "movie4_name": "Alice, Darling",
  "movie4_minAge": "12",
  "movie4_length": "90 minutes",
  "movie4_genre": "thriller",
  "movie4_time": [
    "1:15pm",
    "3:40pm"
  ],
  "movie4_reservation": "13/70"
}
```

Figure 3.7: Non-Pattern V2 (PT2N)

**Question:** How many movies of the category "thriller" can you choose from? (Free-text response)

Two / 2 (Correct)

### Metadata Element Pattern

**Problem:** How can messages be enriched with additional information so that receivers can interpret the message content correctly, without having to hardcode assumptions about the data semantics?

**Solution:** Introduce one or more Metadata Elements to explain and enhance the other representation elements that appear in request and response messages. Populate the values of the Metadata Elements thoroughly and consistently. Process them as to steer interoperable and efficient message consumption and processing.

**Task:** You use the search resource of the API of a shopping platform with the query "Coffee". As a result, you get the following response:

```
{
  "search_parameters": {
    "_nk": "Coffee",
    "_sortOptions": [
      "shippingCost",
      "asc"
    ],
    "_ipg": "25"
  },
  "search_information": {
    "organic_results_state": "Results for exact spelling",
    "total_results": 580000,
    "query_displayed": "Coffee"
  },
  "organic_results": [
    {
      "title": "Premium Coffee Kaffee TO GO Pappbecher 200ml Tee",
      "condition": "Brandneu",
      "rating": "5/5",
      "price": {
        "raw": "EUR 31,49"
      },
      "shippingCost": "EUR 0,00"
    },
    {
      "title": "T24 Premium Coffee Getränke TO GO Pappbecher 200ml Kaffeebecher Coffeebecher",
      "condition": "Brandneu",
      "rating": "4/5",
      "price": {
        "raw": "EUR 12,49"
      },
      "shippingCost": "EUR 2,50"
    },
    {
      "title": "Edelstahl Thermobecher Coffee to go Becher Isolierbecher Kaffeebecher 500 ml",
      "condition": "Brandneu",
      "rating": "5/5",
      "price": {
        "raw": "EUR 15,99"
      },
      "shippingCost": "EUR 6,50"
    }
  ]
}
```

**Figure 3.8:** Pattern V2 (MD2P)

```
{
  "query_displayed": "Coffee",
  "organic_results": [
    {
      "title": "Premium Coffee Kaffee TO GO Pappbecher 200ml Tee",
      "condition": "Brandneu",
      "rating": "5/5",
      "price": {
        "raw": "EUR 31,49"
      },
      "shippingCost": "EUR 0,00"
    },
    {
      "title": "T24 Premium Coffee Getranke TO GO Pappbecher 200ml Kaffeebecher Coffeebecher",
      "condition": "Brandneu",
      "rating": "4/5",
      "price": {
        "raw": "EUR 12,49"
      },
      "shippingCost": "EUR 2,50"
    },
    {
      "title": "Edelstahl Thermobecher Coffee to go Becher Isolierbecher Kaffeebecher 500 ml",
      "condition": "Brandneu",
      "rating": "5/5",
      "price": {
        "raw": "EUR 15,99"
      },
      "shippingCost": "EUR 6,50"
    }
  ]
}
```

**Figure 3.9:** Non–Pattern V2 (MD2N)

**Question:** How did the API sort the retrieved products?

- The products are sorted in ascending order regarding the rating.
- The products are sorted in descending order regarding the rating.
- The products are sorted in descending order regarding the title.
- The products are sorted in descending order regarding the base price.
- The products are sorted in ascending order regarding the shipping costs. **(Correct)**

#### Wish List Pattern

**Problem:** How can an API client inform the API provider at runtime about the data it is interested in?

**Solution:** As an API client, provide a Wish List in the request that enumerates all desired data elements of the requested resource. As an API provider, deliver only those data elements in the response message that are enumerated in the Wish List.

**Task:** You are working at a university and frequently need to access student information via an API. An example student object looks like this:

```
{
  "studentName": "Peter",
  "studentID": "28741993",
  "enrolled": true,
  "registeredExams": [],
  "birthday": "1989-12-31",
  "address": "University Street 13",
  "postalCode": "7640"
}
```

**Figure 3.10:** Pattern and Non-Pattern V1 (WL1P/WL1N)

For a follow-up API request, a student object is required with the following attributes only: `studentName`, `birthday`, and `postalCode`.

*Additional hint for Pattern Task version:*

You can retrieve a customized student object with this resource and syntax:

```
const student = GET https://my-university.org:8080/students/$id?fields=
attribute1,attribute2,...
```

*Additional hint for Non-Pattern Task version:*

You can retrieve and refine a student object with this resource and syntax:

```
const response = GET https://my-university.org:8080/students/$id
const student = {
  attribute1: response.attribute1,
  ...
}
```

**Question:** Using the described syntax, formulate the necessary request and potential follow-up actions to retrieve the described object for the student with identifier (ID) '20'. (Free-text response)

**Correct– Pattern Version:**

```
const student = GET https://my-university.org:8080/students/20fields=
studentName,birthday,postalCode
```

**Correct– Non-Pattern Version:**

```
const response = GET https://my-university.org:8080/students/20
const student = {
  studentName: response.studentName,
  birthday: response.birthday,
  postalCode: response.postalCode
}
```

## API Gateway Pattern

**Problem:** How do the clients of a Microservice-based application access the individual services?

**Solution:** Implement an API Gateway that is the single entry point for all clients. It handles requests in one of two ways. Some requests are simply routed to the appropriate service. Other requests are sent out to multiple services.

**Task:**

- *Pattern Version:* In a distributed system, we have three clients that communicate with an API Gateway. This API Gateway is connected to four services, providing functionality. All three clients are using each of the four services.

- *Non-Pattern Version:* In a distributed system, we have three clients that communicate with four services, providing functionality. All three clients are using each of the four services.

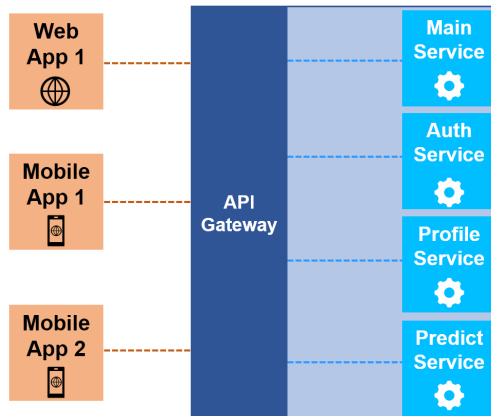


Figure 3.11: Pattern V2 (AG2P)

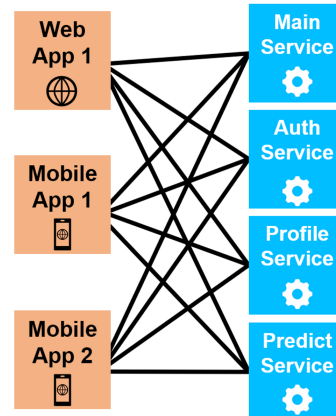


Figure 3.12: Non-Pattern V2 (AG2N)

**Question:** Due to a company site consolidation, *Main Service* is moved to another machine in a new data center, which changes its Internet Protocol (IP) address and endpoint. Name all components from the diagram that have to be changed as a result of this migration. (Free-text response)

API Gateway (**Correct– Pattern Version**)

Web App 1, Mobile App 1, Mobile App 2 (**Correct– Non-Pattern Version**)

### Request Bundle

**Problem:** How can the number of requests and responses be reduced to increase communication efficiency?

**Solution:** Define a Request Bundle as a data container that assembles multiple independent requests in a single request message. Add metadata such as identifiers of individual requests and bundle element counter.

**Task:** You are responsible for the customer management at a retail store. To obtain information about customers, you communicate directly with an API that provides customer data from a database. The API offers the following resource to retrieve the details of a customer by ID:

```
GET https://my-retail-store.com/customers/$id
```

*Additional hint for Pattern Task version:* Additionally, it allows you to fetch multiple customers at once by using comma-separated IDs:

```
GET https://my-retail-store.com/customers/$id1,$id2,...
