Technische Universität München
TUM School of Computation, Information and Technology

API Management Practices and Patterns for Public, Partner, and Group Web APIs with a Focus on Knowledge Transfer and Collaboration

Gloria Mercedes Bondel

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung einer

Doktorin der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 10.01.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 17.06.2024 angenommen.

Abstract

Web Application Programming Interfaces (APIs) are the de facto standard for making data and functionality accessible across organizational boundaries. As a result, Web APIs allow provider organizations to realize new business models, become platform providers, support efficient partner integration, or enable compliance (e.g., in banking). However, Web APIs require careful design and management to realize these benefits. Nevertheless, to the best of the author's knowledge, explicitly formulated API management guidance is scarce in academia.

Analyses of the API Economy ecosystem show that relatively young, digital organizations in the US dominate the provision of APIs across organizational boundaries. Also, managing web APIs at the interface between several stakeholders inside and outside an organization makes API management an inherently collaborative organizational function. Therefore, this dissertation aims to identify API management best practices focusing on knowledge transfer and collaboration for different API provider organizations, including established organizations in traditional sectors seated in Europe.

First, we focus on knowledge transfer between the API provider and potentially unknown, heterogeneous, and distributed API consumers. We identify 48 best practice candidates for code examples in API documentation from 17 research papers and 13 expert interviews and categorize them according to their knowledge types. In addition, we evaluate eight best practice candidates using a case study. The case study results show six best practice candidates that can be confirmed to be actual best practices. Furthermore, we derive several implications for code examples in API documentation. One such implication states that the identified best practices are context-dependent.

Hence, the main contribution of this dissertation is the API management pattern catalog (AMPC). The AMPC presents 22 patterns and 37 pattern candidates for managing Web APIs used across company boundaries and focuses on an API management team's interactions with external and internal stakeholders. In addition, the AMPC presents an overview of stakeholders involved in API management and relates the patterns of AMPC to related pattern languages and catalogs. We created the AMPC using a design science approach encompassing the initial analysis of 14 case descriptions and refinement based on the scientific pattern community's feedback. The subsequent evaluation employing a survey with 18 IT professionals shows the AMPC's applicability, comprehensibility, usability, and correctness. However, the evaluation also indicates that the AMPC is incomplete but provides a starting point for designing a holistic approach to API management that requires extension, refinement, and regular updates in the future.

Zusammenfassung

Web Application Programming Interfaces (Web APIs) sind der De-facto-Standard für den Daten- und Funktionsaustausch über Unternehmensgrenzen hinweg. Web APIs ermöglichen es Anbietern neue Geschäftsmodelle zu verwirklichen, zu Plattformanbietern zu werden, Partner effizient zu integrieren oder Regularien einzuhalten (z. B. im Bankwesen). Um diese Vorteile zu realisieren, müssen Web APIs jedoch sorgfältig konzipiert und gesteuert werden. Nach bestem Wissen der Autorin gibt es in der Wissenschaft aktuell jedoch kaum explizit formulierte API Management Handlungsempfehlungen.

Analysen des API Economy Ökosystems zeigen, dass relativ junge, digitale Organisationen in den USA die Bereitstellung von APIs über Organisationsgrenzen hinweg dominieren. Zudem macht das Management von Web APIs an der Schnittstelle zwischen verschiedenen Stakeholdern innerhalb und außerhalb einer Organisation das API Management zu einer inhärent kollaborativen Organisationsfunktion. Daher ist es das Ziel dieser Dissertation, Best Practices für API Management mit einem Fokus auf Wissenstransfer und Kooperation zu identifizieren, welche anwendbar für verschiedene Typen von API Anbietern sind, einschließlich etablierter Organisationen in traditionellen Sektoren mit Sitz in Europa.

Zunächst konzentrieren wir uns auf den Wissenstransfer zwischen dem API Anbieter und potenziell unbekannten, heterogenen und verteilten API Nutzern. Wir identifizieren 48 Best Practice Kandidaten für Codebeispiele in der API Dokumentation in 17 Wissenschaftsbeiträgen und 13 Experteninterviews und kategorisieren sie nach ihren Wissenstypen. Darüber hinaus evaluieren wir acht Best Practice Kandidaten in einer Fallstudie. Das Ergebnis der Fallstudie bestätigt sechs der Best Practice Kandidaten als Best Practices. Darüber hinaus leiten wir mehrere Implikationen für Codebeispiele in API Dokumentation ab. Eine dieser Implikationen besagt, dass die identifizierten Best Practices kontextabhängig sind.

Der Hauptbeitrag dieser Dissertation ist daher der API Management Patternkatalog (AMPC). Der AMPC enthält 22 Pattern und 37 Pattern Kandidaten für das Management von unternehmensübergreifend genutzten Web APIs mit einem Fokus auf die Interaktionen eines API Management Teams mit externen und internen Stakeholdern. Darüber hinaus bietet der AMPC einen Überblick über die am API Management beteiligten Akteure und setzt die Pattern des AMPC mit verwandten Pattern Sprachen und Katalogen in Beziehung. Wir haben einen Design Science Ansatz genutzt um den AMPC zu erstellen, der die anfängliche Analyse von 14 Fallbeispielen und eine Verbesserung auf der Grundlage des Feedbacks der wissenschaftlichen Pattern Community umfasst. Die anschließende Evaluation anhand einer Umfrage mit 18 IT-Experten zeigt die Anwendbarkeit, die Verständlichkeit, die Benutzerfreundlichkeit und die Korrektheit des AMPC. Die Evaluation zeigt jedoch auch, dass der AMPC noch unvollständig ist, jedoch einen Ausgangspunkt für die Entwicklung eines ganzheitlichen Ansatzes für API Management bietet, der in Zukunft erweitert, verbessert und regelmäßig aktualisiert werden muss.

Acknowledgment

First and foremost, I want to express my special appreciation and thanks to my supervisor Prof. Dr. Florian Matthes. He allowed me to conduct this research under his supervision and trusted me with many exciting projects. I am immensely grateful for everything that I learned during my time at the chair. I further want to thank Prof. Dr. Martin Bichler for being my second supervisor.

In addition, I would like to thank my mentor PD Dr.-Ing. Sven-Volker Rehm for his support during my research endeavor. Our discussions were very motivating and helped me cast new perspectives onto my research.

This work benefited greatly from the cooperation of many industry experts. Hence, I want to thank all industry partners who supported this research by participating in interviews, discussions, case studies, and surveys.

Many great colleagues accompanied me during my time at the chair. My special thanks go to Anne Faber, Dominik Huth, Ulrich Gallersdörfer, Adrian Hernandez-Mendez, and Daniel Braun for their constructive cooperation and the good times. I am also grateful for all the students who entrusted me with advising their theses and collaborated with me on various projects. Special thanks go to Andre Landgraf, Arif Cerit, Duc Huy Bui, Sharada Sowmya, Fridolin Koch, and Kevin Baumer.

Finally, I want to thank my parents Brigitte and Richard, my husband Sebastian, my family, and my friends for their unwavering support, patience, and encouragement. I am beyond grateful and full of love to have you all in my life, knowing you will always have my back.

München, 05.01.2024

Gloria Bondel

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction and Motivation

The use of Information Technology (IT) to exchange data across organizational boundaries and thus enabling inter-firm processes has been investigated since the 1960s (Lyytinen and Damsgaard, 2011). However, the success of early inter-firm communication and data exchange approaches based on proprietary networks like Electronic Data Exchange (EDI) was limited due to the need for high upfront, asset-specific investments leading to point-to-point integrations (Christiaanse et al., 2004). However, with the rise of the internet as an open, non-proprietary infrastructure, inter-firm data exchange became cost-efficient, enabling new forms of relationships between organizations (Christiaanse et al., 2004; Lyytinen and Damsgaard, 2011). More precisely, due to their simplicity and understandability (Kopecký et al., 2014; Tan et al., 2016; Maleshkova et al., 2010), *Web APIs* emerged as the de facto standard for data exchange between organizations.

Web APIs[1] are machine- and human-readable interfaces that allow *client applications* to access functionality and data provided by *backends* (Bermbach and Wittern, 2016; Santoro et al., 2019). Web APIs rely on Web technologies, particularly using the Hypertext Transfer Protocol (HTTP) as application protocol to transfer messages (Wittern et al., 2017; Bermbach and Wittern, 2016). By encapsulating data in messages, Web APIs allow the instantiation of the *information hiding* principle (Parnas, 1972). Therefore, a backend and client applications can exchange messages independent of their underlying platform. Moreover, the communicating components can evolve without affecting each other as long as the Web API contract remains stable (Daigneau, 2011).

Web APIs can be categorized according to the audience that can access them. First, *public* Web APIs are generally accessible to all third-party developers that accept the Web API's terms and conditions and other contractual agreements (Jacobson et al., 2012). In comparison,

[1] In this dissertation, we use the terms API and Web API interchangeably to refer to Web APIs if not explicitly stated otherwise.

partner Web APIs are accessible only to a restricted group of developers belonging to one or several organizations outside the API provider organization (De, 2017). Such partner APIs often provide the basis for integrations guided by individual contractual agreements (Jacobson et al., 2012). Next, *group* Web APIs are accessible to subsidiaries belonging to the same group as the API provider (Bondel et al., 2021b). Lastly, *internal* Web APIs are accessible exclusively to developers inside the API provider organization (Jacobson et al., 2012). Since this dissertation focuses on Web APIs with the API provider and consumers belonging to different organizations, public, partner, and group Web APIs are in scope.

Current Information Systems (IS) research conceptualizes public, partner, and group Web APIs as resources at the interface between an organization and third-party developers (Evans and Basole, 2016; Basole, 2016, 2019) enabling several advantages for API providers.

First, Web APIs enable API providers to realize new business models (Evans and Basole, 2016; Basole, 2016, 2019). Organizations can make previously internal data, functionality, or products accessible to external consumers (Weiss and Gangadharan, 2010; Basole, 2019). As a result, these provider organizations can benefit from new revenue streams by monetizing access to their assets using different business models like subscriptions, freemium, or pay-as-you-go (Evans and Basole, 2016; Basole, 2016, 2019). A well-known example for an organization that successfully built its business model on Web APIs is Salesforce[2], which initially offered its Customer Relationship Management (CRM) tool only via Web APIs. Similarly, Twilio[3] successfully offers Short Message Service (SMS), Email, and other communication services exclusively via Web API (Iyer and Subramaniam, 2015).

Secondly, Web APIs enable organizations to become product platform providers (Yoo et al., 2010). Product platform providers allow consumers to access their platform's core modules to develop complementing applications (Ghazawneh and Henfridsson, 2013). Examples of successful product platforms are the iPhone iOS[4] and Android[5] operating systems with their proliferating app ecosystems (de Reuver et al., 2018).

Thirdly, Web APIs enable efficient partner integration. Organizations can become consumers of Web APIs and rent functionality, benefiting from the "Everything-as-a-Service (XaaS)" paradigm (Basole, 2019). As a result, they allow for more efficient IT management (Hagel III and Brown, 2001). A recent study questioning 1,050 IT leaders across the globe revealed that at the beginning of 2023, 81% of organizations used public APIs (MuleSoft, 2023).

Finally, in some business sectors, regulators or consortia require organizations to make data accessible to specific consumers. For example, in banking, the Revised Payment Services Directive (PSD2) (EU Directive 2015/2366, 2015) forces retail banks to make customer accounts accessible to particular types of organizations. Banks can implement such access through Web APIs. Similarly, in the automotive sector, Original Equipment Manufacturers (OEMs) committed themselves to providing secure access to vehicle-generated data to consumers through Web APIs (ISO 20077-1; ISO 20078-1; ISO 20080).

[2]https://www.salesforce.com/de/
[3]https://www.twilio.com/en-us
[4]https://www.apple.com/de/ios/ios-17/
[5]https://www.android.com/intl/de_de/

On the other hand, consumers also benefit from public, partner, and group Web APIs. First, consumers can combine Web APIs (Weiss and Gangadharan, 2010) to create *mashups* (Evans and Basole, 2016; Basole, 2019; Basole et al., 2018). Also, they can efficiently source IT functionality through XaaS offers.

As a result, a new service ecosystem consisting of Web API providers, consumers, and other stakeholders like aggregator platforms and regulators has emerged, i.e., the so-called *API Economy* (Basole, 2019; Weiss and Gangadharan, 2010). Within the API Economy, providers and consumers can generate new value using Web APIs (Huhtamäki et al., 2017; Basole, 2016; Evans and Basole, 2016; Basole, 2019). Moreover, Web APIs enable in parts unanticipated innovation through service recombinations (Yoo et al., 2010; Eaton et al., 2015; Basole, 2019). Therefore, Web APIs are strategically valuable resources, and firms must design and maintain them carefully (Yoo et al., 2010).

1.1. Problem Statement

However, past ecosystem analyses show that the provision of successful Web APIs is not distributed evenly across organizations. Instead, relatively young, digital organizations are more likely to provide public Web APIs, and those APIs are integrated more often by third-party developers than APIs of established organizations, i.e., organizations that were traditionally not digital (Evans and Basole, 2016; Basole, 2019). For example, Amazon[6] provides several Web APIs and these APIs are integrated in many mashups, while the established retailers Walmart[7] and Macy's[8] provide only a small number of Web APIs which are rarely integrated (Evans and Basole, 2016).

Moreover, organizations located in major entrepreneurial regions, especially in Silicon Valley in the United States of America (US), provide the majority of successful open APIs (Huhtamäki et al., 2017). Specifically, California alone provides more public Web APIs than the whole of Europe, suggesting an untapped opportunity for European firms (Huhtamäki et al., 2017). Also, companies providing services related to social, mapping, search, online payment, image sharing, video, and messaging provide more successful Web APIs compared to firms operating in traditional sectors like banking, insurance, pharmaceuticals, food, transportation, or energy (Evans and Basole, 2016).

Nevertheless, our experiences and interactions with European organizations in different traditional sectors indicate that many plan or already started providing public, partner, or group Web APIs in the last few years.

However, Web APIs must attract heterogeneous consumers to be successful (Yoo et al., 2010). Since consumers rely on the performance and availability of the Web APIs that they integrate (Bermbach and Wittern, 2016), they do not integrate poorly designed, unattractive, or too expensive Web APIs. Also, they abandon Web APIs if they stop meeting their needs (Huhtamäki et al., 2017). Therefore, it is not sufficient for an organization to simply publish a Web

[6] `https://www.amazon.com/`
[7] `https://www.walmart.com/`
[8] `https://www.macys.com/`

API (Basole, 2019; Ghazawneh and Henfridsson, 2010), even if the functionality and data it makes accessible are valuable. Instead, a Web API management strategy, including technical, social, organizational, and process-oriented aspects, is essential for organizations to tap into the potential of the API Economy (Yoo et al., 2010; Basole, 2019). A Web API management strategy must, among other things, consider Web API design, discoverability, support services, community management, monitoring, and monetization (Basole, 2019).

Research on Web APIs as boundary resources is relatively abstract making it difficult for organizations to derive recommendations for action easily. As de Reuver et al. (2018) states: *"[...] research on digital platforms has so far not revealed much direct design knowledge"* (de Reuver et al., 2018, p. 129). On the other hand, practice-driven literature provides more specific API management guidelines but these guidelines are often concerned with rather technical aspects of Web API management, e.g., how to design endpoints to achieve RESTful compliance or how to implement authentication and authorization mechanisms to increase security (see Chapter 3).

Furthermore, API management is an organizational function at the interface between provider organization and an external consumers. Therefore, knowledge transfer between the API provider team and different stakeholders inside and outside of the organization is necessary (Islind et al., 2016). Supporting this assumption, Yoo et al. (2010) calls for the identification of *"[...] appropriate principles that govern the social context of developments of boundary resources and digital components [...]* (Yoo et al., 2010, p. 733) in platform settings. Nevertheless, to the best of the authors' knowledge, only few best practices focusing on knowledge transfer and collaboration in API management have been explicitly formalized.

1.2. Research Questions

The research objective of this dissertation is:

> The identification of API management best practices and patterns focusing on knowledge transfer and collaboration in public, partner, and group API initiatives, for different types of API provider organizations, including established organizations in traditional sectors seated in Europe.

We aim to achieve this objective by answering the following Research Questions (RQs).

RQ1: What is the current state of research on API management?

First, we analyzed the current state of research in the field of API management.

RQ 1.1 *How is API management defined in academia?*
To build a solid foundation and introduce a clear terminology for the dissertation at hand, we reviewed scientific and practice-driven literature on API management, API management best practices, and API management patterns. We present definitions of relevant terms in Chapter 2, including a working definition for API management.

RQ 1.2 *What patterns for API management exist in research and practice?*
We identified and reviewed 15 pattern collections concerned with API design, service design including SOA and microservices patterns, middleware design, object-oriented software design, and software architecture design. We extracted API management patterns and categorized them along the API management lifecycle presented in Fig. 2.4.

RQ2: What are best practices for transferring knowledge to API consumers using code examples in official public, partner, and group Web API documentation?

Consumers using public, partner, and group Web APIs usually do not have easy access to the developers of the respective API. Hence, API providers need to transfer knowledge to API consumers to enable the use of the Web APIs. Therefore, we identified best practices for code examples in API documentation.

RQ 2.1 *What are best practice candidates for code examples in official public, partner, and group Web API documentation?*
As presented in Chapter 4, we reviewed scientific literature and analyzed expert interviews to identify best practice candidates for code examples in API documentation. As a result, we identified 48 best practice candidates for code examples in Web API documentation. Moreover, we categorized the identified best practice candidates according to the different knowledge types (Thayer et al., 2021) that they aim to convey.

RQ 2.2 *What are validated best practices for code examples in official public, partner, and group Web API documentation?*
Drawing on the results of Chapter 4, we evaluated a subset of best practice candidates in a case study with 12 professional developers. The case study comprised the developers solving tasks using a Web API with the help of a basic or an enhanced documentation version. We describe the case study approach and the results in Chapter 5.

RQ3: What are API management patterns for public, partner, and group Web APIs focusing on collaboration?

API management of Web APIs used across organizational boundaries requires the API provider to collaborate with stakeholders internal and external to the API provider organization. Moreover, the suitability of API management best practices depends on their context. Therefore, we created the *API Management Pattern Catalog for Public, Partner, and Group Web APIs with a Focus on Collaboration* (Bondel and Matthes, 2023), which we abbreviate as AMPC. The AMPC focuses on collaboration between an API management team and internal and external stakeholders of public, partner, and group Web API initiatives. The AMPC constitutes the core contribution of the dissertation at hand.

RQ 3.1 *Who are the stakeholders involved in public, partner, and group Web API management?*
We used a design science research approach (Hevner et al., 2004; Hevner, 2007) as described in Section 6.1 to create the API management pattern catalog (AMPC). As a data basis, we assembled a case base holding 12 cases derived from 16 expert interviews from which we derived stakeholders using Grounded Theory Methodology (GTM) (Wiesche et al., 2017). As described in Section 6.4.2, the AMPC presents nine stakeholders.

RQ 3.2 *What are API management patterns for public, partner, and group Web APIs with a Focus on collaboration?*
The same design science research approach used to answer RQ3.1 yielded the patterns of the AMPC. In addition, we iteratively added more relevant information and refined the pattern form based on feedback from the scientific pattern community. As described in Chapter 6, the AMPC comprises 22 patterns and 37 pattern candidates.

RQ 3.3 *How do the identified API management patterns relate to existing pattern languages and catalogs?*
We reviewed pattern languages and catalogs concerned with the design of APIs or interfaces and interactions between distributed software components in Chapter 3. In Section 6.4.4, we relate these API management patterns to the patterns of the AMPC. Moreover, we present a visual presentation of these relations in Fig. 6.6.