```



**Question:** Describe how you would request the information of the four customers with the IDs '5', '9', '13', and '24'. (Free-text response)

**Correct– Pattern Version:**

```
GET https://my-retail-store.com/customers/5,9,13,24
```

**Correct– Non-Pattern Version:**

```
GET https://my-retail-store.com/customers/5
GET https://my-retail-store.com/customers/9
GET https://my-retail-store.com/customers/13
GET https://my-retail-store.com/customers/24
```

## 3.7 Data Analysis

In total, 98 participants took part in the online survey, from which 66 participants completed it. The responses of these 66 people were used for further data extraction and analysis. The responses of the remaining 32 people were discarded, because incomplete surveys are problematic for the analysis and interpretation of results. The majority of discarded responses belong to participants that stopped the survey within the first two questions. Reasons might be the discovery of the size or complexity of the tasks or simple curiosity that tempted users to start the survey, to just view the first one or two tasks. After the two-week period of response collection for the experiment has ended, the data was afterwards exported to a comma-separated values (CSV) file, via the built-in function of LimeSurvey. Simple data cleaning and analysis scripts were written in Python 3.10.5<sup>5</sup> for the respective tasks. First, the written program iterated through all of the recorded timings of the participants and marked all of the responses as *invalid*, for which a person needed **more than 400 seconds** or **less than 10 seconds** for the submission of the answer. For this controlled experiment, the responses for **four tasks were marked invalid**. The responses of one participant were discarded completely, since for 3 out of the 12 presented tasks, the time per task exceeded the 400 seconds threshold. In this case, the total time of participation for the whole survey exceeded 37 minutes, while over all participants **the median time was 14,5 minutes** and **the average time 15,5 minutes**. The motivation for the full exclusion of this participant is the high probability that the survey was not conducted inside an optimal environment. Instead, it is likely that the participant took long breaks, handled different activities or looked something up on the internet that could help to solve the tasks, all while the survey was ongoing. A more detailed discussion regarding this topic can be found in Section 5.2 Threats to Validity. After this procedure, the data of **65 participants are seen as valid** and was taken into account for future steps.

The next step is the verification for the given responses. For the single-choice tasks, this was trivial since the **correct option was always encoded as a 1** and the four **incorrect options were encoded as a 0**. For the free-text tasks the script only encoded the most common, correct and incorrect answers, respectively as a *1* or *0*. The other responses for these tasks were reviewed and

---

<sup>5</sup><https://www.Python.org/>

normalized manually in the next step by the researcher. The normalization step deleted all redundant white spaces and paragraphs that were added by the participants. Obvious spelling mistakes in the responses of the participants were corrected manually, such as *Main Srvice* instead of *Main Service* or *reponse* instead of *response*. However, the checks for syntax, in the case of responses containing HTTP requests or source code, were more strict. This concludes the cleaning and normalization steps for the gathered data of the experiment.

Next, the TAU values for all responses over all tasks, were calculated. Furthermore, the mean and the median were calculated for each of the 24 data columns, 12 tasks and 2 task versions per task. With the application of the *Shapiro-Wilk test* [SW65] on the data, it was confirmed that 22 out of the 24 data groups are not normally distributed. Therefore, the decision was made to use the **Mann–Whitney U test**, which is a statistical non-parametric test for selected values of two independent groups [MW47]. To perform the tests, the TAU data of both Pattern versions P per task was concatenated. Analogously, the same was done for the Non-Pattern versions N of all tasks. As a result there are 6 statistical tests in total, one per pattern. Since there are several tests conducted for this research, the *multiple comparisons problem* should not be ignored [Mil81]. To combat this issue, the *Bonferroni correction* was an option [MJM00]. While this method reduces type 1 errors in general and is straightforward to calculate, the **Holm-Bonferroni method** was used instead [Hol79]. It is a modification of the single-step *Bonferroni correction* and provides a higher statistical power [Hol79]. The resulting corrected p-values will be displayed in Chapter 4.

To determine the dimension of the results, an appropriate effect size had to be chosen. **Cohen's d** was chosen since the analysis for this research is based on a comparison of data from two distinct groups, with a sample size of over 30 for both [Coh88]. Cohen's d was calculated with the *standard pooled deviation for two independent samples*. The interpretation of the values for *d* are taken from Cohen and Sawilowsky [Coh88], [Saw09]:

- $d \leq 0.2$  : very small effect
- $0.2 \leq d < 0.5$  : small effect
- $0.5 \leq d < 0.8$  : medium effect
- $0.8 \leq d < 1.2$  : large effect
- $1.2 \leq d < 2.0$  : very large effect
- $d \geq 2.0$  : huge effect

For the secondary research question (RQ2), first the extracted demographic data was normalized. Data bucketing was performed in some cases, for example to aggregate the years of experience or professional roles that were only stated once or twice. The demographic data of the participants was compared between the two survey groups. Since this analysis concerns an exploratory research question, the correlation between the demographic data and the values of TAU was calculated. First, *Pearson's correlation coefficient* [Pea95] and *Spearman's rank correlation coefficient* [MW02] were considered. However, having disadvantages like the strong reaction caused by outliers or assuming normality, the *Kendall rank correlation coefficient* was chosen, instead [Ken38]. To further examine possible correlations between the demographic data and values of TAU, linear regression models were constructed for the self-assessed experience categories. For this exploratory question, the significance level was set to  $\alpha = 0.5$ .

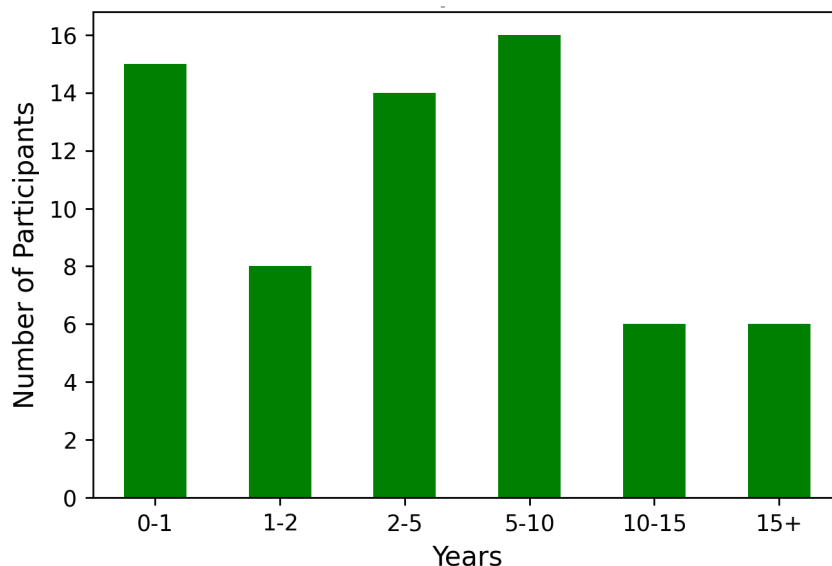
The results of the calculations, statistical tests and the visual representation with box plots and tables can be seen in Chapter 4, with an interpretation of these findings in Section 5.1.

## 4 Results

This chapter presents the findings that were collected and processed in the previous steps. Depending on the type of data, different visualization techniques are used. First, the demographic data is aggregated and displayed in a short overview. Afterwards, the focus lies on the findings of the comprehension questions of the experiment. Finally, the results regarding the influence of demographic data will be shown.

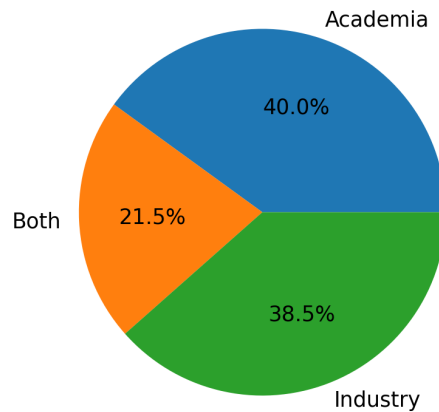
### 4.1 Participant Demographics

Due to the measures that were taken to attract participants with different professional background and years of experience, the demographics of the participants are certainly diverse. In Figure 4.1, the distribution for the participants in regards to their years of professional experience with IT or software can be viewed. The recorded experience ranged from under 1 to over 35 years. While a big portion of the participants only have a few years of experience, 25% of the participants have 5 to 10 years of experience. Furthermore, 4.6% of the participants have over 30 years of experience in this field. With a **median of 4 years** and a **mean of 6.6 years**, the group of candidates is altogether quite experienced.

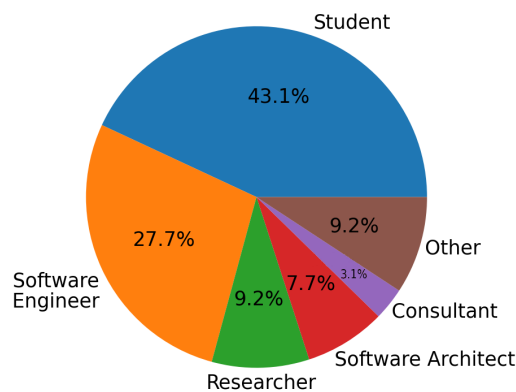


**Figure 4.1:** Distribution of participants in regards to their years of experience in IT

In Figure 4.2, the professional background of the participants is displayed. The fragment of academic and industry environments is almost equal in size (40% academia and 38.5% industry). 21.5% of candidates answered that they have a background in both of these fields. Additionally, the distribution for the experiment subjects in regards to their main role can be seen in Figure 4.3. 43% declared that their main role is being a student, 27.7% are software engineers, 9.2% researchers, 7.7% software architects and 3.1% consultants. **9.2%** are being displayed in the category **other**, where only one or two occurrences per role are present, examples are *teachers*, *managers*, and *technical supporters*.



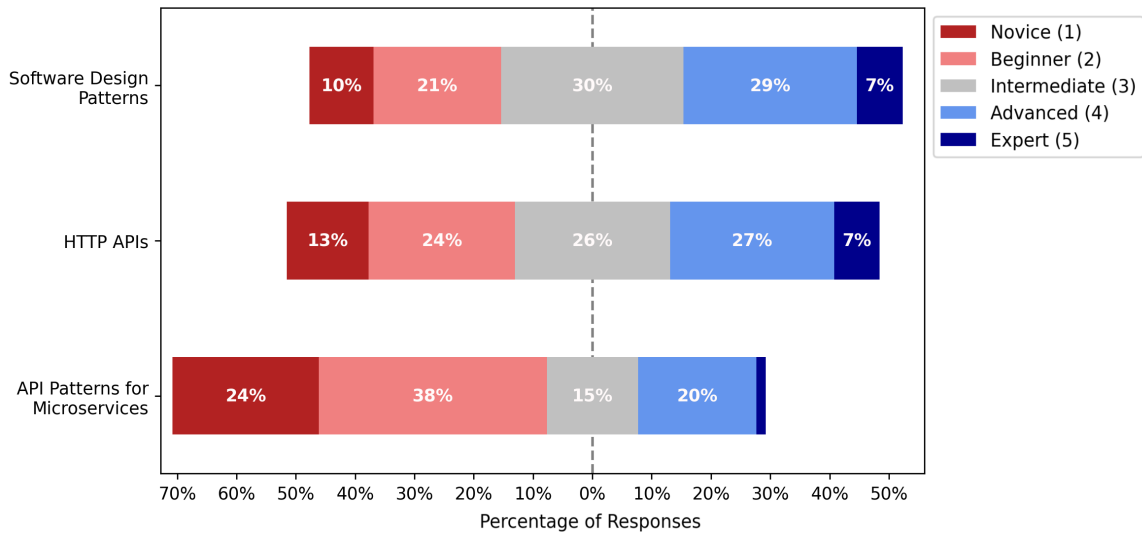
**Figure 4.2:** Pie chart with the distribution of participants in regards of their professional background



**Figure 4.3:** Pie chart with the distribution of participants in regards of their main role

At the end of the online survey, the participants were asked to perform a self-assessment on their experience with *software design patterns*, *HTTP APIs* and *MAP*. A 5-point Likert scale [Lik32] was provided which ranges from level *novice (1)*, to *expert (5)*. The results in Figure 4.4 show that 36% of participants see themselves as advanced or expert users of software design patterns. The distribution is almost identical with 34% in regards to HTTP APIs. Only a small portion thinks that they are novice users for the first two topics (10% for software design patterns and 13% for HTTP APIs). However, the assessment for the experience with Microservice API patterns shows a noticeable contrast. Only 23% of participants feel like advanced or expert users from which 3% see themselves as experts. On the contrary, 24% of all participants have stated that they feel like novice users in regards to their experience with MAP. While the recorded differences

in experience were anticipated, since the design patterns for Microservices are rather new and they also represent a subset of all software design patterns, they are still discussed further in Chapter 5.



**Figure 4.4:** 5-point Likert scale for three self-assessment topics

The comparison of demographics between the two randomized groups can be seen in Table 4.1. Even though the participants were placed in either group 1 or 2 randomly, one group ended up having 5 more people than group 2. Additionally, the years of experience in IT is higher for group 1 than group 2 (28% difference). The comparison for the self-assessment questions shows also quite some differences. The self-assessed experience with design patterns was higher for participants of group 1 (17% difference), for HTTP APIs (9% difference) and also for Microservice API patterns (30% difference). Although there are visible differences between the two groups, the differences are minor and thus should not pose a problem for the interpretation of results. Nevertheless, the threats that might appear due to these differences are discussed further in the next chapter.

Group	# Participants	Years in IT	M.E. (Patterns)	M.E. (HTTP APIs)	M.E. (MAP)
1	35	7.34	3.23	3.03	2.63
2	30	5.73	2.77	2.77	2.03

**Table 4.1:** Demographic data of the participants in regards to the two randomized groups (Mean experience: M.E.)

## 4.2 Impact of Patterns (RQ1)

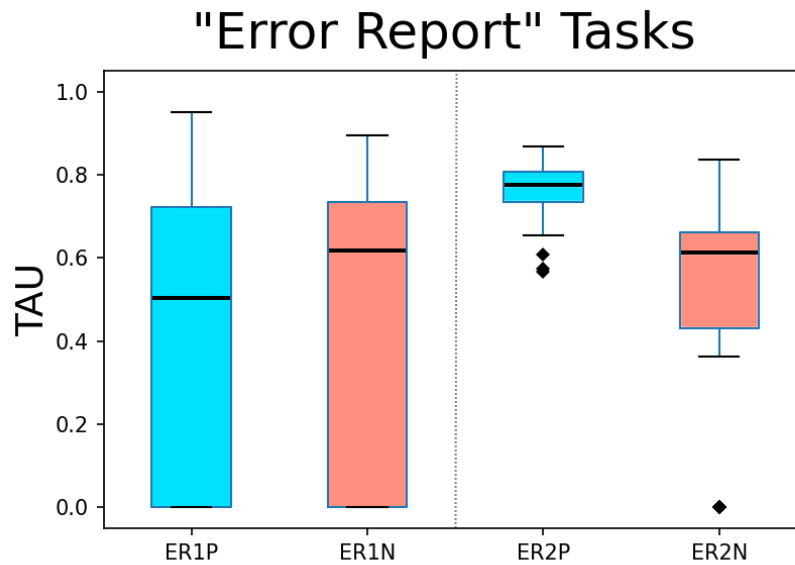
After following the extraction, cleaning, normalization and analysis steps that were presented in detail in Section 3.7, the results that are relevant to answer RQ1 are presented in the following. The main goal is to find out if the usage of Microservice API patterns truly increases the quality of the software significantly.

Task Name	Mean TAU		Correct Answers (%)		Mean Duration (s)	
	Pattern P	Non-Pattern N	Pattern P	Non-Pattern N	Pattern P	Non-Pattern N
Error Report 1	0.3804	0.4259	57.14	60.00	43.92	39.57
Error Report 2	0.7593	0.5001	100.00	82.86	41.46	67.45
Parameter T. 1	0.5104	0.0907	66.67	17.14	38.24	56.70
Parameter T. 2	0.6264	0.6724	100.00	100.00	38.62	33.87
Wish List 1	0.3767	0.3240	54.55	53.33	131.24	163.04
Wish List 2	0.5481	0.3872	70.00	60.00	80.91	119.14
Metadata 1	0.7841	0.6463	96.67	88.57	62.05	88.05
Metadata 2	0.6389	0.5572	100.00	93.33	53.44	54.07
API Gateway 1	0.7287	0.5864	91.43	72.41	84.25	76.91
API Gateway 2	0.6045	0.5844	80.00	97.14	42.83	55.94
Req. Bundle 1	0.5432	0.4154	66.67	54.29	64.90	91.56
Req. Bundle 2	0.4589	0.4265	74.29	63.33	45.45	39.25

**Table 4.2:** Several statistical values for all pattern tasks and versions (N and P)

Table 4.2 displays a detailed assortment of the mean TAU values, the proportion of correct answers (%) and the mean duration (s), for all six design patterns and the two tasks per pattern. For **10 out of the 12 tasks, the Timed Actual Understandability (TAU) was higher for the Pattern versions P** in comparison to their respective Non-Pattern versions N. The two tasks where mean values of TAU were worse for the Pattern versions are *Error Report 1 (ERIP)* and *Parameter Tree 2 (PTIP)*. The relative portion of correct answers for *ERIP* was smaller than for the Non-Pattern version (*ERIN*) (57.14% against 60%) and the mean duration to solve the task was also higher for *ERIP* than *ER2P* (43.92s against 39.57s). In contrast, *PT2P* and *PT2P* were both answered correctly by all participants from the respective seeding groups. However, the mean duration was higher for the Pattern version than for the Non-Pattern version (38.24s against 33.87s) and thus the value for TAU was better for the latter.