RQ 3.4 *How do practitioners perceive the usefulness of API management patterns for public, partner, and group Web APIs with a focus on collaboration?*
We collected practitioner feedback using a survey. The survey evaluated the applicability, comprehensibility, usability, completeness, and correctness of the AMPC from a practitioner's viewpoint. Overall, 18 experienced practitioners participated in the survey. We describe the survey approach and results in Chapter 7.

1.3. Contributions

This dissertation contributes to research and practice in the field of Web API management with a focus on Web APIs used across organizational boundaries, i.e., public, partner, and group Web APIs.

First, we contribute to the field by introducing a **concise terminology** of terms related to Web API management, best practices, and patterns. Also, we present an analysis of the **state of research** for API management patterns. As a result, we identify the lack of guidance on social aspects of Web API management as a **research gap**.

Afterward, we investigate knowledge transfer between API providers and consumers through documentation to address this research gap. We present **48 best practice candidates** for code examples in Web API documentation. The best practice candidates contribute to practice by providing an overview of potential best practices that can inspire the design of code examples in Web API documentation. Moreover, the best practice candidates are a starting point for evaluating the impact of each best practice candidate on the productivity and satisfaction of API consumers.

Next, we evaluated a subset of the best practice candidates for code examples in public, partner, and group Web API documentation, i.e., in settings in which the API consumers usually do not have easy access to the Web API developer team. The analysis yields **six validated best practices**. These best practices can guide practitioners when creating code examples as part of official Web API documentation. Furthermore, practitioners and researchers can use the best practices to inform the design of tools for the automated generation of code examples in Web API documentation.

Moreover, we derive **eight implications**. Thus, the best practices and implications form a starting point for creating a deeper understanding and holistic approaches for presenting code examples in Web API documentation. Furthermore, the best practice candidates, best practices, and implications can contribute to future theory building.

Next, we present the AMPC comprising **22 patterns and 37 pattern candidates** for managing Web APIs used across organizational boundaries, which forms the core contribution of this dissertation. In addition, we relate the identified API management patterns to existing pattern languages and catalogs. Furthermore, we detail **API management stakeholders** and discuss **observations** made during the API management pattern design.

The AMPC contributes to practice in several ways. First, the AMPC documents proven best practices for managing Web APIs used across organizational boundaries focusing on stakeholder collaboration. This operational knowledge supports practitioners in designing new API initiatives (see Section 7.4). In addition, practitioners can use the AMPC to benchmark current API initiatives with state-of-the-art practices. Also, the AMPC presents a consistent taxonomy that stakeholders can use to communicate. Next, the AMPC documents knowledge that can help educate developers on Web API management. Hence, it supports different types of organizations in reaping the potential of public, partner, and group API initiatives by addressing the challenge of API provider teams. Finally, the documentation of future changes to the pattern catalog will allow researchers to create knowledge on the evolution of the discipline of Web API management.

We conclude this dissertation by reflecting on the contributions and presenting **future work**.

1.4. Outline

Fig 1.1 presents an overview of this dissertation's structure. In the following, we describe the content of each subsequent chapter.

Chapter 2, '**Foundations**', first defines the software artifacts and stakeholders involved in providing and using a Web API, followed by an introduction to typical architectural Web API styles. Focusing on public, partner, and group Web APIs, the chapter further presents an analysis of the API Economy and IS platform research. The analysis shows that Web APIs are strategically essential resources (Yoo et al., 2010) which are unevenly distributed between organizations, whereby traditional companies are at a disadvantage (Evans and Basole, 2016; Basole, 2019; Weiss and Gangadharan, 2010; Huhtamäki et al., 2017). Furthermore, the provision of successful Web APIs requires active management including knowledge transfer and collaboration with API consumers. Hence, the chapter presents a definition and lifecycle for Web API management. Finally, it provides definitions of the terms *best practices*, *practices*, and *patterns.*

Chapter 3, '**Related Work**', reviews pattern collections concerned with API design, service design, middleware design, object-oriented software design, and software architecture pattern languages and catalogs to identify API management patterns. Moreover, we categorize identified API management patterns along the API management lifecycle phases presented in Fig. 2.4.

Chapter 4, '**Identification of Best Practice Candidates for Code Examples in Web API Documentation**,' identifies 46 best practice candidates for code examples in public Web API documentation from literature and interviews. Furthermore, it categorizes the best practice candidates into knowledge types according to Thayer et al. (2021) and identifies a new category capturing the form of the code examples. Finally, the chapter presents implications derived from this categorization.

Chapter 5, '**Evaluation of Best Practices for Code Examples in Web API Documentation**', builds on the previous chapter by evaluating a subset of eight best practice candidates to see if they are actually best practices using a case study. In addition to confirming that six best practice candidates are best practices, the chapter updates and extends the previously presented implications.

Chapter 6, '**Design of the API Management Pattern Catalog (AMPC)**', describes the approach used to design the AMPC. Moreover, it presents an overview of major AMPC contents, i.e., the stakeholders, patterns, their relations to other pattern collections, and pattern candidates.

Chapter 7, '**Evaluation of the API Management Pattern Catalog (AMPC)**', describes the approach to evaluating the AMPC from a practitioner's point of view using a survey. Moreover, it presents the evaluation results. Also, it documents final changes to AMPC before its publication.

The dissertation ends with chapter 8, '**Conclusion and Future Work**', summarizing the findings of the dissertation, answering the research questions, and presenting limitations and future work.

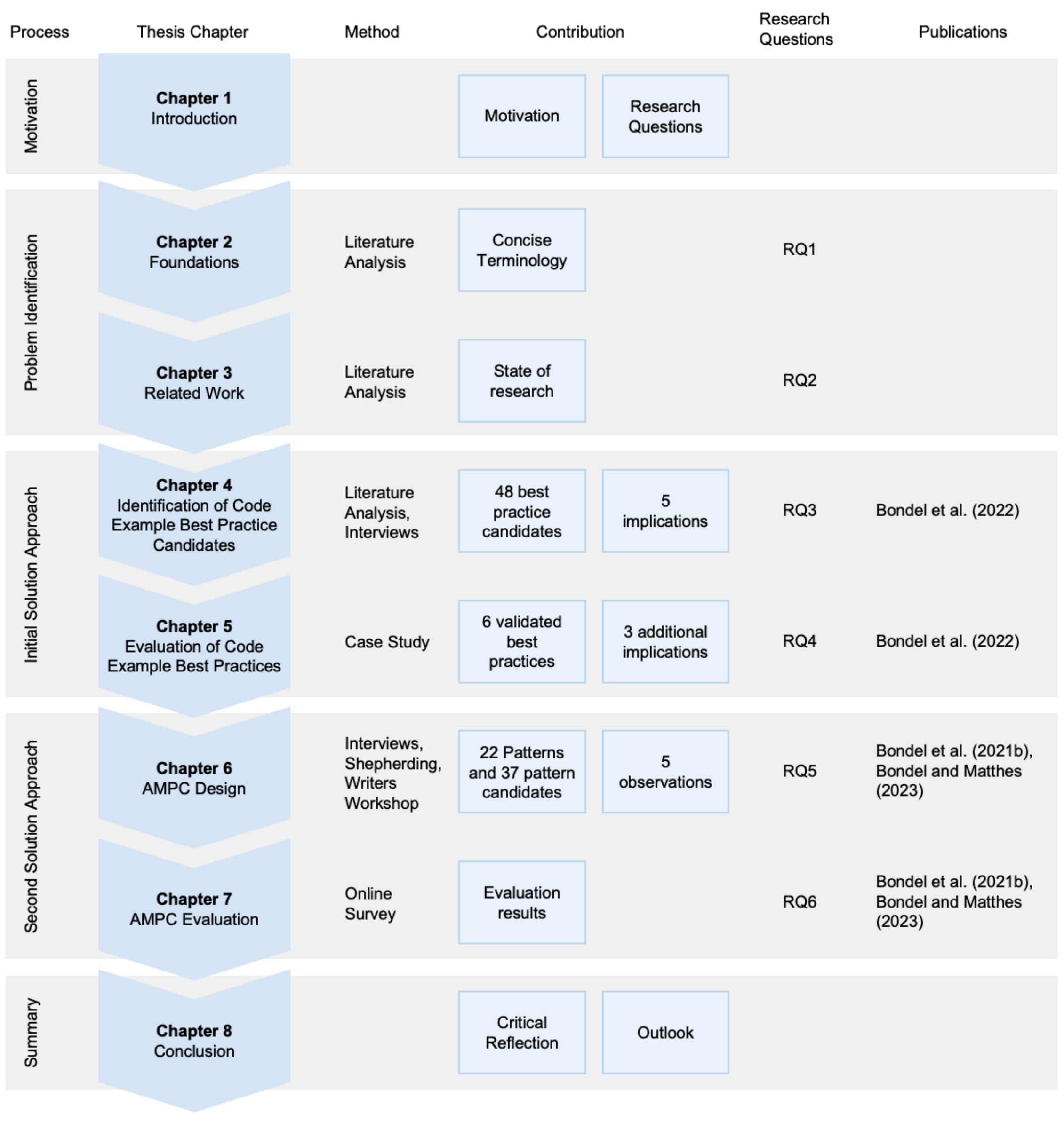
1.5. Citation Style and Conventions

This dissertation builds on prior publications and student theses advised by the dissertation's author. Appendix A presents an overview of these publications and student theses. We highlight direct quotes of previously published contents with an indentation and quotation marks. Also, we present the source of the content at the end of the quote (e.g., – (Bondel et al., 2022)). A quote can span several paragraphs. We use squared brackets and three dots to indicate removed text ([...]). Additions to the quoted text are enclosed in squared brackets ([This is an addition]).

However, we do not highlight cosmetic changes enhancing the readability without changing the meaning of the quoted text. Such changes comprise the update of references to the bibliography, tables, and figures to meet this dissertation's style. Also, we mapped IDs to the IDs used in this dissertation.

In addition, we apply the following conventions:

- We choose an inclusive and gender-neutral writing style. Hence, we use the singular, gender-neutral personal pronoun "they" and its derivative forms when referring to interview partners, evaluation participants, and other persons involved in the research endeavors presented in this dissertation.
- This dissertation contains some previously published figures. We indicate that a figure has been previously published by stating that it is 'adopted from' or 'adapted from' the respective source. We use 'adopted from' to indicate that we include the figure without changing its meaning from an external source. However, we might changed the color coding to match this dissertation's style. In comparison, we mark figures as 'adapted from' if we extended or evolved the figure, leading to changes in its meaning.
- All URLs presented in footnotes have last been accessed on 01.01.2024.
- We use a blue box without headings to present the overall research objective of this dissertation.
- We use blue boxes with headings to present definitions.
- We use *italic font* to highlight newly introduced terms.
- We use `monospaced font` to highlight pattern names.



Process	Thesis Chapter	Method	Contribution		Research Questions	Publications
Motivation	**Chapter 1** Introduction		Motivation	Research Questions		
Problem Identification	**Chapter 2** Foundations	Literature Analysis	Concise Terminology		RQ1	
	Chapter 3 Related Work	Literature Analysis	State of research		RQ2	
Initial Solution Approach	**Chapter 4** Identification of Code Example Best Practice Candidates	Literature Analysis, Interviews	48 best practice candidates	5 implications	RQ3	Bondel et al. (2022)
	Chapter 5 Evaluation of Code Example Best Practices	Case Study	6 validated best practices	3 additional implications	RQ4	Bondel et al. (2022)
Second Solution Approach	**Chapter 6** AMPC Design	Interviews, Shepherding, Writers Workshop	22 Patterns and 37 pattern candidates	5 observations	RQ5	Bondel et al. (2021b), Bondel and Matthes (2023)
	Chapter 7 AMPC Evaluation	Online Survey	Evaluation results		RQ6	Bondel et al. (2021b), Bondel and Matthes (2023)
Summary	**Chapter 8** Conclusion		Critical Reflection	Outlook		

Figure 1.1.: Structure of this dissertation including contributions and publications.

CHAPTER 2

Foundations

This chapter aims to present foundational knowledge and a concise terminology for the dissertation at hand. In addition, we motivate the need for API management, especially for established organizations in Europe. In the following, we discuss relevant foundations related to Web APIs, followed by an introduction to best practices and the concept of patterns in software engineering.

2.1. Web Application Programming Interfaces (Web APIs)

The concept of APIs dates back to the 1950's (Goldstine and Von Neumann, 1948; Wheeler, 1952) and is well known in Computer Science. According to Shnier (1996), APIs are:

> *"The calls, subroutines, or software interrupts that comprise a documented interface so that a (usually) higher-level program such as an application program can make use of the (usually) lower-level services and functions of another application, operating system, network operating system, driver, or other lower-level software program."* (Shnier, 1996, p.31)

Hence, APIs allow instantiating one of the most influential principles in software engineering, i.e., the principle of *information hiding* introduced by Parnas (1972). Information hiding aims to decrease dependencies between software modules so that changes in one module do not affect the functioning of other modules (Parnas, 1972). By enabling modularization, APIs have been proven to allow for reuse (Goldstine and Von Neumann, 1948; Robillard, 2009; Myers and Stylos, 2016; Watson et al., 2013), thereby reducing cognitive load for software developers (Wheeler, 1952) and thus enabling easier development, testing, and maintainability of software systems (Wheeler, 1952; Cotton and Greatorex, 1968). Also, the reuse of functionality allows developers to use existing mature implementations, improving software quality and speeding up development (Inzunza et al., 2018; Duala-Ekoko and Robillard, 2012). Furthermore, modularization enables collaboration and distribution of work on large software systems (Cotton and Greatorex, 1968; de Souza et al., 2004).

In addition, APIs enable data exchange and, thus, processes across organizational boundaries. In the 1980s and 1990s, proprietary networks like EDI were used for inter-firm communication and exchanging documents electronically (Christiaanse et al., 2004). These proprietary networks required high upfront, asset-specific investments leading to point-to-point integrations, inhibiting successful collaborations and the emergence of electronic markets (Christiaanse et al., 2004). However, in the 2000s, the internet emerged as an open, non-proprietary infrastructure (Christiaanse et al., 2004). As a result, Web APIs[1] established themselves as the de facto standard for data exchange between organizations.

The following presents the software artifacts and stakeholders involved in a data exchange via a Web API. Afterward, we dive into different Web API types and the most common architectural styles used for their implementation. Next, we present the categorization of Web APIs according to their accessibility to different audiences. Also, we present the API Economy as a new service ecosystem based on Web APIs and the benefits for participating organizations. However, we also show that the API provision of successful Web APIs is not distributed evenly across different types of organizations. Finally, we review the definition of Web API Management in research and practice.

2.1.1. Web API Software Artifacts

This section introduces the software artifacts involved in data exchange via a Web API as illustrated in the lower part of Fig. 2.1.

The basic assumption is that an API provider wants to make *business assets* accessible to API consumers (Jacobson et al., 2012). Business assets can be functionality or data (Fielding and Reschke, 2014a; Zimmermann et al., 2022), e.g., a payment process or user profiles (Bermbach and Wittern, 2016). A *backend* is the software component implementing the respective business asset. Therefore, we define a backend as follows:

Definition - Backend

A backend is a software component providing data and/or functionality that the API provider wants to make accessible to API consumers via a Web API.

Next, we characterize *Web APIs*. First, Web APIs rely on Web technologies and standards (Wittern et al., 2017). More specifically, Web APIs use HTTP as application protocol. Therefore, Web APIs make data and functionality accessible at a network addressable location, using a

[1] In this dissertation, we use the terms API and Web API interchangeably to refer to Web APIs if not explicitly stated otherwise.

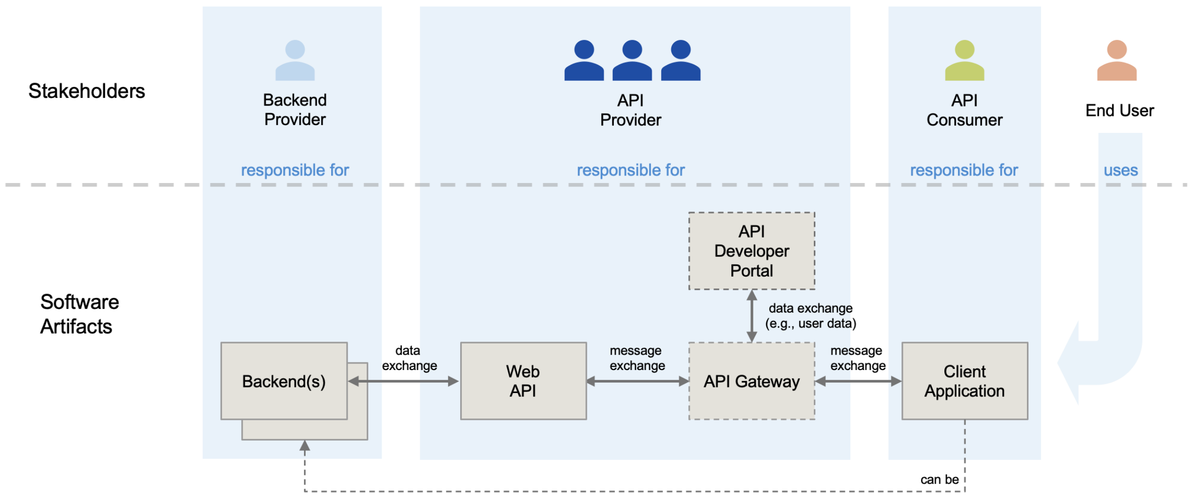


Figure 2.1.: Software artifacts and stakeholders involved in a data exchange via a Web API adapted from Bondel et al. (2021b).

Unified Resource Identifier (URI) (Fielding and Reschke, 2014a; Bermbach and Wittern, 2016). Moreover, Web APIs use messaging, i.e., request and response messages, to request and transfer data (Fielding and Reschke, 2014a). The request message expresses the action that it wants to perform on a resource using a HTTP method, e.g., GET, POST, PUT, PATCH, or DELETE (Fielding and Reschke, 2014b; Dusseault and Snell, 2010). The response message indicates the result of the backend processing a request using a predefined status code[2] (Fielding and Reschke, 2014b). The message formats for data exchange between the Web API and the client application are usually JavaScript Object Notation (JSON) or Extensible Markup Language (XML) (Wittern et al., 2017). HTTP relies on the Transmission Control Protocol (TCP) on the transport and the Internet Protocol (IP) on the network layer (Bermbach and Wittern, 2016). Moreover, API providers can increase security by establishing an end-to-end secured connection using HyperText Transfer Protocol Secure (HTTPS) (Fielding and Reschke, 2014a). HTTPS uses the Transport Layer Security (TLS) protocol on top of the TCP transport protocol to secure communication between communication partners using authentication and encryption (Rescorla, 2018).

As a result, a Web API is a machine- and human-readable interface (Santoro et al., 2019) that hides the implementation details of the backend (Bermbach and Wittern, 2016; Tan et al., 2016). Instead, it exposes a set of endpoints with which client applications can interact (Zimmermann et al., 2022). Therefore, a Web API is a *facade*, i.e., a (simplified) interface that allows clients to interact with a system (Gamma et al., 1995). A facade decouples the server implementation from the interface with which the client interacts (Gamma et al., 1995).

Moreover, on a conceptual level, a Web API presents a *contract* between a functionality providing and a functionality consuming component (Maalej and Robillard, 2013; De, 2017; Jacobson et al.,

[2]The Internet Assigned Numbers Authority (IANA) maintains a complete list of all specified status codes at `https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml`.

2012). The contract specifies possible interactions between the server and the client (Daigneau, 2011; Spichale, 2017), thus establishing a shared understanding (Zimmermann et al., 2022). More specifically, the contract specifies all endpoints and the respective operations the client can use to access the backend functionality or data (Zimmermann et al., 2022). The API provider defines the contract (Zimmermann et al., 2022). However, since the contract needs to meet the consumers' needs, the API provider should involve the API consumers in the design of the contract (Zimmermann et al., 2022).