To go more into detail on the findings for the individual Microservice API patterns and also show potential differences between task number 1 and 2 for each of the patterns, several box plots will be presented in the following.



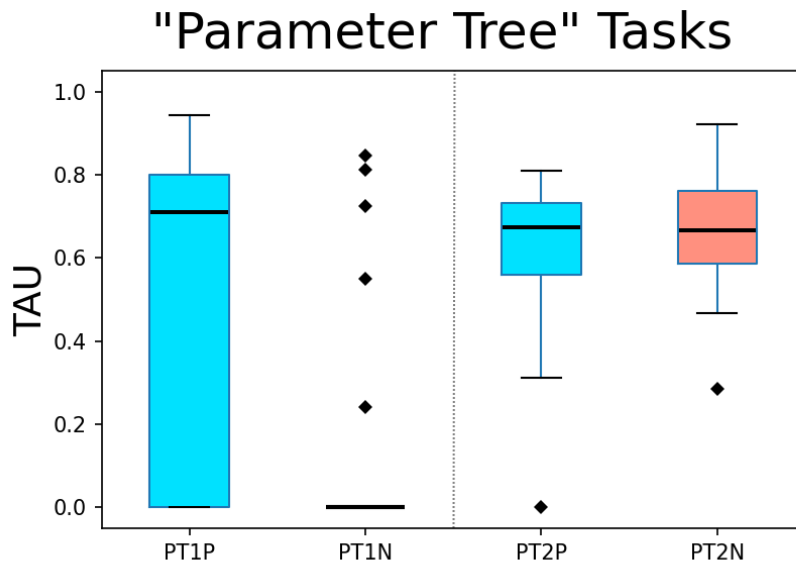
**Figure 4.5:** TAU values in contrast for the Error Report tasks

The results of the tasks for the **Error Report** pattern show that there is a difference between *ER1* and *ER2* (Figure 4.5). For the former, the distribution for TAU is similar for P and N, but the median is clearly worse in the Pattern-Version. *ER2* shows a different outcome. The median TAU value is significantly better for *ER1* than *ER2*. Additionally, the spread of TAU for *ER2* is substantially smaller.

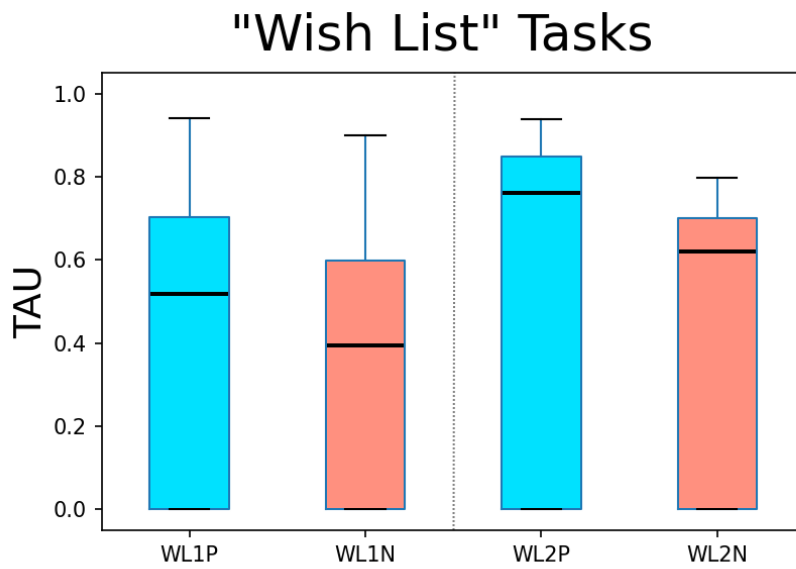
In Figure 4.6, it is shown that for the **Parameter Tree** design patterns, the Pattern version *PT1P* performed way better than the Non-Pattern version *PT1N*. *PT1N* provides the worst value for TAU over all 24 task versions in this controlled experiment. The performance was much worse than for all other task versions and might hint at an ambiguous or very complex task description. The possible reasons will be discussed in Chapter 5. *PT2P* performed just slightly better than the Non-Pattern task version *PT2N*.

For the **Wish List** tasks, the results look very good in favor of the Pattern versions *WL1P* and *WL2P* (Figure 4.7). In both cases, the median value of TAU is significantly higher for the Pattern versions than for their respective antagonists. The upper quantiles are also significantly higher for *WL1P* and *WL2P*. For *WL1P* and *WL1N* the mean duration was the longest for all task versions (131.24s and 163.04s) Over all tasks, the respective differences of TAU for this pattern between Pattern and Non-Pattern versions are most significant.

The **Metadata** tasks (Figure 4.8) display favorable results for the Pattern versions (*MD1P* and *MD2P*) in comparison to their Non-Pattern counterparts. The median TAU value is significantly higher for the Pattern versions. The mean duration and the proportion of correct answers have all better values for the respective Pattern versions, in comparison to the Non-Pattern versions.

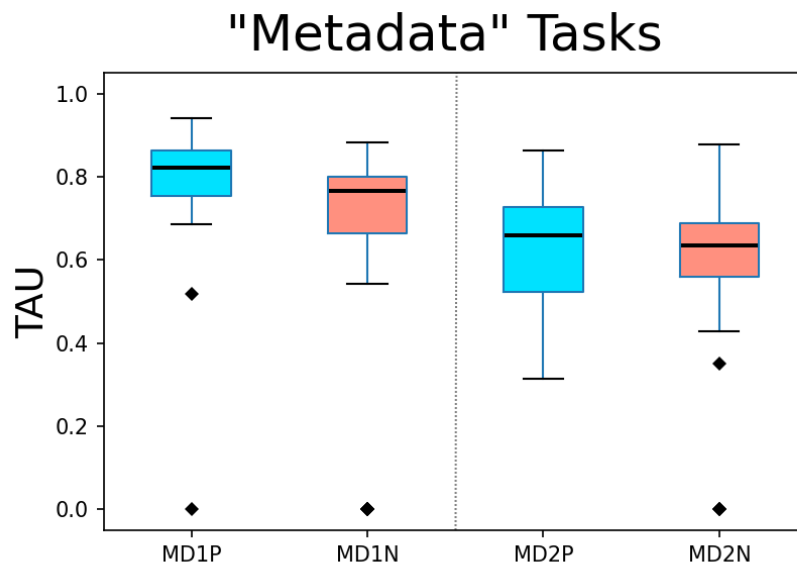


**Figure 4.6:** TAU values in contrast for the Parameter Tree tasks



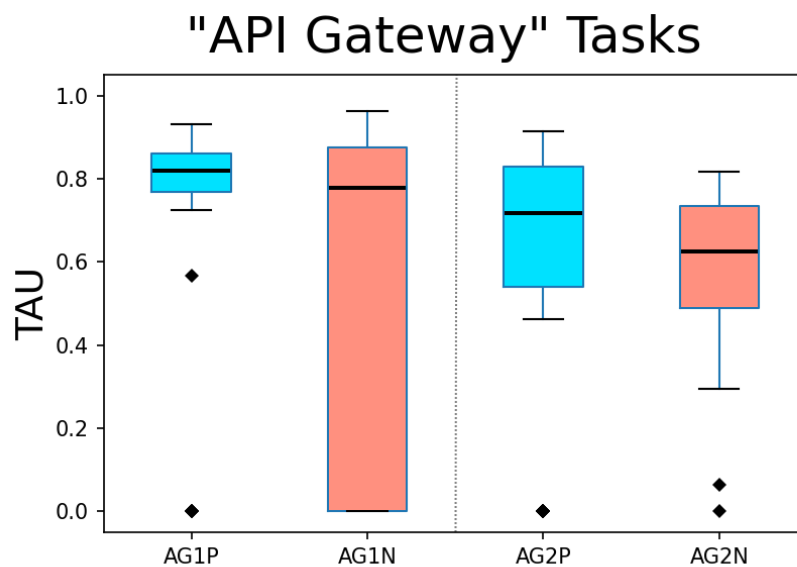
**Figure 4.7:** TAU values in contrast for the Wish List tasks





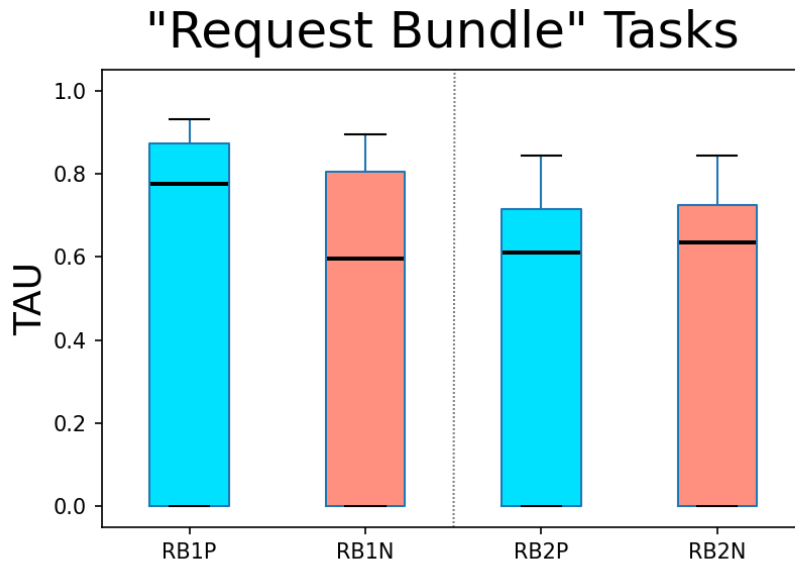
**Figure 4.8:** TAU values in contrast for the Metadata tasks

The results of the **API Gateway** (Figure 4.9) show a similar picture like the results of the previous two patterns. The median value is also significantly higher for *AG1P* and *AG2P* than in the case of *AG1N*, *AG2N*, respectively. The spread for TAU is also less extreme for the Pattern version of *AG1P*. The spread for *AG1N* is quite extreme in comparison to the other three API Gateway task versions. The possible reasons will be discussed in Chapter 5.



**Figure 4.9:** TAU values in contrast for the API Gateway tasks

Figure 4.10 exhibits the results of the **Request Bundle** pattern. While the median value of TAU is significantly higher for *RB1P* than *RB1N*, the opposite is true for the second task. TAU of *RB2P* is slightly lower than of *RB2N*. The spread is similar for both task versions, respectively. The share of correct answers is higher for the two pattern versions, *RB1P*: 66.67%, *RB2P*: 74.29% against *RB1N*: 54.29%, *RB2N*: 63.33%. The mean duration was significantly larger for *RB1N* than for the other three versions.



**Figure 4.10:** TAU values in contrast for the Request Bundle tasks

Although Table 4.2 and the box plots display some interesting results, a statistical verification of the findings is needed for the determination of significance. The significance level for the tests is  $\alpha = 0.05$ . In Table 4.3, the results of the hypothesis testing are summarized. The *pattern names*, *values for U (Mann–Whitney U test)* and *Cohen's d* can be seen. Additionally, the *original p-values from the Mann–Whitney U test* and the *corrected p-values from the Holm–Bonferroni correction* are displayed in the table. The last column describes if the *alternative hypothesis  $H_1$*  is accepted for the pattern or not.

The alternative hypothesis  $H_1$  can be accepted for 5 of 6 examined Microservice API patterns.

Pattern Name	M-Whitney U	Original p-value	Corrected p-value	Cohen's d	$H_1$ Accepted
Parameter Tree	2823.0	0.000391	0.002346	0.66	Yes
Metadata	2531.5	0.016801	0.033601	0.42	Yes
API Gateway	2655.5	0.003335	0.016674	0.29	Yes
Wish List	2528.0	0.008809	0.033590	0.28	Yes
Error Report	2621.5	0.008398	0.033590	0.27	Yes
Request Bundle	2371.0	0.109209	0.109209	-	<b>No</b>

**Table 4.3:** Results of the hypothesis tests for all examined Microservice API patterns

The **Request Bundle** pattern has a **p-value of 0.1092** and thus the null hypothesis cannot be rejected. The other 5 patterns are sorted in the table in descending order by the effect size (Cohen's  $d$ ). The **Parameter Tree** pattern showed a **medium effect** ( $0.5 \leq d < 0.8$ ). The **Metadata, API Gateway, Wish List** and **Error Report** patterns all displayed a **small effect** ( $0.2 \leq d \leq 0.5$ ).

### 4.3 Influence of Demographic Data (RQ2)

The results that are relevant to answer RQ2 are presented next. The main goal is to investigate to which extent the demographic background of participants influenced the effectiveness of selected patterns.

For this purpose, several statistical tests were conducted. The significance level for this exploratory research question is  $\alpha = 0.05$ , without a p-value correction. The statistical tests with *Kendall's rank correlation coefficient* delivered no strong statistical evidence for a correlation between the years of experience in IT of the participants and the values of TAU. The two Pattern versions P for each pattern were combined and tested, afterwards the same was done for the 6 Non-Pattern versions N. Of the 12 tests, only one for the **Wish List Non-Pattern N** version displayed a significant result, with a very weak correlation ( $\tau = 0.2044$ ,  $p = 0.0281$ ). For the remaining 11 tests, there are no significant results ( $p = 0.0578 - 0.9540$ ), while the correlation values seem quite arbitrary ( $\tau = -0.0998 - 0.1755$ ).

Additionally, a *linear regression* analysis [MPV21] was used to check if there is any correlation between one of the three self-assessed experience categories and TAU. The linear regression model was constructed separately for the concatenated sets of P task versions and N task versions. The data of the three experience questions were tested independently of each other with each of the concatenated sets. 30% of data was randomly chosen for testing, while the remaining 70% were used for training. Thus, the results can differ, when executing the test script several times. The highest *coefficient of determination* was found for the correlation between *the experience with HTTP APIs* and the **API Gateway P** version ( $R^2 = 0.2433$ ,  $p < 0.001$ ). The found correlation between the **Wish List P** version and *the experience with Microservice API patterns* displayed the second highest value ( $R^2 = 0.2107$ ,  $p < 0.001$ ). The results for all of the remaining tests have lower values with a big spread ( $R^2 = 0.0008 - 0.2032$ ,  $p < 0.001$ ). In conclusion, the constructed linear regression models could not provide reliable predictions for the values of TAU. However, for all cases the computed correlation was either non-existent or weak positive. No negative slope for the regression line was found during the tests, which means that no negative correlation has been observed. Combining the three experience values and testing them together for the 12 sets of concatenated TAU values did not change the results in any significant way. Additionally, there was no significant difference between the quality of model for P and N versions.



## 5 Discussion

This chapter provides an explanation of the collected results from the previous chapter. Additionally, a connection between the findings and the defined research questions is created. Finally, several types of validity are exhibited and the potential threats for the research findings are presented.

### 5.1 Interpretation of Results and Implications

With the results of the controlled experiment, it was shown that for 10 out of 12 prepared tasks the use of design patterns had a positive and significant impact on the software quality. 5 out of the 6 examined Microservice API patterns have significantly increased the value for Timed Actual Understandability, with only the **Request Bundle** pattern being an exception. While this pattern increases the TAU value for task 1, the impact was slightly negative for task 2. The reason might be that some experienced participants applied the pattern implicitly in the N version, even though the pattern name or the task version was never shown to them. A possible reason why the Pattern version P of task 2 has a lower value for *tau* than the Non-Pattern version N, is the trade-off between complexity in form of needed time to understand the pattern and the long-term benefit that can be gained from its usage. It has to be considered that the relative amount of correct answers was almost 10% higher for the Pattern version, but the mean duration also increased by 6 seconds, when using the version with the design pattern. This leads to the speculation that this pattern is still viable in other scenarios, since the time that is needed to fully understand a design pattern is usually only invested once. Another possible reason is the combination of the more complex syntax for P and the availability of the copy-paste function in the online survey. While it should be faster to type

```
GET https://example.org/customers/5,9,13,24
```

instead of

```
GET https://example.org/customers/5
GET https://example.org/customers/9
GET https://example.org/customers/13
GET https://example.org/customers/24
```

the bottom N version can be completed faster with copy-pasting and adapting one line. One additional argument for the claim that the P version syntax might be more complex, are the responses of the participants. A big part of incorrect answers to the Pattern version P task 2 resulted from syntax errors, like

customers/id=5,9,13,24, customers/\$5,\$9,\$13,\$24

These might not seem like big errors, but still had to be marked as false, since such a *http* request would not be accepted.