Web APIs can be realized in different ways, including Remote Procedure Call (RPC) APIs, message APIs and resource APIs. We present these Web API types and typical architectural styles for their implementation in section 2.1.3.

Overall, Web APIs are a flexible and lightweight approach to make functionality and data accessible to internal and third-party applications (Santoro et al., 2019). Using messaging, Web APIs allow data exchange between applications running on different platforms, i.e., combinations of hardware, operating systems, software frameworks, and programming languages (Daigneau, 2011; Spichale, 2017). Consequently, different clients, comprising desktop, mobile, and web application clients, can access a backend's logic (Daigneau, 2011). In addition, Web APIs enable the independent evolution of backend and client applications (Kopecký et al., 2014; Daigneau, 2011; Spichale, 2017). Nevertheless, communication over the internet also introduces latency and the risk of network failures (Daigneau, 2011). Moreover, if a consumer integrates an API provided by a different organization, the consumer has no control over the Web APIs quality (Wittern et al., 2017).

As a result, in this dissertation, we define Web APIs as follows:

Definition - Web Application Programming Interface (Web API)

Web APIs are interfaces relying on Web technologies, specifically HTTP, allowing clients to access encapsulated functionality and data provided by a backend. Moreover, they act as a contract defining the rules of interaction between the backend and the client application.

Besides, two types of software platforms support API management, i.e., the API gateway and the API developer portal (De, 2017).

The *API gateway* is a runtime infrastructure component (Zimmermann et al., 2022) providing a scalable, reliable, and secure environment for API deployment and management (De, 2017; Medjaoui et al., 2018; Spichale, 2017). Hence, the API gateway is usually a reverse proxy sitting in front of the backend, listening for client requests and routing them to the backends (Medjaoui et al., 2018; Spichale, 2017) while hosting the API (De, 2017). Furthermore, it implements common API management capabilities like security (e.g., authorization, encryption, or identification of attacks), traffic management (e.g., consumption quota enforcement or spike arrest), interface translation, caching, service routing, service orchestration, and analytics (De, 2017; Spichale, 2017).

As a result, an API gateway reduces the cost of deploying APIs (Medjaoui et al., 2018, p.71). Nevertheless, an API gateway is another software component the API provider must manage. Hence, an API gateway requires additional design, testing, operations, and maintenance effort (Zimmermann et al., 2022; Richardson).

We define an API gateway as follows:

Definition - API Gateway

An API gateway is an infrastructure platform supporting Web API deployment and management.

The *API Developer Portal* is a consumer-facing web application providing all the information consumers need to discover and use a Web API. Therefore, Jacobson et al. (2012) refers to the API developer portal as *"[...] resource center for the API"* (Jacobson et al., 2012, p.123). On one hand, the API developer portal allows API providers to communicate and share knowledge about the Web API with consumers, e.g., through documentation, blogs, forums, monetization plans, and terms of use (Jacobson et al., 2012; De, 2017). On the other hand, the API developer portal offers self-service capabilities that allow consumers to use the Web APIs, e.g., user sign-up, app registration, API playground, Software Development Kits (SDKs), and contact to the support team (Jacobson et al., 2012; De, 2017).

However, the API provider has to ensure that the API developer portal looks alive and provides up-to-date information to attract API consumers (Jacobson et al., 2012).

We define an API developer portal as follows:

Definition - API Developer Portal

An API developer portal is a web application providing capabilities required to find and use a Web API to API consumers.

In summary, an API gateway provides an API runtime environment, and the API developer portal is a web application that both support API management (De, 2017). The API developer portal is usually a client of the API gateway (De, 2017). Hence, they exchange data, e.g., to ensure that a client making a request is registered. Usually, software vendors provide the API gateway and developer portal as commercial software tools that can be hosted on-premise or in public or private cloud environments (De, 2017). However, both the API gateway and the API developer portal are optional.

Finally, a *client application* integrates one or more Web APIs. The goal of a client application is to attract end users. Such client applications can be, for example, desktop or mobile browsers, mobile clients, or a web application (Richardson). A client application can also provide Web APIs, thus taking the role of a backend server in a different relation. We define client applications as follows:

Definition - Client Application

A client application is an application that integrates one or several Web APIs.

2.1.2. Web API Stakeholder Roles

Next, we define the stakeholders involved in interactions using Web APIs as presented in the upper part of Fig. 2.1.

First, the *backend provider* is a developer or team designing and maintaining the backend server. The backend provider owns the functionality or data exposed via a Web API (De, 2017). We define the backend provider as follows:

Definition - Backend Provider

A backend provider is a developer or team of developers responsible for the design, implementation, and maintenance of a backend.

> "The API provider is responsible for carrying out API management tasks. The API management comprises all tasks related to designing and maintaining the Web API, the API gateway, and the API developer portal. An API provider team includes business and technical roles, with business roles defining and pursuing business goals and technical roles aiming to ensure technical Key Performance Indicators (KPIs) (Medjaoui et al., 2018). The tasks of an API provider comprises all activities aimed towards realizing the goals defined by the API management lifecycle. In many cases, backend developers also design and maintain the respective Web API and thus occupy two roles."
>
> – (Bondel et al., 2021b)

An API provider comprises several more fine grained roles (Medjaoui et al., 2018). On the one hand, technical roles within an API provider team can be API developer, API architect, test and quality assurance engineer, and DevOps engineer (Medjaoui et al., 2018; Jacobson et al., 2012). On the other hand, business roles can comprise an API product manager, an API designer, technical writers, API evangelists, a community manager, and legal and marketing professionals (Medjaoui et al., 2018; Jacobson et al., 2012).

We define the API provider as follows:

Definition - API Provider

An API provider is a developer or team of developers responsible for the management of a Web API, including its design and operation. Also, the API provider is responsible for the design and operation of a potential API gateway and API developer portal.

Next, the *API consumer* is any person or group creating a client application integrating one or several Web APIs. Hence, in most cases, an API consumer is a professional software developer or team of developers. Therefore, the API consumers are also referred to as *third-party developers*[3], especially in the context of IS platform research, e.g., in Eaton et al. (2015); Islind et al. (2016); de Reuver et al. (2018). In a company setting, sometimes business roles are involved in the choice of a Web API, e.g., business owners or marketing employees (Medjaoui et al., 2018). However, an API consumer can also be any person using a mashup platform providing low-code or no-code capabilities to integrate Web APIs (Weiss and Gangadharan, 2010), e.g., Zapier[4] and IFTTT[5]. We define the API consumer as follows:

Definition - API Consumer

An API consumer is a person or team responsible for the design and operation of a client application that integrates a Web API.

Finally, the *end user* is the person using the client application (Medjaoui et al., 2018). The end user is interested in the client applications functionality from a non-technical perspective (Treiber et al., 2009). Hence, the expectations of the end user guide the functionality and Quality of Service (QoS) attributes of a Web API (Treiber et al., 2009; Medjaoui et al., 2018). We define an end user as follows:

Definition - End user

An end user is a user of a client application.

We find similar definitions of stakeholders in the literature.

First, Schmiedmayer (2022) defines three stakeholder groups. The *web service developers* design and maintain web services throughout their lifecycle. The *web service hosting providers* provide cloud infrastructure for hosting web services. Finally, the *web service clients* interact with a web service through its interface instantiated through a Web API.

In the context of Web services evolution, Treiber et al. (2009) introduces the *provider* perspective. The provider is responsible for the Web API. However, a *developer* implements the web service. A *service integrator* integrates several services into an application while the *user* uses the application. Finally, a *broker* manages a web service repository.

[3]In this dissertation, we use the terms *API consumer* and *third-party developer* interchangeably.
[4]`https://zapier.com/`
[5]`https://ifttt.com/`

Jacobson et al. (2012) distinguishes between an API provider, developers, and end users. The *API provider* is an organization that provides or wants to provide Web APIs. The *developers* use a Web API to create an application. Finally, the *end users* are the users of the new application.

Similarly, De (2017) distinguishes between an API provider, app developers, and end users. However, De (2017) differentiates between the *asset owner* and the *API provider*. While the asset owner and API provider are usually the same organization, it is also possible that the asset owner employs another organization as API provider to expose their assets and share the resulting revenue.

Gunturu (2022) distinguishes the *API provider*, *API consumer*, and *API end-user*. In addition, they mention the *API customer*, i.e., the stakeholder commercially paying for the API services provided by the API provider and consumer.

Finally, Zimmermann et al. (2022) identifies two communication partners, i.e., the *API provider* and the *API client*. Similarly, Mathijssen et al. (2020) identifies the *(API) developer and organization* and *(API) consumer* as main actors performing API management activities.

2.1.3. Web API Types and Architectural Styles

Based on HTTP, developers can realize three different types of Web APIs depending on the extent to which they use HTTP properties (Daigneau, 2011), as illustrated in Tab. 2.1. These three types are RPC APIs, message APIs, and resource APIs (Daigneau, 2011). In the following, we present each type of Web APIs and the most common architectural styles used for their realization.

2.1.3.1. Remote Procedure Call (RPC) APIs

A *Remote Procedure Call (RPC) API* allows clients to invoke a procedure on a remote server as if it were a local procedure (Daigneau, 2011). The RPC API defines operations that the client invokes through cross-network messages (Maleshkova et al., 2010). These cross-network messages have the same structure as in-memory object messages, i.e., consist of the remote procedure's name and required parameters that map to the remote procedure signature (Daigneau, 2011). Hence, RPC APIs use HTTP purely to transport messages (Daigneau, 2011; Kopecký et al., 2014) using one HTTP method (Maleshkova et al., 2010), usually GET.

Several RPC frameworks and technologies exist, e.g., gRPC[6], Apache Thrift[7], Distributed Component Object Mode (DCOM)[8], and Common Object Request Broker Architecture (CORBA)[9]. These RPC frameworks support the implementation of RPC APIs, e.g., by abstracting away network communications (Erl, 2008).

The advantages of RPC APIs are that they are easy to understand and implement (Daigneau,

[6]`https://grpc.io/`

[7]`https://thrift.apache.org/`

[8]Developers were able to download DCOM for Windows 95 as beta version in 1996, see `https://news.microsoft.com/1996/09/18/microsoft-releases-beta-version-of-dcom-for-windows-95/#Microsoft`.

[9]`https://www.corba.org/`

Table 2.1.: Overview of Web API styles and their characteristics (Daigneau, 2011).

Characteristics	**RPC API**	**Message API**	**Resource API**
Use of HTTP properties	Use of one HTTP method to transport messages	Use of one HTTP method to transport messages	Use of HTTP to define the semantics for Web API behavior
Message	Messages are coupled to the procedures signature	Messages are decoupled from the implementation	Messages are decoupled from the implementation
Common architectural styles	RPC API	Web services (SOAP/WSDL); GraphQL	REST

2011) and developers can use familiar procedure invocation approaches to call them (Vinoski, 2008). Also, existing frameworks enable developers to expose class methods simply by annotating methods with keywords (Daigneau, 2011). Moreover, providers can automatically generate RPC API documentation using XML Schema Definitions (XSDs) (Daigneau, 2011).

A downside of RPC APIs is that the client developers must understand data encoding and the interpretation of the remote procedure (Daigneau, 2011). In addition, interoperability between different remoting technologies, e.g., CORBA and DCOM, is restricted (Daigneau, 2011). While third-party products exist to bridge these technologies, they are complex and expensive (Daigneau, 2011).

Also, the procedure and the client are tightly coupled (Daigneau, 2011; Santoro et al., 2019). RPC APIs expect parameter lists with arguments that require an exact order, and changes to the order of arguments result in breaking changes (Daigneau, 2011). Hence, changes to the procedure require modifications of the client and, if existing, a proxy (Daigneau, 2011; Kopecký et al., 2014). As a result, RPC APIs are more suitable for data exchange within an organization and less for exchanging data with external business partners (Daigneau, 2011).

2.1.3.2. Message APIs

Message APIs are also called *document APIs* (Daigneau, 2011). Like RPC APIs, message APIs use the HTTP protocol to transport messages (Daigneau, 2011). However, compared to an RPC API, a message API does not derive the messages exchanged between consumer and provider from the procedure's signature (Daigneau, 2011). Instead, message APIs decouple the client from the procedure by defining self-descriptive messages, that the client sends to an URI (Daigneau, 2011).

However, the provider has to specify a separate message for each action applied to each endpoint of a message API (Daigneau, 2011). Such actions usually comprise Create, Read, Update, and Delete (CRUD) commands (Daigneau, 2011), e.g., "GetPublication" or "CreatePublication." In addition, the provider has to define two messages for each action, i.e., a request and a response message (Daigneau, 2011).This leads to a fast proliferation of pre-defined messages (Daigneau, 2011).

A common implementation of message APIs are Web Services based on SOAP[10] and Web Service Description Language (WSDL). Furthermore, GraphQL is a query language using platform self-descriptive messages sent over HTTP for data retrieval. Therefore, in the following, we describe Web Services based on SOAP and WSDL, and GraphQL as common architectural styles for implementing message APIs.

Web Services based on SOAP and WSDL

The term *Web services* has been defined in many different ways. Some publications understand Web services as any service a client can invoke over HTTP, including REST APIs, e.g., Daigneau (2011). However, we adopt the definition presented in Booth et al. (2004), stating that Web services use SOAP messages to interact and WSDL to specify their interface.

The World Wide Web Consortium (W3C) defines SOAP as:

> *"The formal set of conventions governing the format and processing rules of a SOAP message. These conventions include the interactions among SOAP nodes generating and accepting SOAP messages for the purpose of exchanging information along a SOAP message path."* (Gudgin et al., 2007).

Hence, SOAP is a protocol for exchanging information in decentralized, distributed environments aiming at simplicity and extensibility (Gudgin et al., 2007). It is independent of a particular programming model or implementation-specific semantics (Gudgin et al., 2007). The SOAP messaging framework is defined as a recommendation by the W3C[11]. It consists of the SOAP processing model, the SOAP extensibility model, the SOAP binding framework, and the SOAP messaging construct (Gudgin et al., 2007). The latest version of SOAP is SOAP 1.2 (Gudgin et al., 2007).

The Web Service Description Language (WSDL) is a description language for Web Services (Chinnici et al., 2007). WSDL defines a model and XML format documenting how clients can interact with a service (Chinnici et al., 2007). A WSDL file consists of nested components that specify valid input and output messages and data constraints (Chinnici et al., 2007).

Major software vendors led SOAP and WSDL standardization, e.g., IBM[12], Microsoft[13], Sun[14], and Oracle[15] (Kopecký et al., 2014). However, the standardization process suffered from political arguments between these vendors, leading to alternative standards and thus decreased interoperability (Kopecký et al., 2014). Also, developers dislike the complexity of SOAP and WSDL (Kopecký et al., 2014). A further downside of SOAP is that mobile platforms support it poorly (Kopecký et al., 2014). Nevertheless, nowadays, SOAP/WSDL-based APIs are still often

[10]Since the publication of SOAP version 1.2, SOAP is not an acronym of *Simple Object Access Protocol* anymore (Gudgin et al., 2007).
[11]`https://www.w3.org/`
[12]`https://www.ibm.com/de-de`
[13]`https://www.microsoft.com/de-de`
[14]Sun Microsystems was acquired by Oracle Corporation in 2010 as communicated in `https://www.oracle.com/corporate/pressrelease/oracle-buys-sun-042009.html`.
[15]`https://www.oracle.com/de/`

used for data exchange between enterprise applications within organizations (Kopecký et al., 2014; Tan et al., 2016).

GraphQL

GraphQL is a query language and server-side runtime aiming to meet the requirements of frontend engineers (The GraphQL Foundation, c). It was created by Facebook[16] (The GraphQL Foundation, b). A GraphQL API specifies the data accessible to clients (The GraphQL Foundation, d). The client sends a message to the GraphQL API detailing which data it wants to receive at field-level (The GraphQL Foundation, b). As a result, clients can specify the requested data and its structure, thus preventing over- or under-fetching (The GraphQL Foundation, b). However, GraphQL APIs are computationally expensive for servers (Santoro et al., 2019).

Clients usually interact with GraphQL over HTTP (The GraphQL Foundation, a), using the HTTP GET and POST methods (The GraphQL Foundation, e). Since GraphQL uses a hierarchical entity graph as a conceptual model (The GraphQL Foundation, e), a GraphQL API usually has one endpoint accessible via a Unified Resource Locator (URL) instead of a set of URLs for each resource (The GraphQL Foundation, a).

2.1.3.3. Resource APIs

Finally, *resource APIs* leverage HTTP as an application protocol that defines service behaviors (Daigneau, 2011). First, resource APIs provide access to data and functionality conceptualized as resources and accessible via a URI (Daigneau, 2011). Clients can manipulate these resources through representations sent as self-descriptive messages (Daigneau, 2011). Moreover, resource APIs leverage HTTP standard methods to send messages, e.g., PUT, GET, POST, and DELETE (Daigneau, 2011). Also, resource APIs use the standard status codes defined by HTTP (Daigneau, 2011).

A significant advantage of resource APIs is the reduced coupling between client and server. As HTTP separates the message from the service implementation, resource APIs can provide service access to many different types of clients, e.g., web browsers, mashups, and mobile applications (Daigneau, 2011). In addition, resource APIs are easier for developers to understand than message APIs (Tan et al., 2016).

However, publicly available resource APIs also carry security risks. For example, attackers can exploit resource APIs by analyzing the URI scheme and replacing URI segments to access data that should not be exposed (Daigneau, 2011).

While most resource APIs claim to conform to the REST paradigm, this is only true in some cases (Daigneau, 2011). Therefore, we introduce the REST paradigm in the following.

Representational State Transfer (REST)

In his dissertation (Fielding, 2000), Roy Fielding defines the *REpresentational State Transfer* (REST) architectural style that guides the architecture of an internet-scale, distributed hypermedia system, i.e., the World Wide Web (WWW). REST aims *"[...] to minimize latency and*

[16] `https://www.facebook.com/`

network communication while at the same time maximizing the independence and scalability of component implementations" (Fielding, 2000, p. 148).

To achieve this goal, Fielding (2000) defines six constraints. First, REST uses a client-server architecture to enable the separation of concerns between the interface and data storage. Secondly, servers do not store any context or state of a specific client, i.e., they are stateless. Also, server responses are cachable. Next, REST prescribes a layered system architecture style, i.e., two components communicating via an interface can only see the behavior of the immediate communication partner. Optionally, code-on-demand allows clients to download and execute code from servers, i.e., applets or scripts. Finally, a uniform interface defines a general, standardized form of transferring information between components. The uniform interface lets clients access data and functionality as resources addressable via a resource identifier. A client manipulates resources through representations. A server should also implement Hypermedia as the Engine of Application State (HATEOAS). The realization of HATEOAS means including hyperlinks to representations of other potentially interesting resources in response messages. Thus, a client can explore and navigate the resources of a REST API without the need for prior knowledge of the URI structure.

All six constraints must be satisfied to realize the benefits of the REST architectural style (Fielding, 2000). These benefits are performance, scalability, simplicity, modifiability, visibility, portability, and reliability of a distributed hypermedia system (Fielding, 2000).

On the downside, the REST architecture style is not an official standard (Rodríguez et al., 2016) and lacks clear guidelines on how to realize the architecture (Kotstein and Bogner, 2021; Salvadori and Siqueira, 2015). Moreover, even though REST guided the design of the HTTP protocol (Fielding, 2000), HTTP does not enforce compliance with REST. Thus, many Web APIs do not comply with all REST constraints (Palma et al., 2014, 2015; Rodríguez et al., 2016; Neumann et al., 2021; Renzel et al., 2012; Brabra et al., 2019; Petrillo et al., 2016; Belkhir et al., 2019; Haupt et al., 2017). As a result, many Web APIs falsely claim to conform to the REST architectural style, which led Roy Fielding to emphasize that not all HTTP-based interfaces automatically conform to REST (Fielding, 2008).