The analysis for the **Parameter Tree** pattern resulted in the strongest determined effect size ( $d = 0.66$ ) of the whole experiment. However, there are big differences between the first and second task for this pattern. For the first task, the value for TAU was 5 times higher for the P version than for N. The amount of correct answers for N is 17.14% and thus represents the lowest value in this research, while for P 66.67% answered correctly. One possible reason might be the long HTTP response that was shown to the participants. A majority of the incorrect answers counted 8 *computer science students*, while the correct answer was 4. However, 8 students were shown in total in the HTTP response, which leads to the assumption that the attribute *studentX\_course* for the student objects went unnoticed or the participants considered all students to be in this study course. The problem was not discovered during the pilot phase of the survey, but if the controlled experiment should get reproduced or repeated, the task description should be altered. As a result, the relative high effect size for the **Parameter Tree** pattern has to be viewed with the disclaimer that the tasks description might not be ideal. Additionally, this pattern is often used implicitly by software engineers and finding a N version where the pattern was not used, with the task version still being comparable to the P version, was a big challenge. The second task for this pattern was additionally the only task where all participants answered correctly, for the P and N version of the task. One possible cause might be the just described challenge to find appropriate tasks for both versions. Having also the two lowest mean durations of the whole experiment, suggests that the second task was also too unnatural and thus too easy.

The **API Gateway** is the only of the six chosen patterns that is taken from Richardson [Ric18]. The differences between task 1 and 2 for this pattern are surprising. While TAU is higher for both tasks, the relation of correct answers was only for the first task much higher for P than N. For the second task, 80% of participants correctly answered the P version, while 97.14% correctly answered the N version. Since both tasks and described scenarios for this pattern are almost identical, with only the naming of components being changed, the potential reasons for this finding are few. Also, since the order of the tasks is fixed and both these versions are shown on the 11th position for the respective group, effects like fatigue or boredom are unlikely reasons. Instead, the cause is most likely related to the order of which a participant gets to see the N and P version of the **API Gateway** tasks. If a participant is put in group 1, *AG1P* is shown at position 5 and *AG2N* at position 11. For participants in group 2, *AG1N* is shown at position 5 and *AG2P* at position 11. Since for group 2, participants already got in touch with the Non-Pattern version (*AG1N*) early on, which should in theory require a more complex solution than for the Pattern version P task, they remembered the context and noticed differences in the task scenario when *AG2P* was shown. Thus, deriving or remembering possible advantages of the pattern and therefore answering the comprehension question correctly. This phenomenon might initiate a discussion for future experiments, whether tasks in a crossover design should not be too similar. The trivial counterargument is that they might be less comparable if the scenarios are too different. A good balance between these two extremes has to be found.

In conclusion, the use of Microservice API patterns does indeed increase the software quality in a significant manner, measured via TAU. The null hypothesis  $H_0$  can be discarded and the alternative hypothesis  $H_1$  can be accepted for 5 out of 6 examined patterns. It has to be noted that the effect size for all significant findings was only of small or medium size. Additionally, it must be emphasized that this study focused on the impacts on understandability and usability and the patterns were all

examined in isolation. For the implications for the industry, this means that the findings might be only valid or noticeable in specific circumstances. Implications of high value for industry practice could be deducted with additional research with less artificial scenarios, different patterns or a focus on different quality attributes.

Regarding the exploratory part of this study with the examination of demographic data and their potential impact on the thematic results, no significant correlations could be found. The main profession, academic or industrial background and the years of experience in the IT area do not seem to impact the gathered results in any obvious way. Furthermore, with the construction of linear regression models, only a very weak positive correlation between the experience in the three self-assessed areas and the experience of participants could be found. However, it has to be noted that this was only a secondary research objective with no clear hypothesis. Future work on this topic might thus be necessary, to fully understand the influence on the effectiveness of (Microservice) design patterns, by the demographic background of users.

## 5.2 Threats to Validity

The following section describes the threats to validity that could be present in the results of this research. Additionally, the actions that were taken to diminish threats are explained. The distinction between constructed, internal and external validities is used.

### 5.2.1 Construct Validity

The construct validity describes how effectively the research examines the theories that it was supposed to examine [CC79]. To combat the possible threats, these steps were taken when designing and executing the experiment: The experiment contained two tasks per pattern, which lowers the threats to validity in general [Fei22]. However, the design patterns were all examined in isolation where the Pattern version is being compared to the Non-Pattern version of the given pattern. No combination of at least two patterns was created for the tasks of the experiment. Further, the focus laid on the two quality attributes understandability and usability. The variables of this study were thus defined to operationalize these attributes. For more general results, the inspection of other or additional attributes is needed. Additionally, other concepts that might be typical in the Microservices and API domain or are common in combination with HTTP requests, were not explicitly added to the tasks for the sake of task simplicity and direct comparability. At the start of the online survey, the participants were informed that the goal of this research is to examine the impact of some API design patterns on software quality. While it was not revealed that the research is about Microservice API patterns, the probability of *hypothesis guessing* [CC79] should be minimized with this information. While the task version (N or P) was not shown to the participants, it is not completely unlikely that some participants identified a difference between the tasks and could then classify them and adjust their answers. Also, the choice of TAU [SBV+21] as a variable that represents a combination of the needed time and the correctness, should decrease the *mono-method bias* [CC79] for this experiment. It is debatable whether the choice of TAU as a variable provides a good operationalization of the impact on software quality. However, TAU as an aggregation of correctness and needed time is used in multiple other computer science studies.

### 5.2.2 Internal Validity

Internal validity depicts the confidence in the collected findings and the conclusions that are derived from these results [WRH+12]. It considers the relationship between a cause and a noticed effect and takes possible alternative causes of an observation into account. Many experiment design decisions were made at the start of this research, to minimize possible threats to the internal validity of the findings: The online survey had a mean duration of 15.5 minutes and was thus within a duration range that should restrict the amount of subjects that stopped the survey before the end, because they got bored, fatigued or demotivated (*mortality*) [WRH+12]. Additionally, the crossover design of the experiment adds randomization, so participants get one of two predefined task orders. The two possible task orders were arranged in a structured way, to minimize the threat through *familiarization*. The order of possible answers in single-choice tasks was also randomized. Although, randomization was used for the seeding of the participants, the two groups still show an imbalance in regards of thematic experience, as shown in Section 4.3. The absence of *random irrelevancies* [WRH+12] cannot be guaranteed, especially since the survey was conducted online. Therefore, there was no direct control of the participants by the research team during the survey. The participants could conduct the survey in any environment, as long as a internet connection and necessary hardware was present. To minimize the threat of participants getting distracted during the survey, the time constraint for tasks was introduced in the data analysis step. Thus, responses were discarded if participants needed considerably longer than the average participant. Since the participation in the survey was completely anonymous, there was no possibility to prevent people from participating multiple times.

### 5.2.3 External Validity

The external validity describes the degree to which the results can be generalized to a similar context or to a broader part of the population [Mad10]. In a controlled experiment the external validity of the results can be affected by threats, via the number of participants or the unequal distribution of experience between the two experiment groups. The understandability of the survey tasks and the time and effort that is needed to complete them, can also form a threat. To mitigate these threats, the following measures were implemented in the experiment design: The experiment was conducted in a form of an online survey, to increase the total number of eligible participants. At the same time, the variety of participants was increased by formulating the requirements in a way so that experts from academia and industry, as well as students on a M.Sc. level could take part. The formulated tasks covered six Microservice API patterns that focused mainly on the improvement of usability and understandability. A generalization to a superset of software design patterns, especially one with patterns that focus on other software attributes, might not be feasible. With almost 100 total participants, providing 65 valid response sets that were used for the analysis and interpretation, the sample size is satisfactory. However, even more participants could further increase the power of the results and reduce threats. The tasks were all formulated with care, so they resemble real tasks that could be found in practice. Nonetheless, the differences of the test environment to a real-world setting are present. The participants of the experiment were only provided with a description for a scenario, a screenshot and sometimes a syntax hint. In practice, developers could look up a pattern documentation, HTTP tutorial or ask colleagues for help.