Several models categorize the compliance of interfaces with REST constraints (Algermissen, 2010; Salvadori and Siqueira, 2015; Santoro et al., 2019). The most renowned model is the *Richardson Maturity Model* (Richardson, 2009; Fowler, 2010), which distinguishes between four levels of maturity, each building on the previous level as illustrated in Fig. 2.2. The base level describes any API using HTTP but not conforming to any REST constraints. Instead, HTTP enables remote interaction mechanisms, usually using the POST method to pass arguments to a remote function, e.g., for XML-RPC or SOAP services. An HTTP API reaches the first maturity level as soon as it identifies resources using URIs. Thus, clients can interact with resources instead of calling one service endpoint. On the second level, an HTTP API additionally uses the HTTP methods, e.g., GET, POST, PUT, and DELETE, according to their standardized semantics. Moreover, the HTTP API uses error codes in its responses. Finally, an API qualifies as a RESTful API if it also implements HATEOAS. HATEOAS enables clients to move through representations easily, reduces coupling between client and server, and makes it easy for the server to change the URI scheme or to add new capabilities.

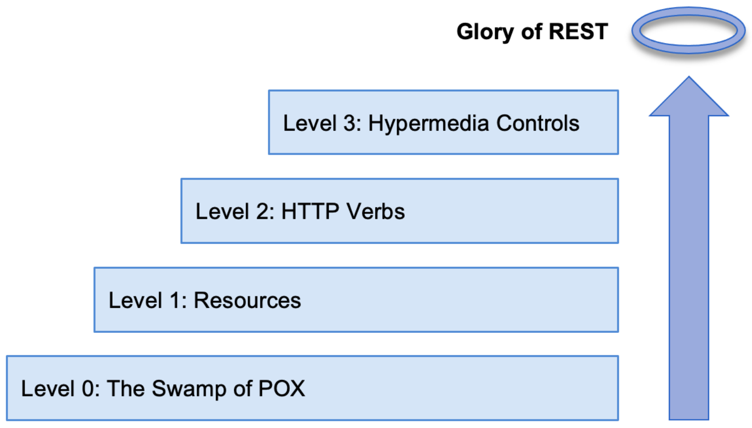


Figure 2.2.: The Richardson Maturity Model for categorizing the compliance of interfaces with REST constraints adopted from Fowler (2010).

Studies show that many Web APIs reach the second maturity level of the Richardson Maturity Model (Rodríguez et al., 2016; Neumann et al., 2021). Hence, these Web APIs comply with resource naming and make resource representations accessible using defined HTTP methods. In contrast, only a few APIs implement HATEOAS, consequently reaching the third level of the Richardson Maturity Model (Rodríguez et al., 2016). However, industry experts consider maturity level three unnecessary (Kotstein and Bogner, 2021). In addition, no agreed standards for designing and including hypermedia in response messages exist, making it difficult for providers and clients to reap its benefits (Rodríguez et al., 2016).

Nevertheless, REST APIs have become the most common approach for implementing Web APIs (Salvadori and Siqueira, 2015), especially in the areas of Web, mobile, cloud, and Internet of Things (IoT) applications (Tan et al., 2016). In comparison to SOAP APIs, REST APIs are more lightweight and easier to understand for developers (Tan et al., 2016). Also, REST APIs increase the probability of serendipitous reuse due to their uniform interface (Vinoski, 2008). As a result, REST APIs are the de facto standard for data exchange between organizations.

In summary, Web APIs comprise RPC APIs, message APIs, and resource APIs which are thus in scope of the dissertation at hand. However, due to the prevalence of resource APIs, we focus mostly on this type of Web API.

2.1.4. Web API Accessibility Categories

Web APIs can be categorized according to the audience that can access them. It is common to distinguish Web APIs into private and public APIs (De, 2017; Jacobson et al., 2012; Santoro et al., 2019; Doerrfeld et al., 2015). In the following, we first present the different categories of private APIs, followed by the description of public APIs. Finally, we define group, partner, and public APIs as in scope of this dissertation.

Private APIs are accessible to a restricted, authorized group of developers (De, 2017; Jacobson et al., 2012). Private APIs are prevalent compared to public APIs (Jacobson et al., 2012; De, 2017). We define private APIs as follows:

Definition - Private API

Private APIs are accessible to a predefined, restricted group of API consumers inside or outside the API provider organization.

Private APIs can be further categorized into internal, group, and partner APIs.

Internal API are available exclusively to API consumers that belong to the API provider's organization, i.e., staff of the API provider organization (Jacobson et al., 2012). Internal APIs enable internal teams to easily access data and functionality. Hence, these internal teams can create new products, realize mobile strategies or create internally used dashboards (Jacobson et al., 2012). Also, they enable the realization of Service-oriented Architectures (SOAs), i.e. the design of services encapsulating reusable units of logic (Erl, 2008). Hence, internal APIs reduce IT expenses and increase an organization's agility (Erl, 2008). We define internal APIs as follows:

Definition - Internal API

Internal APIs are private APIs accessible exclusively to API consumers inside the API provider organization.

Alternatively, in a group setting, a subsidiary can publish and manage *group APIs* accessible exclusively to other subsidiaries of the same group (Bondel et al., 2021b). The API-providing subsidiary is usually a central IT service provider. Compared to internal APIs, group APIs are accessible to other economically independent organizations with individual business goals. Nevertheless, the subsidiaries are all guided by the group's overall objective. We define group APIs as follows:

Definition - Group API

Group APIs are private APIs accessible to API consumers belonging to the same group as the API provider organization.

Moreover, an organization can make an API accessible to selected external partner organizations, i.e., create a *partner API* (Doerrfeld et al., 2015). Zimmermann et al. (2022) refers to this category of API as *community API*, emphasizing the aspired collaborative design of the API. Usually, individual contractual agreements between API provider and consumer govern the access to partner APIs (Jacobson et al., 2012; De, 2017). Organizations often use partner APIs to enable integration (De, 2017; Jacobson et al., 2012). We define define partner APIs as follows:

Definition - Partner API

Partner APIs are private APIs accessible to a restricted group of API consumers outside the API provider organization.

Finally, an API can be *public*, i.e., accessible to all developers (De, 2017). Therefore, public APIs are sometimes referred to as *open APIs* (Jacobson et al., 2012). API consumers can use a public API without negotiating a customized contract with the provider (Jacobson et al., 2012). Instead, the API provider creates general terms and conditions, monetization schemes, and other agreements to which the consumers must agree to access the API (Jacobson et al., 2012). Thus, the difference between a partner and public API is defined by the formality of the business arrangement (Jacobson et al., 2012) and its visibility. Public APIs aim to attract unlimited, unknown, and distributed API consumers outside the API provider organization (Zimmermann et al., 2022). If successful, they enable innovation, generate new business ideas, and maximize the utilization of the provider's existing assets (De, 2017). Also, public APIs can attract developers, thus supporting recruitment and public relations (Jacobson et al., 2012). On the other hand, API consumers use public APIs to experiment with new technologies, provide public services, engage in activism, or generate revenue (Jacobson et al., 2012). However, public APIs also introduce risks, e.g., strategic, legal, and security risks for providers (De, 2017; Jacobson et al., 2012; Doerrfeld et al., 2015). We define public APIs as follows:

Definition - Public API

Public APIs are accessible to API consumers outside the API provider organization that accept the predefined terms of use and other contractual agreements.

However, the categorization of APIs into internal, group, partner, and public APIs is not exclusive. For example, API provider organization internal teams can usually access partner or public APIs provided by their organization. Hence, an API can be accessible to several consumer groups (De, 2017). Also, the accessibility of Web APIs can change along their lifecycle, i.e., private APIs can become public, or public APIs can turn into partner APIs accessible only to specific partner organizations (Jacobson et al., 2012).

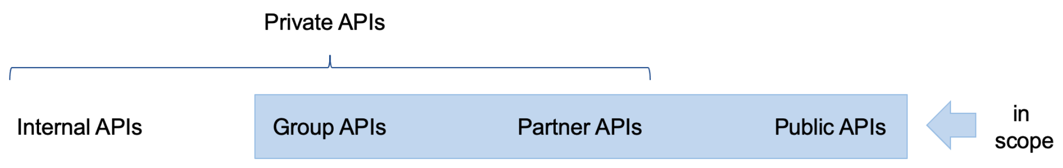


Figure 2.3.: Overview of different Web API accessibility categories (De, 2017; Jacobson et al., 2012; Doerrfeld et al., 2015), highlighting the scope of this dissertation.

The focus of the dissertation at hand is the management of APIs used across organizational boundaries. Hence, we focus on APIs with consumers external to the API provider organization, i.e., group, partner, and public APIs. We present an overview of the different accessibility categories and highlight the scope of this dissertation in Fig. 2.3.

2.2. API Economy

The proliferation of public APIs results in the emergence of the *API economy.* In the following, we introduce the concept of the API economy as a service ecosystem, discuss the benefits for API providers and consumers, and present the results of past ecosystem analyses.

As mentioned above, the API economy is a new service ecosystem resulting from firms creating new relations through Web APIs (Basole, 2019). Service ecosystems are *"[...] complex, evolving systems of highly interdependent human and non-human stakeholders who co-create value and are shaped by institutions and social norms"* (Basole, 2019, p. 479). Ecosystems are dynamic as entities appear and disappear, form and break relations, and the entities and their links change over time (Basole et al., 2018). Iyer and Subramaniam (2015) even argue that relations via Web APIs will become the most common type of relationship between organizations.

The participation of organizations in the API economy yields benefits for all actors. First, API providers can make data, functionality, or products accessible to third parties via public Web APIs (Weiss and Gangadharan, 2010; Basole, 2019). Moreover, they monetize this access using different business models like subscriptions, freemium, or pay-as-you-go (Evans and Basole, 2016; Basole, 2016, 2019). As a result, API providers benefit from new revenue streams (Evans and Basole, 2016; Basole, 2016, 2019). Also, public APIs enable scaling of operations and access to new markets (Basole et al., 2018).

On the other hand, instead of buying, Web API consumers rent functionality, thus enabling the XaaS paradigm (Basole, 2019). Renting instead of owning IT assets allows for more efficient IT management (Hagel III and Brown, 2001).

Alternatively, consumers can combine public Web APIs and potentially own data (Weiss and Gangadharan, 2010) to create new services for end users, i.e., *mashups* (Evans and Basole, 2016; Basole, 2019; Basole et al., 2018; Tan et al., 2016). Mashups range from simply integrating one Web API, e.g., integrating Google Maps[17] into a website, to combining different Web APIs and internal data into new products (Weiss and Gangadharan, 2010). Hence, Web APIs allow organizations to create innovation through service recombinations (Basole, 2019), e.g., consumers can use them to develop extended functionalities or apply the Web APIs in new contexts (Weiss and Gangadharan, 2010). At the same time, the API consumers often get exposure to the user base of the provider organization, e.g., if an API consumer creates a mashup based on a Google API, they get exposure to Google's user base (Weiss and Gangadharan, 2010).

As a result, in the API economy, both the API providers and consumers generate new value (Huhtamäki et al., 2017; Basole, 2016; Evans and Basole, 2016; Basole, 2019).

Moreover, ecosystem analysis reveals several trends in the API economy:

[17]`https://www.google.de/maps/preview`

- There has been a rapid growth of the public API ecosystem between 2005 and 2016 (Basole, 2019). Moreover, the number of mashups has grown even faster than the number of APIs, indicating that core services already exist (Basole, 2019; Weiss and Gangadharan, 2010).
- The API economy has a *"core-periphery structure"* (Basole, 2016, p.24). Thus, some APIs are at the center of the ecosystem, i.e., participate in many mashups, while others are located at the periphery, i.e., are integrated only in niche applications (Basole, 2016; Weiss and Gangadharan, 2010).
- Recently established digital organizations like Google[18] and Facebook[19] dominate the API economy (Evans and Basole, 2016; Basole, 2019; Weiss and Gangadharan, 2010). In comparison, few traditional firms successfully participate in the API economy as API providers (Evans and Basole, 2016).
- Also, the bigger the user base of an API, the more attractive it is for other users and, thus, the more consumers will use it in new mashups (Weiss and Gangadharan, 2010).
- Moreover, organizations located in major entrepreneurial regions, especially in Silicon Valley in the US, provide the majority of successful public APIs (Huhtamäki et al., 2017). Huhtamäki et al. (2017) assumes that co-location supports context setting and fine-tuning during Web API development.
- There are different categories of APIs, some being central and some peripheral to the API ecosystem (Basole, 2016). The most central categories are mapping, e-commerce, and social APIs (Basole, 2019). In these categories, competition is high (Basole, 2016). Hence, a provider needs to have a differentiating value proposition to compete in these categories (Basole, 2016).

Overall, participation in the API economy yields value for API providers and consumers. Nevertheless, analyses of the API economy ecosystem show that recently established digital organizations located in major US entrepreneurial regions dominate the provision of successful Web APIs. Moreover, if an organization decides to participate in the API economy as an API provider, careful design and management of APIs are required to succeed (Basole, 2019; Yoo et al., 2010).

2.3. APIs as Boundary Resources

In IS platform research, Web APIs are conceptualized as a kind of *(platform) boundary resource* and used to analyze platform dynamics. Therefore, we describe the concept of platformization, summarize research on boundary resource design, and derive implications for the design of Web APIs as boundary resources.

The basic assumption of platformization in IS is that embedding digital technology into physical products creates a *layered modular architecture* (Yoo et al., 2010). A layered modular architecture combines the modular architecture of physical products and the layered architecture of

[18] `https://about.google/`
[19] `https://www.facebook.com/`

digital products. Hence, the layered modular architecture allows designers to combine different modules on various loosely coupled, horizontal product layers, enabling innovation through *generativity* (Yoo et al., 2010). Generativity is *"[...] a technology's overall capacity to produce unprompted change driven by large, varied, and uncoordinated audiences"* (Zittrain, 2006, p. 1980). Due to the layered modular architecture, a product can simultaneously become a *digital product platform* (Yoo et al., 2010). A digital product platform enables third parties to innovate upon it by giving them access to core platform modules (Yoo et al., 2010; Ghazawneh and Henfridsson, 2013). However, platforms rely on consumers to create new applications (Ghazawneh and Henfridsson, 2010). Therefore, firms must attract large numbers of heterogeneous and unexpected actors to build components to be successful (Yoo et al., 2010; Ghazawneh and Henfridsson, 2010, 2013).

Platform providers can stimulate third-party engagement through the design of technical and social boundary resources (Yoo et al., 2010). According to Ghazawneh and Henfridsson (2013), boundary resources are *"[...] the software tools and regulations that serve as the interface for the arm's-length relationship between the platform owner and the application developer"* (Ghazawneh and Henfridsson, 2013, p. 174). Thus, boundary resources reduce direct coordination and communication efforts between the platform provider and third parties (Dal Bianco et al., 2014). Technical boundary resources are tools that support third parties in developing applications on top of the platform (Ghazawneh and Henfridsson, 2010). The most commonly recognized technical boundary resources are APIs and SDKs (Ghazawneh and Henfridsson, 2010; Dal Bianco et al., 2014). Social boundary resources restrict the interaction between third-party developers and the platform, e.g., developers' agreements (Ghazawneh and Henfridsson, 2010).

Therefore, boundary resources enable platform owners to transfer design capabilities to third-party developers to create innovation while at the same time maintaining control over the platform (Ghazawneh and Henfridsson, 2010), thus balancing *resourcing* and *securing* according to the *boundary resource model* (Ghazawneh and Henfridsson, 2013). Also, boundary resources can prevent a platform's exploitation through forking (Karhu et al., 2018). In contrast, Eaton et al. (2015) posit that boundary resource design is not predominantly driven by platform owners alone but by different actors within the innovation ecosystem. Such actors include regulators, the user base, partner organizations, and the public opinion in the blogosphere. These actors have different goals and levels of power to assert the design of boundary resources in their favor. As a result, *"[...] boundary resources of service systems enabled by digital technology are shaped and reshaped through distributed tuning, which involves cascading actions of accommodations and rejections of a network of heterogeneous actors and artifacts"* (Eaton et al., 2015, p. 217). Therefore, the design of boundary resources is strategically important (Yoo et al., 2010).

As explained above, boundary resources guide the *"arm's-length relationship"* (Ghazawneh and Henfridsson, 2013, p. 174) between platform providers and third-party developers. However, research on boundary resources mainly presents ex-post studies of large-scale successful cases like the iPhone iOS[20] or Android[21] (de Reuver et al., 2018). In contrast, Islind et al. (2016) analyze the role of boundary resources during the initial creation phase of a small-scale platform. The research shows that simply embedding knowledge into APIs and SDKs and expecting third-party

[20]https://www.apple.com/de/ios/ios-17/
[21]https://www.android.com/intl/de_de/

developers to create complementarities is insufficient. Instead, platform owners must co-create boundary resources to meet the skill set of the third-party developers. Also, intimate knowledge communication between the platform owner and the third-party developers is a requirement. This intimate and co-creative knowledge communication process is referred to as *fine-tuning*.

The presented research on boundary resources in the context of platformization yields several implications for Web API design. First, Web APIs are boundary resources that need active management to drive a platform's attractiveness for third-party developers while preventing its exploitation (Ghazawneh and Henfridsson, 2010). Hence, Web APIs must support many different types of applications, be easily understandable, and foster third-party creativity (Dal Bianco et al., 2014). Furthermore, as the concept of distributed tuning shows (Eaton et al., 2015), the platform owner cannot design Web APIs in isolation, but different actors influence its design. Moreover, in small-scale platform settings, platform owners should focus even more on co-designing Web APIs and knowledge communication (Islind et al., 2016). The relevance of Web API co-design and knowledge communication is reinforced by Huhtamäki et al. (2017), who assume that co-location supports context setting and fine-tuning during Web API development. Similarly, Zimmermann et al. (2022) and Spichale (2017) emphasize the importance of knowledge transfer and communication between the API provider and consumer to enable consumers to write API clients.

2.4. API Management

As argued above, API providers have to design and manage Web APIs intentionally to attract consumers. However, in research, no framework or overview providing a holistic view on API management and no comprehensive and widely accepted definition of API management exist (Mathijssen et al., 2020). Also, research on API management does currently not provide any theory and focuses on technological dimensions like API design and usage instead of social dimensions (Ofoeda et al., 2019). At the same time, IS research on platformization does not present clear knowledge on designing successful platforms (de Reuver et al., 2018). Hence, we present a working definition of API management and an API management lifecycle in this section.

2.4.1. API Management Definition

As a result of the analysis of 24 unique definitions of *API management*, Mathijssen et al. (2020) defines API management as follows:

> *"API Management is an activity that enables organizations to design, publish and deploy their APIs for (external) developers to consume. API Management capabilities such as controlling API lifecycles, access and authentication to APIs, monitoring, throttling and analyzing API usage, as well as providing security and documentation are often implemented through an integrated platform, which is supported by an API gateway."* (Mathijssen et al., 2020, p. 11)

This definition incorporates the claim that API providers realize API management largely through API platforms, i.e., API gateways and developer portals (De, 2017). Similarly, Spichale (2017) emphasizes the role of API management tooling.

In comparison, Medjaoui et al. (2018) describes API management as follows:

> *"[...] API management involves more than just governing the design, implementation, and release of APIs. It also includes the management of an API ecosystem, the distribution of decisions within your organization, and even the process of migrating existing APIs into your growing API landscape."* (Medjaoui et al., 2018, p. 2)

In addition, Spichale (2017) states that the goal of API management is to ensure that the API provider meets the API consumers needs along the API lifecycle. Hence, we combine these definitions and statements and the findings presented above, i.e., that API management is a function at the interface between an API provider organization and external consumers that requires collaboration and knowledge transfer. As a result, in this dissertation, we define API management as follows:

Definition - API Management

API management is an organizational function comprising all activities of an API provider (team) along the API lifecycle aiming to provide a successful Web API. In addition to the technical realization of (Web) APIs, API management requires collaboration and knowledge transfer with API consumers, backend providers, and other stakeholders.

In summary, API management is a discipline largely driven by practitioners. However, in the last ten years, research on Web API management increased (Mathijssen et al., 2020; Ofoeda et al., 2019).

Finally, we define an organization's endeavor of providing a Web API as an *API initiative*:

Definition - API Initiative

An API initiative denotes any planned or ongoing endeavor of an organization to provide a Web API to potential (external) API consumers.

2.4.2. API Management Lifecycle

Diving deeper into typical activities that API management comprises, we provide an overview of API management tasks along an API management lifecycle in the following. To the best of the author's knowledge, there is no API management lifecycle that research or practice agrees upon. Therefore, we derived an API management lifecycle from API management lifecycles put forward in practice that explicitly differentiate activities aimed towards external consumers and internal stakeholders (Bondel et al., 2021b). Hence, we reviewed the API management lifecycles presented by the major API management tool providers and API practitioner communities (Masse, 2019;

Axway Inc.; Doerrfeld et al., 2015) yielding the API management lifecycle illustrated in Fig. 2.4. An API keeps iterating through all stages shown in the lifecycle until the API provider deprecates it (Bondel et al., 2021b). Adapted from Bondel et al. (2021b), in the following, we describe each phase of the API management lifecycle and name typical activities of the API provider.

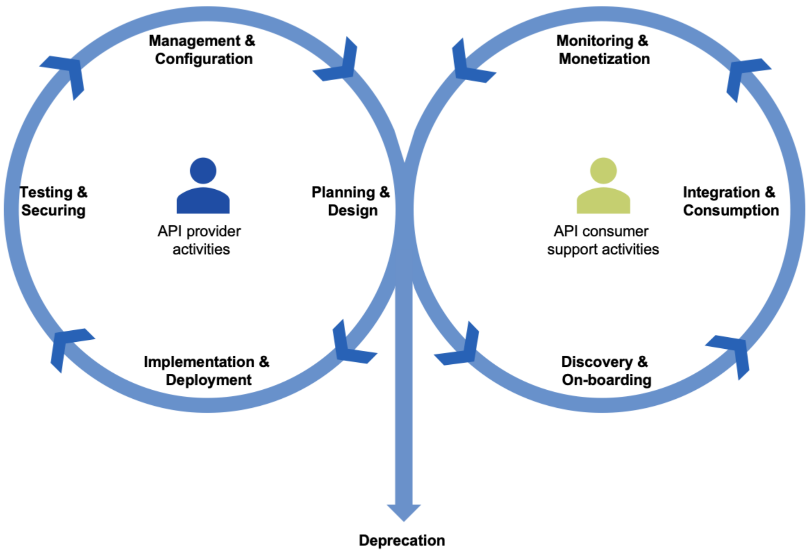

Management & Configuration
Testing & Securing
API provider activities
Planning & Design
Implementation & Deployment
Monitoring & Monetization
API consumer support activities
Integration & Consumption
Discovery & On-boarding
Deprecation

Figure 2.4.: Overview of an API management lifecycle derived from Masse (2019); Axway Inc.; Doerrfeld et al. (2015) and adopted from Bondel et al. (2021b).

- *Planning and Design:* Planning comprises the API provider defining the overall business objective of the API and the strategy to realize it (Doerrfeld et al., 2015; Masse, 2019). Activities in the planning phase are, e.g., the creation of the API's mission statement, a business plan, a usage forecast, and a marketing strategy (Doerrfeld et al., 2015). The API provider can design the Web API using a *contract-first* approach[22] (Masse, 2019). A contract-first approach means that the provider specifies the API design, comprising the definition of the schema, the base URLs, methods, request and response messages structures, authentication mechanisms, and payload formats, without implementing it yet (Axway Inc.). The API provider can share the designed contract with future consumers to gather early feedback (Masse, 2019). The API provider can also create a mock to simulate the behavior of the designed API (Masse, 2019; Axway Inc.).
- *Implementation and Deployment:* In this phase, the API provider develops the API, potentially using agile methods (Masse, 2019). The API provider can realize the implementation by aggregating and abstracting existing backend functionality or implementing new backend functionality. The use of *Continuous Integration (CI)* and *Continuous Deployment (CD)* pipelines supports the deployment of the API to the production environment (Masse, 2019; Axway Inc.). Also, commercial API gateways support the deployment of APIs (De, 2017).

[22]Also referred to as API-first approach (Axway Inc.).

- *Testing and Securing:* The testing and securing phase is intertwined with the implementation and deployment phase. Testing aims to validate that an API meets specified functional and non-functional requirements. Since we focus on Web APIs, i.e., APIs accessible via the public internet, security is paramount. Hence, security testing, including static analysis and vulnerability testing, is crucial (Masse, 2019). Also, the API provider has to implement security mechanisms, e.g., authentication, access control, spike arrests, and prevention of Distributed Denial of Service (DDoS) attacks (De, 2017).

- *Management and Configuration:* The manage and configure phase comprises minor changes to the API and bug-fixing (Doerrfeld et al., 2015). Moreover, the API provider configures API access, e.g., by defining policies, consumer quotas, or access rights (Axway Inc.).

- *Discovery and On-boarding:* The goal of the discovery and on-boarding phase is to promote the API (Masse, 2019) and engage with API consumers (Doerrfeld et al., 2015). Approaches to promoting an API can comprise *hackathons*, the definition of an *API evangelist* role, engagement in the blogosphere, or organizing physical events (Doerrfeld et al., 2015). Also, the API provider has to design and maintain the API developer portal, which provides documentation, access to support, and on-boarding facilities like API sandboxes, SDKs, and app registrations (Doerrfeld et al., 2015).

- *Integration and Consumption:* In this phase, the API consumers implement applications integrating the API (Masse, 2019). The API provider supports the API consumer. Also, the API provider can require the API consumer to submit the application to an application validation process (Masse, 2019).

- *Monitoring and Monetization:* The API provider monitors the API concerning its health, performance, and usage (Masse, 2019; Axway Inc.; Doerrfeld et al., 2015). Monitoring can identify improvement potentials feeding into the next design cycle (Masse, 2019; Doerrfeld et al., 2015). Also, monitoring allows billing API consumers according to their usage (Masse, 2019).

- *Deprecation:* After passing through one or several design cycles, the API provider can decide to retire an API, e.g., due to limited use, the use of outdated technologies, or opposing strategic or financial goals (Axway Inc.). Also, the API provider can decide to deprecate the old version of an API when publishing a new, breaking version (De, 2017). Before deprecating the API, the API provider should schedule a retirement plan, announce the plan, and create resources to ease the transition for existing API consumers (Doerrfeld et al., 2015).

2.5. Best Practices and Patterns

Managing Web APIs is critical and requires considering technical, social, organizational, and process-oriented aspects. A means to support organizations in becoming API providers is to present best practices for API management. Therefore, in this section, we present the concepts of *best practices*, *good practices*, and *patterns* in software engineering.

2.5.1. Best Practices

To the best of the author's knowledge, there is no generally agreed-upon definition of *best practices* in IS. However, Bretschneider et al. (2005) present a definition in the field of public administration research, stating that a best practice *"[...] implies that it is best when compared to any alternative course of action and that it is a practice designed to achieve some deliberative end"* (Bretschneider et al., 2005, p. 309).

Moreover, identifying best practices relies on several conditions according to Bretschneider et al. (2005). First, best practices are derived from existing cases, but the case base must be complete, i.e., identifying a best practice requires the analysis of all relevant cases. Secondly, the cases in scope must be comparable, e.g., regarding the context. Thirdly, a clear cause-and-effect relationship between a practice and an outcome must exist. An approach aiming to identify best practices that does not meet these requirements can identify "good" but not "best" practices (Bretschneider et al., 2005). However, researchers can rarely obtain completeness (Bretschneider et al., 2005). Also, in most design domains, the effects of a practice are multidimensional (Bretschneider et al., 2005).

In industry, the National Institute of Standards and Technology (NIST)[23] defines best practice as:

> *"A procedure that has been shown by research and experience to produce optimal results and that is established or proposed as a standard suitable for widespread adoption."* (NIST)

Also, Gartner[24] defines best practices as:

> *"[...] a group of tasks that optimizes the efficiency (cost and risk) or effectiveness (service level) of the business discipline or process to which it contributes. It must be implementable, replicable, transferable and adaptable across industries."* (Gartner)

Summarizing, all definitions emphasize that a best practice optimizes the specific outcome that an applicant wants to achieve. However, Bretschneider et al. (2005) requests that best practices are derived from complete and comparable case bases with clear cause-and-effect relationships between a practice and an outcome. Yet, this is not possible in the field of API management. First, creating a complete case base of all public, partner, and group cases is impossible. In addition, isolating a clear cause-and-effect relationship between practices and outcomes is impossible. Therefore, in this dissertation, we identify *good practices* according to Bretschneider et al. (2005). However, in industry, it is common to refer to practices that have been observed to support the achievement of a specific goal in several cases as *best practices*. We adopt this convention and refer to such practices as API management best practices in the remainder of this dissertation.

Hence, we define best practices as follows:

[23] `https://www.nist.gov/`
[24] `https://www.gartner.de/de`

Definition - Best Practice

A best practice is an activity or set of activities that optimizes a specific outcome in a defined context. Best practices are derived from existing cases.

In the following, we dive into a specific approach to documenting best practices, so-called *patterns*.

2.5.2. Patterns

Patterns are a means to capture best practices. The idea of patterns was first introduced by Christopher Alexander (Alexander, 1973; Alexander et al., 1977) in the domain of building architecture and town structures (Coplien, 1996). With object-oriented design's advent, patterns were adapted to software engineering in the mid-1980s (Coplien, 1996). Quickly, a fast-growing community evolved around the topic of software patterns (Coplien, 1996). Today, patterns have established themselves as a renowned tool for communicating best practices among software engineers. Moreover, patterns have spread from capturing best practices in structured domains like building architecture and software engineering to unstructured problem-solving domains, e.g., processes, organization, or training (Coplien, 1996).

In the following we first introduce the concepts of patterns, pattern languages, and pattern catalogs. Afterward, we present advantages and disadvantages of the pattern form. Finally, we introduce the major structural elements of patterns.

2.5.2.1. Patterns, Pattern Languages, and Pattern Catalogs

First and foremost, a *pattern* is a solution to a recurring problem in a specific context (Coplien, 1996; Meszaros and Doble, 1997; Buschmann et al., 1996). More precisely, patterns are a form of documentation (Coplien, 1996) that captures proven solutions that evolved and took shape over time, thus are born from experience (Gamma et al., 1995; Buschmann et al., 1996). According to Coplien (1996), good patterns should *"[...] capture important structures, practices, and techniques that are key competencies in a given field, but which are not yet widely known"* (Coplien, 1996, p. 4). However, patterns are not simply step-by-step instructions (Coplien, 1996). Instead, a pattern captures an abstraction, i.e., a *"[...] solution [that] has enough detail that the designer knows what to do, but it is general enough to address a broad context"* (Coplien, 1996, p. 5). Thus, a user has to adopt the concrete implementation of a pattern solution to their exact context (Coplien, 1996). Therefore, Alexander et al. (1977) states that a pattern describes a solution *"[...] in such a way that you can use this solution a million times over, without ever doing it the same way twice"* (Alexander et al., 1977, p. x).

Patterns are mined from experience (Gamma et al., 1995), i.e., document long-proven solutions and do not codify new ideas (Coplien, 1996). Hence, Coplien (1996) introduces the *rule of three*, i.e., a solution is only a pattern if a pattern designer observes it in at least three successful implementations.

We adopt the following definition of a pattern:

Definition - Pattern

A pattern documents a solution to a recurring problem in a specific context (Coplien, 1996; Meszaros and Doble, 1997). A pattern solution has to meet the rule of three, i.e., a solution is only a pattern if it was observed to successfully solve the respective problem in three different cases (Coplien, 1996).

Building on the concept of patterns, a *pattern language* is a set of related patterns that jointly resolve a higher-level problem (Coplien, 1996; Meszaros and Doble, 1997). At the same time, within a pattern languages context, each pattern addresses its own (sub-)problem, ideally allowing each pattern to be used independently (Meszaros and Doble, 1997). Patterns within a pattern language can relate in different ways. First, they can build on each other sequentially, i.e., a pattern solves a problem in a specific context, resulting in a new context that enables the application of further patterns (Coplien, 1996). Alternatively, a pattern language can comprise patterns that solve the same problem but in different contexts, thus requiring different solutions (Coplien, 1996; Meszaros and Doble, 1997). Also, patterns can generalize or specialize each other (Coplien, 1996; Meszaros and Doble, 1997). Transparent relationships between patterns allow readers to assess alternative solutions and identify complementary patterns (Meszaros and Doble, 1997). Hence, we define a pattern language as follows:

Definition - Pattern Language

A pattern language is a set of related patterns that fully resolve a higher-level problem in a specific domain (Coplien, 1996; Meszaros and Doble, 1997).

In comparison, a *pattern catalog* is a set of related patterns that do not fully resolve a higher-level problem in a specific domain (Coplien, 1996). Nevertheless, pattern catalogs have proven valuable, e.g., the pattern catalog put forward by Gamma et al. (1995) still is a standard reference book for object-oriented design nowadays. Hence, we define a pattern catalog as follows:

Definition - Pattern Catalog

A pattern catalog presents a set of related patterns that partially resolve a higher-level problem in a specific domain (Coplien, 1996).

In contrast to patterns, *anti-patterns* document common but destructive practices (Coplien, 1996). Since we do not present any anti-patterns as part of this dissertation, we do not present a definition.

2.5.2.2. Advantages and Challenges of Patterns

Patterns can realize several advantages. First an foremost, patterns provide practitioners with operational knowledge on current practices in a domain (Buckl et al., 2013, 2008; Khosroshahi et al., 2015). Practitioners already engaged in the respective domain can review patterns to benchmark their current practices with best practices used in other organizations (Buckl et al., 2008; Khosroshahi et al., 2015). Also, patterns and pattern languages enable knowledge dissemination in the respective domain (Buckl et al., 2013).

Next, designers can solve a problem step-by-step using complementary patterns of a pattern language (Alexander, 1973; Buckl et al., 2008). Consequently, the designers solve a problem incrementally instead of designing an overall solution to a complex problem upfront (Alexander, 1973, p.117ff.). However, patterns within a pattern language or catalog must rigorously adhere to a uniform terminology to enable easy integration of patterns (Buckl et al., 2013).

In the context of software patterns, patterns increased productivity by relieving developers from the repeated discovery of sound solutions (Coplien, 1996; Gamma et al., 1995). Thus, they prevent designers from making unnecessary errors and inspire solution designs (Coplien, 1996). In addition, software systems implementing patterns can be easier to understand for maintainers (Coplien, 1996).

Furthermore, patterns can provide a common taxonomy between stakeholders (Coplien, 1996; Buschmann et al., 1996). Developers can use patterns to better communicate with clients and each other (Coplien, 1996; Gamma et al., 1995).

Finally, patterns contribute to research by providing a basis for theory building in a specific domain (Buckl et al., 2013, 2008). Also, the documentation of pattern and pattern language changes over time allows researchers to derive knowledge on the evolution of a discipline (Buckl et al., 2013, 2008; Khosroshahi et al., 2015).

Still, patterns are only a tool supporting designers and do not automatically turn designers into experts (Coplien, 1996). Therefore, realizing these advantages is contingent on the pattern's quality and the designers' ability to adapt the pattern to the actual context (Coplien, 1996).

Moreover, pattern design requires experience. Hence, patterns can only be mined in domains with prior experience and not new domains (Coplien, 1996). In addition, pattern maintenance is laborious. Pattern designers should continuously refine and elicit new patterns from real-world cases (Buckl et al., 2013; Coplien, 1996). Finally, designers require training to reap the full potential of applying patterns within an organization (Coplien, 1996).

2.5.2.3. Pattern Elements

Patterns can have many different structures (Coplien, 1996; Meszaros and Doble, 1997). However, a pattern should comprise at least a *name*, a *context*, a *problem*, *forces*, and a *solution* (Meszaros and Doble, 1997). The relations between these structural elements are illustrated in Fig. 2.5.

The pattern *name* enables readers to find and refer to a pattern (Meszaros and Doble, 1997;

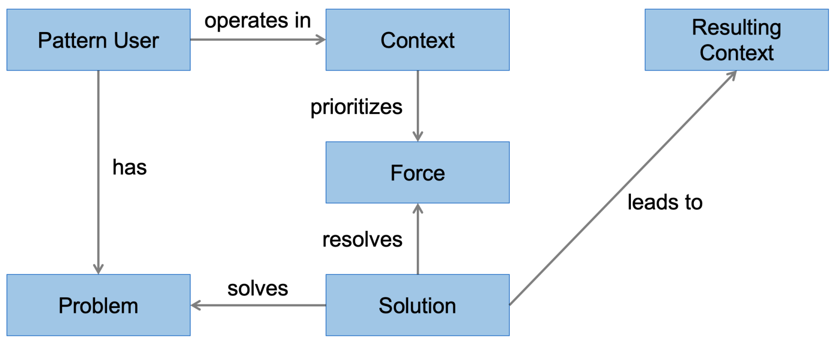


Figure 2.5.: Relations between structural elements of a pattern description adapted from Meszaros and Doble (1997).

Coplien, 1996). Also, the name allows referencing between patterns of the same or different pattern languages (Meszaros and Doble, 1997). Thus, the pattern author should choose a pattern name that is meaningful, understandable even out of context, and memorable (Meszaros and Doble, 1997). A pattern name often describes the solution of the pattern or uses a metaphor or analogy (Meszaros and Doble, 1997; Coplien, 1996). Optimally, the name of a pattern captures the content of a pattern in such a way that it creates a common taxonomy between users (Meszaros and Doble, 1997; Coplien, 1996). Additionally, an author can define alternative names for a pattern using aliases to address different readers' information needs Coplien (1996).

According to Meszaros and Doble (1997), the *context* describes *"The circumstance in which the problem [that] is being solved imposes constraints on the solution"* (Meszaros and Doble, 1997, p. 7). In simpler terms, a problem can arise in different (real-world) settings. Depending on these settings, a pattern solution can be more or less suitable. The context defines the characteristics of a setting that lead to differences in a pattern's applicability (Coplien, 1996; Meszaros and Doble, 1997). More precisely, the pattern context determines which forces a solution should prioritize (Coplien, 1996; Meszaros and Doble, 1997). Also, applying a pattern changes the context and thus creates a new context, i.e., the *resulting context* (Coplien, 1996), also denoted as *consequences* (Gamma et al., 1995). Further patterns can address the resulting context, hence linking patterns (Coplien, 1996).

The problem statement presents the concrete problem that a pattern solves (Coplien, 1996; Meszaros and Doble, 1997). The reader uses problem statements to identify relevant patterns (Coplien, 1996). A problem description can be a single question or a detailed narrative (Coplien, 1996). A problem should be *"context-free"*, meaning that a problem is independent of its context and can therefore arise in different settings (Meszaros and Doble, 1997).