## 6 Conclusion and Outlook

The Microservice API patterns represent a set of new templates for frequent occurring problems in the environment of Microservices. One main motivation for the use of software design patterns is the increase in software quality. While there is not much empirical evidence for that claim, the research in regards to Microservice patterns is even scarcer. Therefore, a controlled experiment using an online survey was conducted. Initially 98 participants took part in the survey, **65 of which delivered usable responses** for further analysis. The main focus for this study lies on the impact on the two quality attributes *understandability* and *usability*. 5 fitting design patterns from Zimmermann et al. and 1 pattern from Richardson were chosen for the experiment [ZSL+22] [Ric18]. The introduction of the variable *Timed Actual Understandability* was introduced, to fairly compare the responses of the participants in regards to the *correctness* and the *needed time* for the submission of responses. The correctness was the dominant component, which means that no matter how fast a response was given, the value for incorrect answers of TAU was always 0. **For 5 out of the 6 design patterns** the measured software quality was indeed significantly higher for the Pattern versions P of the tasks. Therefore, the **null hypothesis  $H_0$  was rejected for 5** of the patterns with a confidence level of 0.95, after a **p-value correction** with the *Holm-Bonferroni* method was performed [Hol79]. The examination of effect sizes with *Cohen's d* showed that the **Parameter Tree** pattern results can be classified as a **medium-sized effect** and for the other **4 patterns** with significant findings, the effect size was of **small size**. Even though the **Request Bundle** pattern did not result in a significant improvement of software quality in our experiment, the mean TAU values were still slightly better for the versions, where the pattern was incorporated.

As a conclusion, the compliance of the examined design patterns is recommended by the author for the development of Microservice-based systems, if the understandability or usability of the system is of high importance. The use of these patterns can contribute to the improvement of understandability or usability of the developed software. However, as shown in various related work, the usage of design patterns is not guaranteed to always improve the quality of a system, especially if the environment or the requirements are ignored. The found improvements of the two quality attributes by the patterns seem to be unaffected by demographic factors. The experience of the participants, expressed by the number of years in IT, experience with HTTP APIs, software design patterns and Microservice API patterns, do not seem to have any significant impact on the Timed Actual Understandability. For the future, the replication of these findings with a different sample of participants or newly formulated tasks is very much welcomed. Additionally, testing other Microservice API patterns, focusing on other quality attributes, combining several patterns within a task, further investigating the demographic impact, or having a different experiment structure, can further reinforce the impact of such design patterns on software quality. Finally, to enable future work and replication, the experiment materials including non-identifiable survey data can be found on *Zenodo*<sup>1</sup>.

---

<sup>1</sup><https://doi.org/10.5281/zenodo.7754269>



# Bibliography

- [AE13] M. Ali, M. O. Elish. “A Comparative Literature Survey of Design Patterns Impact on Software Quality”. In: *International Conference on Information Science and Applications* (2013), pp. 1–7. doi: [10.1109/ICISA.2013.6579460](https://doi.org/10.1109/ICISA.2013.6579460) (cit. on p. 26).
- [AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. ISBN: 0-19-501919-9 (cit. on p. 17).
- [AP19] A. Akbulut, H. G. Perros. “Performance Analysis of Microservice Design Patterns”. In: *IEEE Internet Computing, Vol. 23, No. 6* (Nov. 2019), pp. 19–27. doi: [10.1109/MIC.2019.2951094](https://doi.org/10.1109/MIC.2019.2951094) (cit. on pp. 24, 27).
- [Avo01] N. M. Avouris. “An Introduction to Software Usability”. In: *Workshop on Software Usability, 8th Panhellenic Conference on Informatics, Vol. 2* (Nov. 2001), pp. 514–522 (cit. on p. 24).
- [Bad02] A. N. Badre. *Shaping Web Usability: Interaction Design in Context*. Addison-Wesley, Jan. 2002. ISBN: 0-201-72993-8 (cit. on p. 24).
- [BBK+78] B. W. Boehm, J. Brown, H. Kaspar, M. Lipow, G. MacCleod. *Characteristics of Software Quality, TRW Series of Software Technology, Vol. 1*. North Holland, 1978. ISBN: 044-4-851-054 (cit. on p. 24).
- [BC87] K. Beck, W. Cunningham. “Using Pattern Languages for Object-Oriented Program”. In: *OOPSLA '87 workshop on Specification and Design for Object-Oriented Programming* (Sept. 1987) (cit. on p. 17).
- [BDD+19] A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, A. Sadovykh. *Microservices: Science and Engineering*. Springer International Publishing, 2019. ISBN: 9783030316464 (cit. on p. 20).
- [BHJ16] A. Balalaie, A. Heydarnoori, P. Jamshidi. “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture”. In: *IEEE Software, Vol. 33, No. 3* (2016), pp. 42–52. doi: [10.1109/MS.2016.64](https://doi.org/10.1109/MS.2016.64) (cit. on p. 20).
- [BPS+12] S. Barney, K. Petersen, M. Svahnberg, A. Aurum, H. Barney. “Software quality trade-offs: A systematic map”. In: *Information and software technology, Vol. 54, No. 7* (2012), pp. 651–662 (cit. on p. 24).
- [BWZ19] J. Bogner, S. Wagner, A. Zimmermann. “On the impact of service-oriented patterns on software evolvability: a controlled experiment and metric-based analysis”. In: *PeerJ Computer Science 5* (Aug. 2019). doi: [10.7717/peerj-cs.213](https://doi.org/10.7717/peerj-cs.213) (cit. on p. 25).
- [CC79] T. D. Cook, D. T. Campbell. *Quasi-Experimentation: Design & Analysis Issues for Field Settings*. Houghton Mifflin, July 1979. ISBN: 0395307902 (cit. on p. 55).
- [Coh88] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Routledge, 1988, pp. 19–20. ISBN: 978-1-134-74270-7 (cit. on p. 42).

- [EB18] J. Eloff, M. B. Bella. *Software Failure Investigation: An Overview*. Springer International Publishing, 2018, pp. 7–24. ISBN: 978-3-319-61333-8 (cit. on p. 23).
- [Fei22] D. G. Feitelson. “Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension”. In: *Empirical Software Engineering*, Vol. 27, No. 6 (2022), p. 123 (cit. on p. 55).
- [FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. ISBN: 3-70-911953-7 (cit. on p. 17).
- [Gef18] J. Gefroh. “Why consistency is one of the top indicators of good code”. In: *Medium* (Sept. 2018). URL: <https://jgefroh.medium.com/why-consistency-is-one-of-the-top-indicators-of-good-code-352ba5d62020> (cit. on p. 19).
- [GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN: 978-0-201-63361-0 (cit. on pp. 17, 18).
- [Hol79] S. Holm. *A Simple Sequentially Rejective Multiple Test Procedure*. Scandinavian Journal of Statistics, Vol. 6, No. 2, 1979, pp. 65–70 (cit. on pp. 42, 57).
- [Int01] International Organization For Standardization/International Electrotechnical Commission and others. *ISO/IEC 9126 – Software Engineering – Product Quality*. Geneve, 2001 (cit. on p. 23).
- [Int11a] International Organization For Standardization. *ISO 26262 - Road vehicles — Functional safety*. 2011 (cit. on p. 23).
- [Int11b] International Organization For Standardization/International Electrotechnical Commission and others. *ISO/IEC 25010 - Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuARE) - System and Software Quality Models*. 2011 (cit. on p. 23).
- [Int20] International Organization For Standardization. *ISO 9241-110:2020 – Ergonomics of human-system interaction — Part 110: Interaction principles*. 2020 (cit. on p. 24).
- [Ken38] M. G. Kendall. “A new measure of rank correlation”. In: *Biometrika*, Vol. 30, No. 1-2 (June 1938), pp. 81–93. DOI: [10.1093/biomet/30.1-2.81](https://doi.org/10.1093/biomet/30.1-2.81) (cit. on p. 42).