A design problem is difficult to solve since there is usually no solution that satisfies all the applicant's goals ideally at once. Instead, a solution leads to trade-offs between different conflicting *forces* (Meszaros and Doble, 1997). For example, a developer implements an interface between software components. Optimally, the developer wants to implement an interface that allows for interoperability between different kinds of clients and simultaneously real-time communication. One approach to realize communication between software components is the implementation of a `Web API`. However, while a Web API enables interoperability for several types of clients due to its uniform interface, it also relies on communication over a network introducing latency (Daigneau, 2011). As a result, the developer has to decide how to resolve the trade-off between interoperability and performance, depending on the context. Hence, according to Meszaros and Doble (1997), forces are the *"[...] often contradictory considerations that must be taken into account when choosing a solution to a problem"* (Meszaros and Doble, 1997, p. 7). The forces of a pattern make the trade-offs of a problem explicit, thus demonstrating why a problem is hard to solve and helping the reader to grasp it fully Coplien (1996). More precisely, forces capture requirements, possible constraints, and desired properties of the solution (Buschmann et al., 1996).

Moreover, the pattern context prioritizes the forces a pattern should solve (Meszaros and Doble, 1997). Picking up on the example introduced above, the developer should prioritize interoperability if they implement an interface for different mobile clients. As a result, they can use the `Web API` pattern. However, if only one type of client uses the service and real-time performance is required, the `Web API` pattern is unsuitable. Thus, the solution focuses on resolving prioritized forces at the expense of others and potentially gives rise to new forces (Meszaros and Doble, 1997). A clear understanding of the forces allows the reader to judge the suitability of a pattern and supports adapting the pattern to the readers' needs (Coplien, 1996). This is the reason Coplien (1996) states that *"[...] forces are the focus of a pattern"* (Coplien, 1996, p. 9).

The pattern *solution* solves a problem and resolves the conflict between forces as prioritized by the context (Coplien, 1996; Meszaros and Doble, 1997). A solution can resolve some forces completely, others partially, and others not at all (Meszaros and Doble, 1997). The pattern author has to balance the solution description to provide enough detail for the user to apply it and, simultaneously, be generic enough to be applicable to a broader context (Coplien, 1996; Gamma et al., 1995). Furthermore, the pattern author should explicitly refer to the resolved forces (Meszaros and Doble, 1997).

Further structural elements of patterns can be *indications* or *symptoms* (Meszaros and Doble, 1997), *intent* (Coplien, 1996), a *rationale* (Meszaros and Doble, 1997), a *sketch* (Coplien, 1996), *examples* or *known uses* (Meszaros and Doble, 1997), *code samples* (Meszaros and Doble, 1997), *related patterns* (Meszaros and Doble, 1997), a *resulting context* or *consequences* (Coplien, 1996; Meszaros and Doble, 1997; Gamma et al., 1995), or *acknowledgments* (Meszaros and Doble, 1997).

2.6. Summary

This chapter aimed to provide foundational knowledge and to establish the taxonomy used in the dissertation at hand.

First, we defined the software artifacts and stakeholders involved in providing and using a Web API. The software artifacts comprise the backend, the Web API, and the client application. In addition, API management platforms, i.e., an API gateway and API developer portal, can realize API management capabilities (De, 2017; Spichale, 2017). The involved stakeholders are the backend provider, API provider, API consumer, and end user.

An API provider can realize Web APIs in different ways. Web API types are RPC APIs, message APIs, and resource APIs. Message APIs can be implemented as Web Services based on SOAP and WSDL or use the query language GraphQL, while resource APIs usually aim to achieve REST compliance. Web services based on SOAP/WSDL are still often used for the integration of internal enterprise applications (Kopecký et al., 2014; Tan et al., 2016), while REST APIs dominate the data exchange between organizations (Salvadori and Siqueira, 2015; Tan et al., 2016). In addition, GraphQL APIs are gaining traction for inter-organizational data exchange.

Next, we categorized Web APIs according to the audience that can access them. It is common to distinguish Web APIs into private and public APIs (De, 2017; Jacobson et al., 2012; Santoro et al., 2019; Doerrfeld et al., 2015). Private APIs are accessible to a restricted audience, i.e., internal APIs are accessible only to staff internal to the API provider organization, group APIs are accessible to subsidiaries belonging to the same group (Bondel et al., 2021b), and partner APIs are accessible to selected partner organizations (De, 2017). In comparison, public APIs are accessible to all developers and organizations agreeing to an API's terms of use and other standard contractual agreements (Jacobson et al., 2012). In this dissertation, we focus on Web APIs with the API provider and API consumer belonging to different organizations, i.e., public, partner, and group Web APIs.

The provision and use of public, partner, and group Web APIs yields value for organizations, thus giving rise to a new service ecosystem, the so-called *API economy* (Huhtamäki et al., 2017; Basole, 2016; Evans and Basole, 2016; Basole, 2019). On the one hand, organizations taking on the API provider role can generate additional revenue streams (Evans and Basole, 2016; Basole, 2016, 2019). On the other hand, API consumers profit from more efficient IT management by sourcing IT capabilities as XaaS (Basole, 2019). Also, API consumers can recombine Web APIs to create new services, i.e., mashups, for end users (Weiss and Gangadharan, 2010; Evans and Basole, 2016; Basole, 2019; Basole et al., 2018; Tan et al., 2016). Nevertheless, analyses of the API economy ecosystem show that recently established digital organizations located in major US entrepreneurial regions dominate the provision of successful Web APIs compared to established organizations from traditional industry sectors (Evans and Basole, 2016; Basole, 2019; Weiss and Gangadharan, 2010; Huhtamäki et al., 2017).

Taking the lens of IS platform research, Web APIs are conceptualized as a kind of *(platform) boundary resource* (Yoo et al., 2010). Boundary resources are software tools and contractual agreements at the interface between API provider and consumer that govern their interactions (Ghazawneh and Henfridsson, 2013). Hence, Web APIs are strategically important resources (Yoo et al., 2010). Research on boundary resources yields several implications for Web APIs. First, Web APIs need active management to attract API consumers and prevent exploitation of provided backend functionality and data (Ghazawneh and Henfridsson, 2010). Secondly, API providers control Web API design, but different actors influence it (Eaton et al., 2015). Also, Web API co-design increases the API's attractiveness for API consumers, especially in small-scale settings (Islind et al., 2016; Huhtamäki et al., 2017). Furthermore, the API provider needs to transfer knowledge to consumers to enable the use of Web APIs (Islind et al., 2016; Zimmermann et al., 2022; Spichale, 2017). Finally, previous research on platformization does not present clear knowledge on designing successful platforms (de Reuver et al., 2018).

Since Web APIs require active management to attract consumers, we present a definition of

Web API management. Reviewing and aggregating different definitions of API management, we define API management as an organizational function comprising all API provider activities along an API lifecycle aiming to provide a successful Web API. Moreover, we make explicit that API management comprises technical as well as social aspects, i.e., the collaboration and knowledge transfer with different stakeholders. These stakeholders include the API consumer but also stakeholders internal to the API provider organization. Also, we present an API management lifecycle providing an overview of API management activities.

Finally, best practices describe activities that optimize a specific outcome in a defined context. These best practices are derived from existing cases. A common approach to document best practices in IS are design patterns. A pattern captures a solution to a recurring problem in a specific context (Coplien, 1996; Meszaros and Doble, 1997; Buschmann et al., 1996). Moreover, a *pattern language* is a system of related patterns that together solve a problem (Coplien, 1996; Meszaros and Doble, 1997). In comparison, a *pattern catalog* denotes a set of patterns that partially solves a higher-level problem (Coplien, 1996). Patterns make operational knowledge accessible (Buckl et al., 2013, 2008; Khosroshahi et al., 2015), allow for incremental problem solutions (Alexander, 1973; Buckl et al., 2008), provide a common taxonomy (Coplien, 1996; Buschmann et al., 1996), and create a basis for theory building in research (Buckl et al., 2013, 2008; Khosroshahi et al., 2015).

CHAPTER 3

Related Work

This chapter aims to review existing patterns for API management. However, to the best of the author's knowledge, no pattern language or catalog explicitly concerned with API management exists. Instead, we review pattern languages and catalogs focusing on the design of APIs or interfaces and interactions between distributed software components. Hence, we analyze API design, service design, middleware design, object-oriented software design, and software architecture pattern languages and catalogs to identify API management patterns. We categorize the identified API management patterns along the API management lifecycle presented in Fig. 2.4. However, we only reviewed pattern collections containing the minimum structural requirements for documenting patterns, i.e., pattern collections with pattern descriptions that have at least a name, and detail the context, problem, forces, and solution (Meszaros and Doble, 1997).

3.1. API Design Patterns

First, we present pattern languages and collections focusing on the design of APIs. We review these pattern languages and catalogs to identify API management patterns.

Patterns for API Design by Zimmermann et al. (2022)

The *Patterns for API Design* (Zimmermann et al., 2022; Zimmermann et al.) has previously been known as *Microservice API Patterns (MAP)* and evolved from several scientific articles (Zimmermann et al., 2020a,b,c; Lübke et al., 2019; Stocker et al., 2018; Zdun et al., 2018; Zimmermann et al., 2017). The *Patterns for API Design* form an extensive pattern language focusing on the structure, evolution, management, and description of request and response messages exchanged between provider and client endpoints (Zimmermann et al., 2022). The authors derive the patterns from public Web APIs and their own experiences with software development and integration projects (Zimmermann et al., 2017; Zdun et al., 2018; Stocker et al., 2018; Zimmermann et al., 2020b). The pattern language comprises 44 patterns[1] organized into five categories (Zimmermann et al., 2022). These categories are 'foundation patterns', 'responsibility patterns', 'structure patterns', 'quality patterns', and 'evolution patterns'.

Discussion

Patterns for API Design (Zimmermann et al., 2022; Zimmermann et al.) emphasizes the structure, evolution, management, and description of request and response messages exchanged between provider and client endpoints. Hence, most of the 'foundation patterns', 'responsibility patterns', 'structure patterns', 'quality patterns', and 'evolution patterns' guide the design of an API in the *planning & design* phase of API management. In the following, we detail patterns that additionally support other API management lifecycle phases as described in Fig. 2.4.

Within the 'foundation patterns', the pattern `API Description` supports the design of new Web APIs, e.g., by presenting specifications to future consumers to get feedback. Moreover, the API developer portal should publish the `API Description` to enable *discovery & onboarding* and *integration & consumption*. In addition, within the 'structure patterns', the `API Key` enables *testing & securing* while the `Error Report` supports *monitoring & monetization*. Also, the `Pricing Plan` within the 'quality patterns' category is part of designing an API strategy. In contrast, the `Rate Limit` enables *testing & security* and enforces *monitoring & monetization*. Also, the `Service Level Agreement` should guide *implementation & deployment* and plays an essential role during *integration & onboarding*. Finally, the 'evolution patterns' initiate a new *planning & design* phase or guide the *management & configuration*. Also, `Aggressive Obsolesce` leads to *deprecation* of APIs.

API Design Patterns according to Geewax (2021)

Geewax (2021) presents patterns for Web APIs, i.e., APIs used remotely over a network. The patterns capture the design and structure of such APIs and are described consistently. Geewax (2021) presents 25 patterns structured into the categories 'fundamentals', 'resource relationships', 'collective operations', and 'safety and security'.

Discussion

The *API design patterns* put forward by Geewax (2021) document the technical implementation of Web APIs. Thus, they support the *planning & design* and the *implementation & deployment* phase of the API management lifecycle presented in Fig. 2.4. In addition, the pattern `Versioning and Compatibility` provides strategies for implementing changes that trigger a new API man-

[1]In comparison to the book (Zimmermann et al., 2022), the website (Zimmermann et al.) publishes the additional pattern `Eternal Lifetime Guarantee`.

agement lifecycle or support the *management & configuration* of a Web API. Furthermore, `Request Authentication` plays a vital role in the phase *testing & security*.

Patterns for RESTful Conversations by Pautasso et al. (2016)

Pautasso et al. (2016) presents a pattern language for RESTful conversations drawing on previous publications (Pautasso and Wilde, 2010; Haupt et al., 2015). A RESTful conversation assumes a client interacts with an API that adheres to the REST architectural style according to Fielding (2000). The patterns capture recurring HTTP request-response interactions between the client and the REST API. Moreover, the patterns support API designers and client developers in addressing non-functional requirements like security, reliability, and scalability during API design and consumption. Overall, Pautasso et al. (2016) describes ten RESTful conversation patterns structured along the CRUD operations. All patterns have a consistent form comprising a visualization.

Discussion

The *RESTful conversation patterns* (Pautasso et al., 2016) describe sequences of interactions between a REST API and a client. These interactions aim to realize different goals, including reliable creation, updating, discovery, and resource protection. These patterns play an essential role during the design of an API, hence supporting the *planning & design* phase and guiding the *implementation & deployment* phase.

Control-Flow Patterns for Decentralized RESTful Service Composition by Bellido et al. (2013)

Bellido et al. (2013) present a set of control-flow patterns for service composition compliant with REST and using HTTP as application protocol. Using WSDL/SOAP-based service control-flow patterns as a starting point, Bellido et al. (2013) derived stateless compositions of RESTful services through callbacks and redirections. The patterns are `sequence`, `unordered sequence`, `alternative`, `exclusive choice`, `iteration`, `structured loop`, `parallel split - synchronization`, `structured discriminator`, `structured partial join`, `local synchronization merge`. Bellido et al. (2013) describe each pattern, but discuss the context, problem, and consequences on a high level for all patterns together. An implementation and evaluation of QoS attributes shows that these control-flow patterns improve the availability and throughput of service compositions while the response time remains stable.

Discussion

The *control-flow patterns for decentralized RESTful service composition* by Bellido et al. (2013) document four types of control-flows with different variations. These patterns can support the design of Web APIs during the API management lifecycle phase *planning & design* as visualized in Fig. 2.4.

3.2. Service Design Patterns

A service is an independently deployable software component with concise functionality and a database (Richardson). Services expose reusable business functions via defined interfaces (Daigneau, 2011). Hence, the management of APIs is integral to service management. In the following, we identify API management patterns in pattern languages and catalogs concerned with services in the context of service design, SOA, and microservices architectures.

Service Design Patterns by Daigneau (2011)

In Daigneau (2011), the author presents web service design patterns for professional enterprise architects, solution architects, and developers. According to the author *"**Web services** provide the means to integrate disparate systems and expose reusable business functions over HTTP. They either leverage HTTP as a simple transport over which data is carried (e.g., SOAP/WSDL services) or use it as a complete application protocol that defines the semantics for service behavior (e.g., RESTful services)"* (Daigneau, 2011, p. 2). Moreover, web services use XML, JSON, or common media-type data-exchange standards. Hence, web services rely on ubiquitous, interoperable standards independent of the underlying technologies. The service design patterns are derived from real-life experiences and have a consistent form influenced by the pattern structures of Alexander (1973) and Hohpe and Woolf (2003).

Overall, Daigneau (2011) describes 25 patterns across six categories. The six categories are 'Web Service API styles', 'Client-Service Interactions', 'Request and Response Management', 'Web Service Implementation Styles', 'Web Service Infrastructures', and 'Web Service Evolution'.

Discussion

Daigneau (2011) presents 25 patterns for web service design, which mainly support the *planning & design* phase and guide the *implementation & deployment* phase of the API management lifecycle presented in Fig. 2.4. The pattern `Consumer-driven contracts` is further relevant during the *testing & securing* phase.

Furthermore, Daigneau (2011) introduces a definition of *web services* that we adopt for the term *Web API* in this dissertation and the AMPC (Bondel and Matthes, 2023).

SOA Design Patterns by Erl (2008)

Erl (2008) presents a service-oriented architecture (SOA) design pattern language that captures real-world solutions to SOA realization challenges. The target audience of the SOA design patterns are IT practitioners involved in SOA implementations. The patterns have been influenced by Christopher Alexander (Alexander, 1973; Alexander et al., 1977), Hohpe and Woolf (2003), Fowler (2003), Gamma et al. (1995) and the POSA books (Buschmann et al., 1996; Schmidt et al., 2000; Kircher and Jain, 2004; Buschmann et al., 2007b,a). The patterns relate to each other and have a consistent form. Overall, 85 SOA patterns are published in Erl (2008). On the highest level, the book structures the patterns into ‘service inventory design patterns’, ‘service design patterns’, and ‘service composition design patterns’.

Discussion

Erl (2008) presents patterns for SOA with a focus on service delivery within an organization’s boundaries. Nevertheless, some SOA patterns can support the design of Web APIs used across organizational borders. First, the ‘service design patterns’ and the ‘service composition design patterns’ support the *planning & design* and the *implementation & deployment* phase of API management. Moreover, the ‘service security patterns’ and the ‘service interaction security patterns’ can aid during the *testing & securing* phase. The pattern `Compatible Change`, `Version Identification`, `Termination Notification`, and `Service Refactoring` can guide the evolution of Web API and are thus relevant during the initiation of a new API management lifecycle or support the *management & configuration.*

SOA Patterns by Rotem-Gal-Oz (2012)

Rotem-Gal-Oz (2012) present technology-neutral architecture patterns for applications adhering to the SOA architectural style. The patterns are presented in a consistent form inspired by Alexander et al. (1977). Overall, Rotem-Gal-Oz (2012) documents 26 patterns covering the ‘structural’, ‘performance, scalability and availability’, ‘security and manageability’, ‘message exchange’, ‘service consumer’, and ‘service integration’ patterns.

Discussion

Rotem-Gal-Oz (2012) presents architectural patterns for implementing a SOA. The ‘foundational structural patterns’, ‘patterns for performance, scalability, and availability’, ‘message exchange patterns’, ‘service consumer patterns’, and ‘service integration patterns’ can guide the *planning & design* and *implementation & deployment* phase of the API management lifecycle presented in Fig. 2.4. In addition, the ‘security and manageability patterns’ support the *texting & securing* phase. Also, the `Service Monitor pattern` can aid during *monitoring & monetization.*

Finally, the pattern `service host pattern` describes the concept of a API gateway.

Microservices Patterns by Richardson (2019)

In Richardson (2019), the author presents a microservices architecture language, i.e., patterns that enable the reader to design applications implementing a microservices architecture. The patterns are published online (Richardson) and the pattern book Richardson (2019) focuses on providing additional details and linking the microservices patterns. The patterns have a consistent form. Overall, the author publishes 52 patterns across ten groups. The first group comprises ‘patterns for decision-making’ supporting the choice of implementing a monolithic or microservices architecture. The other groups are patterns supporting the realization of a 'microservices architecture comprising decomposition', 'communication', 'data consistency', 'microservices querying', 'deployment', 'observability', 'automated testing', 'cross-cutting concerns', and 'security' patterns. In addition to the microservices patterns, Richardson also describes seven microservice adoption anti-patterns.

Discussion

The goal of Richardson (2019) and Richardson is to enable readers to implement applications successfully using microservices, i.e., decomposing and implementing systems into microservices on the infrastructure and application levels. The 'communication patterns' support first and foremost the *planning & design* phase of the API management lifecycle as presented in Fig. 2.4. Generally, the 'testing patterns', the 'security patterns', and the `Circuit Breaker` guide the *testing & securing* phase. The 'observability patterns' can aid during *testing & securing* but also enable *monitoring & monetization.*

Moreover, Richardson (2019) and Richardson present the very influential `API gateway` pattern. An `API gateway` is the single entry point for client requests accessing a microservice. The `API gateway` forwards client requests to the respective APIs. Furthermore, `API gateways` realize API composition and authentication. For example, Montesi and Weber (2016), Müssig et al. (2017), and Akbulut and Perros (2019) all extend the `API gateway` pattern to implement API management functionalities. These findings align with the statements of Mathijssen et al. (2020) and De (2017) claiming that API management platforms play an essential role in realizing API management functionalities.

Microservices Patterns by Newman (2019)

Sam Newman captures 22 microservices-related patterns in a book (Newman, 2019) and on a website (Newman). These microservices-related patterns comprise 'backend management', 'tenancy', 'responsibility', 'migration from monolithic to microservices', and 'database decomposition' patterns.

Discussion

The *Microservices Patterns* (Newman, 2019; Newman) focus on the transition from monolithic systems to microservices. Hence, the microservices patterns focus primarily on service architectures and the responsibility for service changes. The `Backends for Frontends` can be employed during the *planning & design* phase of the API management lifecycle presented in 2.4. The patterns `Single Tenancy`, `Multi-Tenancy`, and `Hybrid Tenancy` can be relevant during the *implementation & deployment* of a Web API. Also, `Open Service Ownership`, `Temporary Service Ownership`, and `Roving Custodians` can guide the *management & configuration* phase.

3.3. Middleware Design Patterns

Middleware introduces an additional layer between distributed applications communicating over a network, thus hiding platform heterogeneity and removing the need to handle low-level network communication details (Zdun et al., 2004). Hence, middleware provides a unified interface to clients, thus requiring management of the APIs between the backend services and the middleware and the API between the middleware and the client applications.

Enterprise Integration Patterns by Hohpe and Woolf (2003)

Enterprise integration focuses on often complex, message-oriented system integrations aiming to automate business processes (Hohpe and Woolf, 2003; Fowler, 2003). The integration of these enterprise applications relies on middleware solutions using queuing technologies (Daigneau, 2011). Hohpe and Woolf (2003) presents technology and product agnostic patterns for enterprise integration focusing on asynchronous messaging. The authors publish the patterns in a book (Hohpe and Woolf, 2003) and online (Hohpe). The patterns are derived from practical experience and follow the Alexandrian (Alexander et al., 1977) pattern form.

Overall, Hohpe and Woolf (2003) publishes 65 patterns organized into seven categories. These categories are ‘integration styles’, ‘messaging channels’, ‘message construction’, ‘message routing’, ‘message transformation’, ‘messaging endpoints’, and ‘system management’. In addition to patterns referring to each other and hierarchies of patterns, the pattern language also comprises ’root patterns’, i.e., patterns that provide a foundation for all other patterns.

Discussion

The *Enterprise Integration Patterns* (Hohpe and Woolf, 2003) aim to enable application developers and system integrators to integrate independent applications using messaging in the context of complex enterprise applications. The ‘message construction’ and ‘messaging endpoint’ patterns can support the *planning & design* phase of a Web API as visualized in Fig. 2.4. Furthermore, the `Test Message` can be employed to enable *testing & securing* and *monitoring & monetization* of a Web API.

Remoting Patterns by Völter et al. (2004)

Völter et al. (2004) use patterns to describe the architecture of distributed object middleware for distributed systems. More precisely, they document remoting patterns that implement the `broker` pattern as the highest-level, compound pattern. Regarding technologies, they focus on CORBA, Web Services, DCOM, Java RMI, .NET Remoting. The pattern language comprises ‘structural’ and ‘behavioral’ patterns. The pattern descriptions have a consistent form derived from Alexander.

Discussion

The *Remoting Patterns* presented by Völter et al. (2004) document the inner workings of distributed object middleware. The pattern `Interface Description` could support the *planning & design* as well as the *integration & consumption* phases of the API management lifecycle for Web APIs used across organizational boundaries presented in Fig. 2.4.

3.4. Object-oriented Software Design Patterns

Object-oriented programming is a paradigm that encapsulates behavior and data in *objects* (Gamma et al., 1995; Daigneau, 2011). An object consists of properties and operations (Gamma et al., 1995). Objects interact via requests that trigger the execution of an operation (Gamma et al., 1995). These objects can be distributed. Objects are usually "fine-gained" modules (Daigneau, 2011).

Patterns of Enterprise Application Architecture by Fowler (2003)

Fowler (2003) is concerned with patterns supporting programmers, designers, and architects in designing the architecture of enterprise applications. Enterprise applications automate business processes while often handling large amounts of data, making them inherently complex. The collection of 51 patterns comprises 'domain logic patterns', 'data source architectural patterns', 'object-relational behavior patterns', 'object-relational structural patterns', 'object-relational metadata mapping patterns', 'web presentation patterns', distribution patterns', 'offline concurrency patterns', 'session state patterns', and 'base patterns'. The author derived the patterns from experience, and they have a consistent form.

Discussion

Overall, the *Enterprise Application Patterns* (Fowler, 2003) focus on the design of complex applications. While some patterns touch the design of Web APIs, most patterns capture general object-oriented programming concepts. The pattern `Remote Facade` is a basic pattern used to realize `Web APIs`, thus it is relevant during the *implementation & deployment* phase of the Web API lifecycle presented in Fig. 2.4. Also, a `Service Stub` can support *integration & consumption.*

Object-Oriented Software Design Patterns by Gamma et al. (1995)

Gamma et al. (1995) published one of the first and most influential pattern catalogs that apply the pattern idea to software engineering, also known as the *"Gang of Four"* or *"GoF"* book (Buschmann et al., 1996). The pattern catalog covers design patterns for object-oriented software resulting from experience to increase the flexibility and reusability of object-oriented software. More specifically, the patterns are *"[...] descriptions of communicating objects and classes that* *are customized to solve a general design problem in a particular context"* (Gamma et al., 1995, p. 3). All patterns have been successfully applied in multiple systems and use a consistent pattern format. Overall, Gamma et al. (1995) describe 23 design patterns. These design patterns are categorized into 'creational', 'structural', and 'behavioral' patterns. In addition to the patterns themselves, Gamma et al. (1995) also describes what patterns are and the benefits of the pattern form.

Discussion

Gamma et al. (1995) presents design patterns concerned with the creation, composition, and interaction between classes and objects in object-oriented systems. Nevertheless, some structural patterns can be relevant for Web API management. First, the `Facade` enables the realization of a Web API, thus it is relevant during the *implementation & deployment* phase of the Web API lifecycle. Also, the `Adapter` pattern can support the integration of Web APIs, thus supporting the *integration & consumption* phase during a Web API management lifecycle as presented in Fig. 2.4. Moreover, the `Proxy` pattern provides the basis for an API gateway.

3.5. Software Architecture Patterns

"A system's architecture specifies the structure of the system, in terms of both the software that implements the system functions and the hardware that provides the operating environment for the software" (Dyson and Longshaw, 2004, p. 2). Hence, we identify API management patterns in software architecture pattern collections in the following. More precisely, we identify and analyze distributed software architecture patterns that support API management.

Pattern-Oriented Software Architecture (POSA) Book Series

The Pattern-Oriented Software Architecture patterns are published across five books, also known as the Pattern-oriented Software Architecture (POSA) series books.

Buschmann et al. (1996) is the first POSA book and an early and very influential publication on patterns in software architecture. The book aims to support novices and experts in designing, maintaining, and changing complex, large-scale systems and effective software production. The patterns rely on the experience of skilled designers and software engineers. Each pattern description has the same structure. Overall, Buschmann et al. (1996) describes three categories of patterns with different levels of abstraction. First, the authors present eight 'architectural patterns', which are the highest-level patterns. Architectural patterns cover the structural organization of software systems. Afterward, Buschmann et al. (1996) cover eight 'design patterns', which prescribe structures of subsystems. Finally, on the lowest level, Buschmann et al. (1996) presents one 'idiom' and refers to sources of more idioms. An idiom is a language-dependent pattern that supports the implementation of behavior in components.

The second POSA book Schmidt et al. (2000) is concerned with patterns for concurrent and networked objects. More precisely, the book covers patterns for 'service access and configuration', 'event handling', 'synchronization', and 'concurrency'.

Kircher and Jain (2004) present the third POSA book and describe patterns for resource management. More precisely, the book covers 'resource acquisition', 'resource lifecycle', and 'resource release patterns'.

The fourth POSA book Buschmann et al. (2007b) relates distributed computing patterns discovered and published by many different software experts to form a pattern language. Overall, the authors present 114 patterns for object-oriented software grouped into 13 problem categories. These patterns also repeat patterns previously published in Buschmann et al. (1996), e.g., the `Proxy` or `Facade` pattern.

Finally, Buschmann et al. (2007a) is the fifth and last POSA book. The book takes a meta-view and discusses the definition and form of patterns, the potential relations between patterns, and the pattern language definition and form.

Discussion

The POSA books are amongst the most renowned software pattern books. The books focus on patterns for object-oriented software systems and the concept of patterns and pattern languages. Hence, as the name states, they focus on the design and architecture of a system.

However, Buschmann et al. (1996) presents the `Facade` pattern, previously published by Gamma et al. (1995). A `Facade` enables the realization of a Web API, and is thus relevant during during the *implementation & deployment* phase of the Web API lifecycle presented in Fig. 2.4.

Moreover, Buschmann et al. (1996) and Buschmann et al. (2007b) present the pattern `Proxy`, i.e., a component that provides security, traffic management, and other housekeeping functionality through which clients communicate with the backend (Buschmann et al., 2007b). Hence, a `Proxy` can realize many API management capabilities.

Patterns for High-Capability Internet-Based Systems by Dyson and Longshaw (2004)

Dyson and Longshaw (2004) presents architecture patterns for high-capability internet technology systems, i.e., systems using internet technology to deliver information and services. The scope is limited to non-trivial, large-scale, mission-critical, enterprise systems. However, these systems can be accessible to internal and external users. The pattern language comprises 26 intern-related patterns categorized into 'fundamental', 'system performance', 'system control', and 'system evolution' patterns.

3.5.0.1. Discussion

The *Patterns for High-Capability Internet-Based Systems* (Dyson and Longshaw, 2004) focus on the overall architecture of internet-based systems, comprising several API management patterns.

First, the ‘system performance patterns’ and ‘system evolution patterns’ can be considered during the *planning & design* phase of the API management lifecycle presented in Fig. 2.4. Moreover, the ‘system control patterns’ are relevant during the *testing & securing* and the *monitoring & monetization* phase.

3.6. Summary

This dissertation aims to support API providers seeking to provide a successful Web API by presenting API management patterns. Hence, the goal of this chapter was to review existing patterns for API management in related literature. However, to the best of the author’s knowledge, no pattern collection explicitly focusing on API management patterns exists. Instead, we reviewed patterns collections concerned with API design, service design including SOA and microservice patterns, middleware design, object-oriented software design, and software architecture to identify API management patterns. Overall, we identified, described, and related 15 pattern collections to the API management lifecycle phases presented in Fig. 2.4.

Overarching observations are that the analyzed pattern collections provide many patterns focusing on the technical implementation of Web APIs, e.g., Web API conversation patterns. In addition, we identified several patterns concerned with API testing, security, performance, monitoring, and evolution. In comparison, we identified only a few patterns for the API lifecycle phases *discovery & onboarding* and *integration & consumption.* Also, the API management patterns that we identified focus primarily on technical solutions and rarely present collaborative, organizational, or process-oriented solution approaches.

CHAPTER 4

Identification of Best Practice Candidates for Code Examples in Web API Documentation

In his seminal paper on sub-routines, Wheeler (1952) already wrote:

> *"However, even after it [a sub-routine] has been coded and tested there still remains the considerable task of writing a description so that people not acquainted with the interior coding can nevertheless use it easily. This last task may be the most difficult."* (Wheeler, 1952, p. 235)

Writing documentation for APIs becomes even more difficult for public, partner, and group Web APIs with unknown, distributed, and heterogeneous API consumers with varying and unknown goals. Since the consumers usually do not have direct access to the backend and Web API developer team, knowledge transfer has to take place in a different way. Therefore, this chapter aims to identify best practice candidates for documenting public, partner, and group Web APIs. We do so by extending the results presented in Bondel et al. (2022) and in the student thesis Cerit (2019).

> "Nowadays, learning new APIs provided by other teams or organizations is a common task for developers (Meng et al., 2018, 2019; Glassman et al., 2018). Hence, developers often face the challenge of evaluating the suitability of an API for solving a specific problem (Meng et al., 2018). Also, developers often have to figure out how to use an API efficiently to solve a specific problem (Meng et al., 2018). Information necessary to accomplish these tasks can comprise knowledge about how domain concepts map to API elements, what use cases the API supports, how different requests affect resource consumption, and how the API reports errors (Robillard and DeLine, 2011; Meng et al., 2018).
>
> API providers use documentation to transfer this kind of information to developers

> in other teams or organizations. Therefore, documentation is a crucial learning resource for API consumers (Robillard, 2009; Lethbridge et al., 2003; McLellan et al., 1998). Moreover, API consumers perceive documentation-related issues, e.g., errors or missing information, as a significant impediment when learning APIs (Meng et al., 2018; Robillard and DeLine, 2011; Robillard, 2009). Thus, the success of a public API can depend on the documentation's ability to meet the consumers' information needs (Meng et al., 2018).
>
> Previous research reveals the vital role of code examples in API documentation (Nykaza et al., 2002; Meng et al., 2018, 2019; Robillard, 2009; Ko et al., 2007; Nasehi and Maurer, 2010; McLellan et al., 1998; Meng et al., 2020; Jeong et al., 2009; McLellan et al., 1998). For instance, examples represent entry points to learning a new API (Meng et al., 2018, 2019) as well as to solve specific problems (Meng et al., 2018, 2019; Nykaza et al., 2002). Also, developers perceive examples as more informative and easier to understand compared to textual descriptions and they convey a feeling of how to best use an API (Meng et al., 2018)."
>
> – Bondel et al. (2022)

For instance, Meng et al. (2019) uncovered that developers consult example and usage scenario sections together about as much as reference documentation. Also, examples and usage scenarios are overall and by far the most viewed resources (Meng et al., 2019). Similarly, in a survey with professional software developers, over half of the participants reported using examples to learn a new API (Robillard, 2009). Nevertheless, code examples in API documentation must meet specific quality criteria to unlock their potential (Meng et al., 2018; Robillard and DeLine, 2011; Robillard, 2009; Nykaza et al., 2002).

While many examples are available online, e.g., on Stack Overflow, developers still attribute more trust to official API documentation (Nasehi et al., 2012). Therefore, we focus on code examples as part of the official documentation published by Web API providers.

In the following, we first define our understanding of code examples in Web API documentation. Afterward, we present the research approach and the resulting best practice candidates. The presentation of these results is followed by their discussion yielding several implications. At last, we summarize the chapter.

4.1. Definition of Code Examples

> "We adopt Robillard and DeLine (2011) definition of code examples in API documentation as: *"[...] listings, of various length, that show an API being used"*. Thus, an example helps developers to understand how to use API elements in a programming context (Watson, 2012; Jiang et al., 2007). There are four different types of code examples, which are *code snippets*, *tutorials*, *samples*, and *production code* (Robillard and DeLine, 2011).
>
> - The smallest type of examples are *code snippets*, which show just one aspect

of an APIs basic functionality (Robillard and DeLine, 2011; Watson, 2012). Similarly, Watson (2012) describes code snippets as short samples of code that show how to use one API element (Watson, 2012).

- A *tutorial* is a sequence of snippets that implement a specific, non-trivial functionality in a step-by-step manner (Robillard and DeLine, 2011; Watson, 2012). A *getting started guide* is a type of tutorial that enables developers to implement a basic usage of the API (Inzunza et al., 2018). Also, a concept which we equate with tutorials are *recipes*. Meng et al. (2019) describes recipes as code examples in a cookbook-like fashion. Each recipe describes how to reach a specific solution given a specific problem (Meng et al., 2018).
- Next, a *sample* is a small and self-contained application (Robillard and DeLine, 2011). In comparison to tutorials that show just one functionality, sample apps demonstrate complete and usable programs comprising several features and potentially additional functionality, e.g., a user interface or error handling (Watson, 2012; Nykaza et al., 2002).
- Finally, API consumers can inspect *production code* for code examples (Robillard and DeLine, 2011). For instance, consumers can extract such code examples from open source systems.

An example does not only consist of code but also entails accompanying explanations (Robillard and DeLine, 2011). These explanations provide information that allows consumers to understand the example code and thus modify it (Nasehi and Maurer, 2010; Nasehi et al., 2012; Ko and Riche, 2011; Meng et al., 2018; Uddin and Robillard, 2015; Thayer et al., 2021; Glassman et al., 2018). Such an explanations can, e.g., comprise a rationale that explains how the code relates to concepts and execution facts (Thayer et al., 2021). The explanations can be incorporated into code in the form of comments (Nasehi et al., 2012) or accompany the example code as textual descriptions (Meng et al., 2018).

Moreover, we distinguish the concept of examples from similar concepts in literature. First, we want to distinguish examples from API *usages*. An API usage captures what an API can be used for (Jiang et al., 2007; Stylos et al., 2009). Hence, examples can demonstrate how to realize usages, i.e., recommended or required sequences of API calls that implement a specific functionality (Jiang et al., 2007). However, an API provider can also use UML2 sequence diagrams to convey usage scenarios as part of API documentation (Jiang et al., 2007).

Next, Thayer et al. (2021) introduces *usage patterns* as a type of knowledge that consumers require to be able to use an API successfully. A usage pattern is a code pattern describing how calls to several APIs should be coordinated and are accompanied by a rationale that explains how the code relates to concepts and execution facts (Thayer et al., 2021). A usage pattern can be a code snippet. Thus, usage patterns convey what a developer can do with an API (Thayer et al., 2021).

Finally, Hoffman and Strooper (2000) and Hoffman and Strooper (2003) introduce the idea of using *test cases* as examples in documentation. A test case comprises preconditions, inputs, and expected results to determine if a system meets specific test objectives (ISO/IEC/IEEE 29119-1:2013)[1]. Test cases are executable examples with expected outputs that guarantee precision, completeness, and machine processability [(Hoffman and Strooper, 2000, 2003)].

As a result, we understand examples as code comprising API requests and expected responses with accompanying explanations that demonstrate how to use an API to realize one or more functionalities. [...]"

– (Bondel et al., 2022)

4.2. Research Approach

This section describes the research approach for identifying best practice candidates as visualized in Fig. 4.1. First, we present a literature review that aims at extracting best practice candidates for Web API documentation from existing research. We identified 17 research papers presenting implications, principles, or observations for API documentation, from which we derived 32 best practice candidates. Afterward, we conducted 13 expert interviews to enrich the previously identified best practice candidates and discovered further candidates yielding a collection of 46 best practice candidates. In addition, we report observations made during the identification of the best practice candidates. As visible in Fig. 4.1, chapter 5 builds on these results.

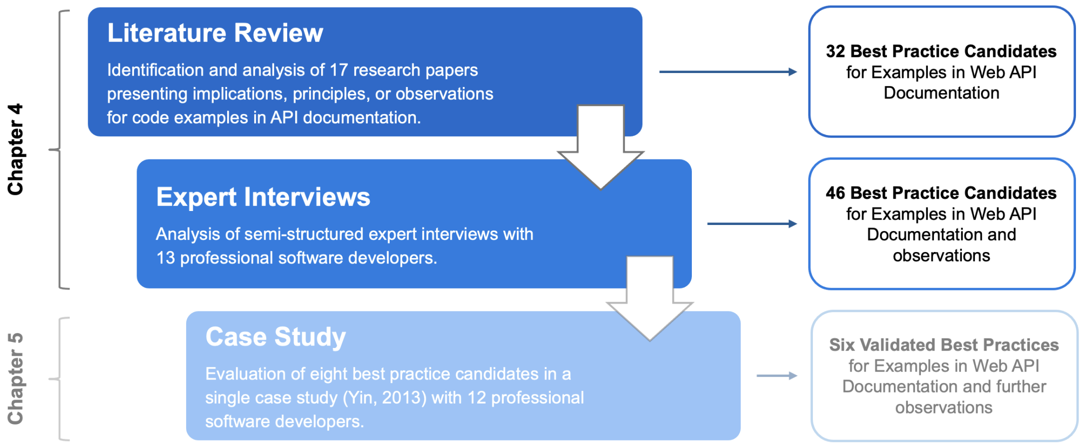

Chapter 4
Chapter 5
Literature Review
Identification and analysis of 17 research papers presenting implications, principles, or observations for code examples in API documentation.
Expert Interviews
Analysis of semi-structured expert interviews with 13 professional software developers.
Case Study
Evaluation of eight best practice candidates in a single case study (Yin, 2013) with 12 professional software developers.
32 Best Practice Candidates for Examples in Web API Documentation
46 Best Practice Candidates for Examples in Web API Documentation and observations
Six Validated Best Practices for Examples in Web API Documentation and further observations

Figure 4.1.: Steps and results of the research approach applied to identify best practice candidates for Web API documentation.