- [KG08] F. Khomh, Y.-G. Gueheneuc. “Do Design Patterns Impact Software Quality Positively?” In: *12th European Conference on Software Maintenance and Reengineering* (2008), pp. 274–278. DOI: [10.1109/CSMR.2008.4493325](https://doi.org/10.1109/CSMR.2008.4493325) (cit. on pp. 19, 25).
- [Kra22] H. Krasner. “The Cost of Poor Software Quality in the U.S.: A 2022 Report”. In: *Consortium for Information & Software Quality* (Dec. 2022). URL: <https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report> (cit. on p. 23).
- [Ley19] F. Leymann. “Towards a Pattern Language for Quantum Algorithms”. In: *Quantum Technology and Optimization Problems*. Vol. 11413. Lecture Notes in Computer Science (LNCS). Springer International Publishing, 2019, pp. 218–230. DOI: [10.1007/978-3-030-14082-3\\_19](https://doi.org/10.1007/978-3-030-14082-3_19) (cit. on p. 18).
- [Lik32] R. Likert. “A Technique for the Measurement of Attitudes”. In: *Archives of Psychology*, Vol. 140 (1932), pp. 1–55 (cit. on p. 44).

- [Löw19] J. Löwy. *Righting Software*. Addison-Wesley, 2019, pp. 73–77. ISBN: 978-0-13-652403-8 (cit. on pp. 20, 21).
- [LW06] J.-C. Lin, K.-C. Wu. “A model for measuring software understandability”. In: *The Sixth IEEE International Conference on Computer and Information Technology (CIT’06)* (2006), pp. 192–192 (cit. on p. 24).
- [Mad10] L. Madeyski. “The impact of Test-First programming on branch coverage and mutation score indicator of unit tests - An experiment”. In: *Information and Software Technology, Vol. 52, No. 2* (Feb. 2010), pp. 169–184. DOI: [10.1016/j.infsof.2009.08.007](https://doi.org/10.1016/j.infsof.2009.08.007) (cit. on p. 56).
- [Mas66] A. H. Maslow. *The Psychology of Science*. Harper & Row, 1966, p. 15. ISBN: 978-0-80-926130-7 (cit. on p. 19).
- [Mat18] S. Matteson. “Software failure caused \$1.7 trillion in financial losses in 2017”. In: *TechRepublic* (Jan. 2018). URL: <https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017> (cit. on p. 23).
- [Mil81] R. G. Miller. *Simultaneous Statistical Inference*. Springer Verlag New York, 1981. ISBN: 978-0-387-90548-8. (Cit. on p. 42).
- [MJM00] R. Mittelhammer, G. Judge, D. Miller. *Econometric Foundations*. Cambridge University Press, May 2000, pp. 73–74. ISBN: 978-0-521-62394-0 (cit. on p. 42).
- [MP09] B. S. Medikonda, S. R. Panchumarthy. “A Framework for Software Safety in Safety-Critical Systems”. In: *SIGSOFT Softw. Eng. Notes* (Feb. 2009), pp. 1–9. DOI: [10.1145/1507195.1507207](https://doi.org/10.1145/1507195.1507207) (cit. on p. 23).
- [MPV21] D. C. Montgomery, E. A. Peck, G. G. Vining. *Introduction to linear regression analysis*. John Wiley & Sons, 2021 (cit. on p. 51).
- [MW02] J. L. Myers, A. D. Well. *Research Design & Statistical Analysis*. Lawrence Erlbaum Associates Inc., 2002. ISBN: 978-0-8058-4037-7 (cit. on p. 42).
- [MW47] H. Mann, D. Whitney. “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other”. In: *Annals of Mathematical Statistics, Vol. 18, No. 1* (Mar. 1947), pp. 50–60. DOI: [10.1214/aoms/1177730491](https://doi.org/10.1214/aoms/1177730491) (cit. on p. 42).
- [Nor96] P. Norvig. *Design patterns in dynamic programming*. Object World 96.5, 1996 (cit. on p. 19).
- [Pea95] K. Pearson. “Note on Regression and Inheritance in the Case of Two Parents”. In: *Proceedings of the Royal Society of London Series I* (Jan. 1895), pp. 240–242 (cit. on p. 42).
- [Pfa21] T. Pfaff. “Haben Design-Regeln Einfluss auf die Verständlichkeit von RESTful APIs? Ein kontrolliertes Experiment”. In: *Master Thesis* (Nov. 2021). DOI: [10.18419/opus-12076](https://doi.org/10.18419/opus-12076) (cit. on p. 25).
- [PZA+17] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, N. Josuttis. “Microservices in Practice, Part 2: Service Integration and Sustainability, Vol. 34, No. 2”. In: *IEEE Software* (Mar. 2017), pp. 97–104. DOI: [10.1109/MS.2017.56](https://doi.org/10.1109/MS.2017.56) (cit. on p. 21).
- [Ric18] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018. ISBN: 9781617294549 (cit. on pp. 15, 22, 23, 35, 54, 57).

- [Rod05] P. Rodgers. “Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity”. In: *CloudComputingExpo* (Feb. 2005) (cit. on p. 20).
- [Saw09] S. Sawilowsky. “New Effect Size Rules of Thumb”. In: *Journal of Modern Applied Statistical Methods*, Vol. 8, No. 2 (Nov. 2009), pp. 597–599. DOI: [10.22237/jmasm/1257035100](https://doi.org/10.22237/jmasm/1257035100) (cit. on p. 42).
- [SBV+21] S. Scalabrino, G. Bavota, C. Vendome, M. Linares, D. Poshyvanyk, R. Oliveto. “Automatically assessing code understandability”. In: *IEEE Transactions of Software Engineering*, Vol. 47, No. 3 (Mar. 2021). DOI: [10.1109/TSE.2019.2901468](https://doi.org/10.1109/TSE.2019.2901468) (cit. on pp. 30, 55).
- [SW65] S. Shapiro, M. Wilk. “An analysis of variance test for normality (complete samples)”. In: *Biometrika*, Vol. 52, No. 3-4 (Dec. 1965), pp. 591–611. DOI: [10.1093/biomet/52.3-4.591](https://doi.org/10.1093/biomet/52.3-4.591) (cit. on p. 42).
- [Til14] S. Tilkov. “How small should your microservice be?” In: *innoQ - Stefan Tilkov's Blog Archive* (Nov. 2014). URL: <https://www.innoq.com/blog/st/2014/11/how-small-should-your-microservice-be> (cit. on p. 20).
- [TMCC10] L. Thabane, J. Ma, R. Chu, J. Cheng. “A tutorial on pilot studies: the what, why and how”. In: *BMC medical research methodology*, Vol. 10 1. (2010). DOI: [10.1186/1471-2288-10-1](https://doi.org/10.1186/1471-2288-10-1) (cit. on p. 34).
- [VAJ16] S. Vegas, C. Apa, N. Juristo. “Crossover Designs in Software Engineering Experiments: Benefits and Perils”. In: *IEEE Transactions on Software Engineering*, Vol. 42, No. 2 (Feb. 2016), pp. 120–135. DOI: [10.1109/TSE.2015.2467378](https://doi.org/10.1109/TSE.2015.2467378) (cit. on p. 30).
- [VCG+22] G. Vale, F.F. Correia, E.M. Guerra, T. de Oliveira Rosa, J. Fritzsich, J. Bogner. “Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs”. In: *2022 IEEE 19th International Conference on Software Architecture* (2022), pp. 69–79. DOI: [10.1109/ICSA53651.2022.00015](https://doi.org/10.1109/ICSA53651.2022.00015) (cit. on p. 26).
- [WA20] F. Wedyan, S. Abufakher. “Impact of design patterns on software quality: a systematic literature review”. In: *IET Software*, Vol. 14, No. 1 (Feb. 2020). DOI: [10.1049/iet-sen.2018.5446](https://doi.org/10.1049/iet-sen.2018.5446) (cit. on p. 25).
- [WRH+12] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in Software Engineering*. May 2012, pp. 123–151. ISBN: 978-3-642-29043-5 (cit. on p. 56).
- [Zim19] O. Zimmermann. “Domain-Specific Service Decomposition with Microservice API Patterns”. In: *International Conference on Microservices* (Feb. 2019) (cit. on p. 20).
- [ZSL+22] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, C. Pautasso. *Patterns for API Design*. 1st Edition. Pearson International, 2022. ISBN: 978-0-13-767010-9 (cit. on pp. 15, 21, 22, 35, 57).

All links were last followed on March 21, 2023.