We present details about the literature review and the expert interviews in the following.

[1]ISO/IEC/IEEE 29119-1:2013 was replaced by ISO/IEC/IEEE 29119-1:2022. However, the essence of the definition of test cases remains the same in the new version.

4.2.1. Literature Review

> "We reviewed existing literature on API documentation following an approach inspired by Webster and Watson (2002) to extract best practice candidates for the design and integration of code examples into Web API documentation that improve consumers' API learning. As a starting point, we conducted an extensive search of the databases ScienceDirect[2], ACM[3], IEEE Xplore[4], and Scopus[5]. Afterward, we uncovered additional relevant publications using forward and backward search. As a result, we collected 17 research papers that present implications, principles, or observations for code examples in the documentation of local APIs, non-public Web APIs, and public Web APIs. We include best practice candidates derived from literature concerned with non-public Web APIs and local APIs since we assume they could also apply to public Web APIs. Still, their applicability first needs to be validated."
>
> – Bondel et al. (2022)

Since the goal of the literature review was to identify best practices specifically related to examples in API provider documentation, we excluded general documentation best practice candidates. For example, a general documentation best practice candidate would be to provide consistent navigation and powerful search facilities for the documentation (Meng et al., 2020; Jeong et al., 2009). While these facilities are of importance, they are not specific to examples and thus disregarded. Further best practice candidates concerning general API documentation can be derived from, e.g., Meng et al. (2020), Robillard (2009), Robillard and DeLine (2011), and Jeong et al. (2009).

Also, since we focus on official API documentation published by the API provider, we excluded best practice candidates only relevant to examples presented in Q&A forums like Stack Overflow[6]. An exemplary best practice candidate of this category is that code examples should indicate who authored them to enable consumers to derive their trustworthiness (Robillard and DeLine, 2011). However, by definition, official API documentation is written or at least reviewed and published by the API provider, thus already providing information on the author.

We present an overview of the analyzed 17 research papers and the API context from which these papers derive their implications, principles, or observations in Tab. 4.1. From these research papers, we derived 32 best practice candidates for code examples in official API documentation.

4.2.2. Expert Interviews

Next, we aimed to enrich and potentially identify further best practice candidates for code examples in Web API documentation using semi-structured expert interviews.

> "Overall, we interviewed 13 professionals, including product owners, architects, and

[2]`https://www.sciencedirect.com/`
[3]`https://www.acm.org/`
[4]`https://ieeexplore.ieee.org/Xplore/home.jsp`
[5]`https://www.scopus.com/home.uri`
[6]`https://stackoverflow.com/`

Table 4.1.: Identified and analyzed research papers that present implications, principles, or observations for code examples in API documentation.

Source	**API Context**
Nasehi and Maurer (2010)	Apache POI API, v3.5 (open source library in Java)
Nasehi et al. (2012)	Java programming language
Ko et al. (2004)	Visual Basic .NET 2003
Meng et al. (2018)	Not specific to an API type (includes class-based APIs, library-based APIs, Web APIs, and others)
Meng et al. (2019)	shipcloud API (public Web API, REST-based)
Meng et al. (2020)	shipcloud API (public Web API, REST-based)
Robillard (2009)	Different types of APIs (e.g., an API for access to personal information manager data on Windows mobile-based devices, classic windowing APIs, and Microsoft's Web application development platform .NET)
Robillard and DeLine (2011)	Different types of publicly released APIs (e.g., API for access to personal information manager data on Windows mobile-based devices, classic windowing APIs, and Microsoft's Web application development platform .NET) but most participants had access to the teams that developed the APIs
Sohan et al. (2017)	WordPress REST API V2 (open Web API, REST-based)
Thayer et al. (2021)	d3.js, Natural, OpenLayers, ThreeJS (JavaScript APIs)
Hoffman and Strooper (2000)	Java `Vector` class
Hoffman and Strooper (2003)	Java `StringBufferTest` class and Java class interactions via command line module
McLellan et al. (1998)	RODE API (library-based API)
Inzunza et al. (2018)	Facebook API (Open Web API), Google TensorFlow (open source library), Microsoft Face API (Open Web API, REST-based), UM4RS (C# library)
Nykaza et al. (2002)	3CS SDK (SDK to interact with a realtime database for call center sooftware)
Glassman et al. (2018)	Java APIs `Map.get`, `Activity.findViewByID`, `SQLiteDatabase.query` (class-based)
Jeong et al. (2009)	SAP Enterprise Service-oriented Architecture (eSOA) API (based on XML and WSDL)

Table 4.2.: Overview of interview experts adopted from Bondel et al. (2022).

ID	Role	Experience [years]	Duration [hh:mm]
I1	Software Architect	12	01:15
I2	Product Owner	15	00:32
I3	Enterprise Architect	12	-
I4	Enterprise Architect	10	00:30
I5	Software Architect	29	00:36
I6	Enterprise Architect	7	00:44
I7	Product Owner	12	00:45
I8	Lead Developer	20	01:05
I9	Senior Developer	15	01:31
I10	Senior Developer	6	00:40
I11	Developer	2	00:36
I12	Senior Developer	8	00:41
I13	Senior Developer	20	00:38
	Mean	**12.3**	**00:39**

software developers employed by a large multinational software vendor. The prerequisites for participation in the interviews were that each interviewee has professional experience with API provision or consumption and was actively working on a project concerned with API design and documentation at the time of the research endeavor. The interviews took place between April and July 2019 and lasted 39 minutes on average. We audio-recorded 13 interviews and transcribed the relevant parts[7]. We provide an overview of the interviewees, including an Identifier (ID), a role description, their years of professional experience in software design, and the duration of the interviews, in Tab. 4.2.

We analyzed the data using open coding, selective coding, and constant comparison as described by Wiesche et al. (2017). The analysis of the interviewees yields 33 best practice candidates, of which 19 support best practice candidates previously identified in the literature. Therefore, we identify 46 unique best practice candidates from literature and expert interviewees."

– Bondel et al. (2022)

We assigned each interviewee an ID consisting of the letter 'I' and a number (see Tab. 4.2) . In the remainder of this chapter, we use the ID to link identified best practice candidates to the interviewees who expressed them to enable the reproducibility of results.

[7]Due to a malfunction of the recording device, we did not record the interview with I3. Instead, we analyzed the field notes we made during the interview.

4.3. Best Practice Candidates for Code Examples in Web API Documentation

Overall, we identified 46 best practice candidates as a result of the literature review and the expert interviews. Of these 46 best practice candidates, we derived 13 best practice candidates exclusively from literature, 14 exclusively from expert interviews, and 19 from both literature and expert interviews.

We categorized the 46 best practice candidates into five categories following the approach visualized in Fig. 4.2.

First, we evaluated if a best practice candidate describes a type of information that API code examples should convey to an API consumer. If so, we identified the information and mapped it to one of the three components of *robust API knowledge* according to Thayer et al. (2021). The three components of robust API knowledge according to Thayer et al. (2021) are:

- *Domain Concepts* are concepts of a domain that exist outside of an API and which an API models. Knowledge of the domain concepts enables API consumers to understand what problems an API can solve. Furthermore, conceptual knowledge helps consumers understand the purpose of code and how to manipulate API abstractions to solve specific tasks, i.e., to figure out what functions to call in which sequence. In addition, this knowledge component captures the terminology that the API and its documentation use to refer to these domain concepts. The knowledge of the terminology helps the API consumer identify relevant information within the documentation.
- *Execution Facts* describe an APIs' runtime behavior, e.g., an API call's output and side effects given a specific input. Therefore, execution facts are simple rules that show developers what inputs are valid. Furthermore, they enable consumers to predict return values, the APIs' internal state, data and control flows, and errors given specific inputs. Execution facts also comprise the effects of execution environments and existing bugs on API behavior. Knowledge of execution realities enables developers to write, test, debug and repair code, handle error messages, and reason about unexpected behavior.
- Finally, *API Usage Patterns* capture how to combine and modify API interactions to realize specific outcomes, e.g., a capability or functionality. In addition, API usage patterns convey best practices for realizing these outcomes. An integral aspect of an API usage pattern is its rationale which explains why a pattern works and why it is designed the way it is by linking it to domain concepts and execution facts. Compared to single examples that show one usage, an API usage pattern, including its rationale, presents a range of possibilities. Hence, API usage patterns also show API consumers how to adapt code to realize different desired behaviors.

Thayer et al. (2021) states that API documentation needs to transfer all three components of knowledge to enable API consumers to understand and use an API effectively.

In addition, we identified two best practice candidates aiming to convey information, but the information does not map to either of the three categories of robust API knowledge. We recognized that these two best practice candidates are concerned with information on the quality

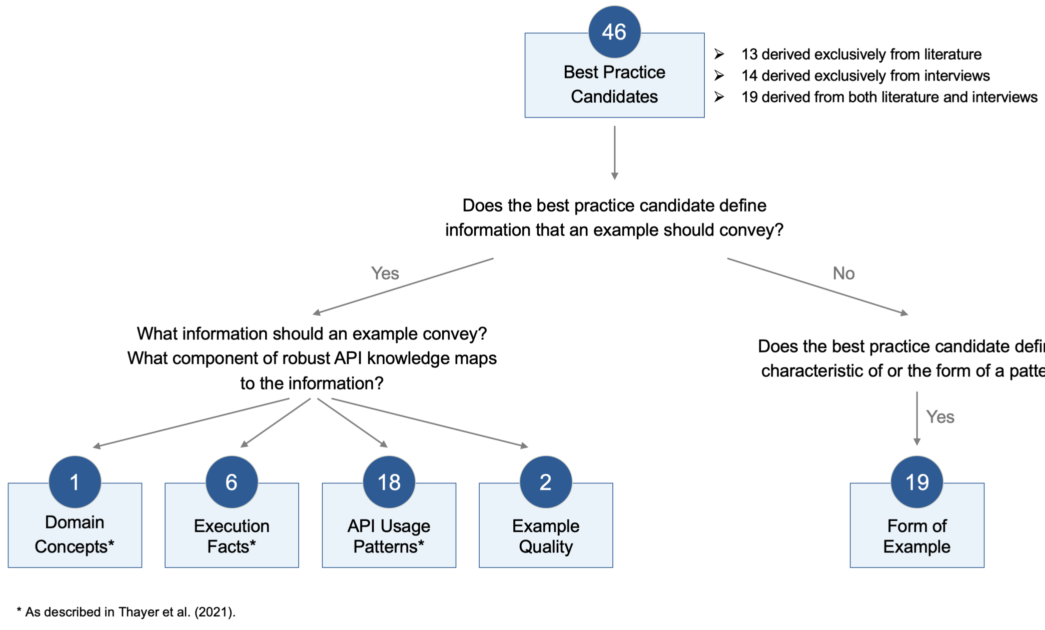

46
Best Practice Candidates
13 derived exclusively from literature
14 derived exclusively from interviews
19 derived from both literature and interviews
Does the best practice candidate define information that an example should convey?
Yes
No
What information should an example convey? What component of robust API knowledge maps to the information?
Does the best practice candidate define a characteristic of or the form of a pattern?
Yes
1
Domain Concepts*
6
Execution Facts*
18
API Usage Patterns*
2
Example Quality
19
Form of Example
* As described in Thayer et al. (2021).

Figure 4.2.: Approach to categorizing the best practice candidates according to the knowledge they aim to transfer or the form they should have.

of the examples. Hence, we introduced the fourth knowledge category *Example Quality*. Information about the quality of examples enables API consumers to judge the examples' reliability and manage their expectations.

We need to differentiate between code examples in the API documentation and the best practice candidates we present. A code example consisting of some valid code, by definition, provides knowledge of execution facts and, in many cases, also of API usage patterns. However, we analyzed what knowledge component the application of a best practice candidate aims to inject into an example. We did so by identifying the information mentioned explicitly in a best practice candidate and mapping that information to the various components of API knowledge.

Afterward, we examined the best practice candidates that do not describe a type of knowledge that they aim to convey to an API consumer. As a result, we realized that all these best practice candidates describe some characteristic of example presentation or the form in which an example should be composed to be valuable. Hence, we subsumed these best practice candidates in the category *Example Form*.

Overall, we identified 27 best practice candidates that aim to transfer knowledge to API consumers; one is concerned with domain concepts, six with execution facts, 18 with API usage patterns, and two with example quality. In addition, we identified 19 best practice candidates that describe characteristics or the form code examples should adhere to to make them useful.

In the following, we first present the best practice candidates that aim to transfer knowledge to API consumers, followed by the best practice candidates concerned with the form of code examples.

4.3.1. Best Practice Candidates Aiming at Knowledge Transfer

First, we present the best practice candidates that aim at transferring knowledge to API consumers.

We provide an overview of the best practice candidates concerning knowledge in examples in official public, partner, or group Web API documentation in Tab. 4.3. We assigned a unique ID to each of these knowledge-related best practice candidates consisting of the letter *K* followed by an increasing number, e.g., K03 for the third best practice candidate. Furthermore, the table presents the literature sources and/or interviews from which we derived the best practice candidates. Finally, we indicated which component of robust API knowledge each best practices candidate aims to convey. The sequence of best practice candidates in the table is of no particular meaning. However, we clustered best practice candidates concerning similar topics together.

In the following, we describe each of these best practice candidates in more detail. The best practice candidate K01 aims to convey knowledge on the domain concepts. It is followed by the best practice candidates K02-K07 which aim to transmit knowledge on execution facts. Finally, the best practice candidates K08-K25 are concerned with knowledge of API usage patterns in code examples and K26-K27 with knowledge of example quality.

K01: Explanations of relevant conceptual knowledge should accompany each example.

According to Thayer et al. (2021), *"Domain concepts are abstract ideas that exist outside of an API, which an API attempts to model [...]"* (Thayer et al., 2021, p.5). API consumers require conceptual knowledge to understand for what purposes they can use an API (Thayer et al., 2021).

Developers have different approaches to learning, i.e., some developers seek to understand concepts of an API before starting to use it, and others do not want to look up conceptual information before starting to use the API (Meng et al., 2019, 2018). Therefore, providing conceptual knowledge relevant to an example together with the example supports both types of learners (Meng et al., 2019). Hence, the examples should include textual explanations that highlight how domain concepts map to API elements (Meng et al., 2019). In addition or alternatively, the API provider can include conceptual information into the example code using comments (Meng et al., 2019).

Similar to literature, the interviewees emphasized the importance of conceptual knowledge (I12, I13). For instance, understanding domain concepts is a prerequisite to understanding the problem that the API solves and how to use the API (I1). Moreover, conceptual knowledge enables consumers to optimize the implementation (I9, I12), as I1 described:

> *"And I think it is important to understand the domain because then you know the context of the problem. You activate your brain, so to speak. Only then do you can think critically about it [the API]. If we didn't know that [the concept], we would only think from input to output. If one knows the whole picture, one might choose the much shorter solution."* (I1)

Table 4.3.: Best practice candidates for code examples in Web API documentation aiming to transfer API knowledge.

ID	Best Practice Candidate	Literature Sources	Interview Partner	Knowledge Component
K01	Explanations of relevant conceptual knowledge should accompany each example.	(Meng et al., 2018; Meng et al., 2019; Thayer et al., 2021)	I1, I9, I12, I13	Domain Concepts
K02	Examples need to present correct request syntax and semantics, including all necessary HTTP headers, valid parameters, valid data types, and data formats.	(Sohan et al., 2017)	I4, I7, I8, I9, I11, I13	Execution Facts
K03	Example explanations should provide information on pre- and post-conditions of API interactions.	(Hoffman and Strooper, 2000; Hoffman and Strooper, 2003)	I11, I13	Execution Facts
K04	Example explanations should provide information on non-deterministic API behavior.	(Hoffman and Strooper, 2000; Hoffman and Strooper, 2003)		Execution Facts
K05	An example should describe valid test data.		I13	Execution Facts
K06	The explanation should describe shortcomings of the API itself and potential workarounds.	(Nasehi et al., 2012)		Execution Facts
K07	Explanations should describe the reasons for error messages and measures to solve these errors.		I1, I6, I7, I8, I11	Execution Facts
K08	Examples should cover all common usage scenarios of an API.	(Nasehi and Maurer, 2010; Ko et al., 2004)	I5, I7, I8, I9	API Usage Patterns
K09	An example should describe the intended usages of the API.		I9	API Usage Patterns
K10	Documentation should entail a sample app that demonstrates the primary usage(s) of the API.		I1, I6	API Usage Patterns
K11	The API provider should describe an API's capabilities at the beginning of the example.	(McLellan et al., 1998)	I12	API Usage Patterns
K12	Examples should demonstrate how to coordinate sequential requests to a single API to implement (more complex) functionality.	(Nasehi and Maurer, 2010; Robillard, 2009; Sohan et al., 2017)	I1, I2, I4, I7, I8, I12	API Usage Patterns

Table 4.4.: Best practice candidates for code examples in Web API documentation aiming to transfer API knowledge (continued).

ID	Best Practice Candidate	Literature Sources	Inverview Partner	Knowledge Component
K13	Examples should demonstrate how to coordinate requests to multiple APIs to implement (more complex) functionality.	(Robillard and DeLine, 2011; Thayer et al., 2021)	I2, I13	API Usage Patterns
K14	Examples should entail or be accompanied by all information necessary for successful authentication.		I2, I7, I11, I13	API Usage Patterns
K15	Explanations accompanying examples should describe the limitations of the solution that the example presents.	(Nasehi et al., 2012)		API Usage Patterns
K16	Examples should implement "best practices" specific to an API.	(Robillard and DeLine, 2011; Robillard, 2009; Nasehi et al., 2012)	I8, I9	API Usage Patterns
K17	Examples should adhere to general and community-specific programming conventions.	(Meng et al., 2018; McLellan et al., 1998)		API Usage Patterns
K18	Explanations of code examples should reference related or alternative solutions.	(Nasehi et al., 2012)		API Usage Patterns
K19	Examples should demonstrate alternative solution approaches, including corner cases.		I2, I8, I11, I12	API Usage Patterns
K20	Examples should also demonstrate "unhappy paths".		I3, I8, I11	API Usage Patterns
K21	Tutorials should demonstrate the APIs basic functionality and common use cases.	(Meng et al., 2018; Nykaza et al., 2002)	I1, I2, I6	API Usage Patterns
K22	Documentation should include advanced tutorials and applications that transfer information on complex API interactions.	(Robillard, 2009)		API Usage Patterns
K23	Examples should convey information on the API design rationale.	(Nykaza et al., 2002; Robillard, 2009)		API Usage Patterns
K24	Explanations should describe the rationale for each part of an example request.		I11, I12, I13	API Usage Patterns

Table 4.5.: Best practice candidates for code examples in Web API documentation aiming to transfer API knowledge (continued).

ID	Best Practice Candidate	Literature Sources	Inverview Partner	Knowledge Component
K25	Example explanations need to explain how the example code relates to low-level API elements.	(Thayer et al., 2021)	I9	API Usage Patterns
K26	The documentation should not indicate that an example is old if it still works.	(Robillard, 2009; Robillard and DeLine, 2011)		Example Quality
K27	Examples should present indicators that provide information on code example quality.	(Glassman et al., 2018)		Example Quality

In addition, I1 acknowledged that describing concepts in an easy-to-consume way might not always be easy but worthwhile since developers abandon APIs if they do not understand them.

However, Ko and Riche (2011) and Jeong et al. (2009) show that documentation should also introduce conceptual knowledge at the entry point into the documentation since knowledge about concepts enables API consumers to search the documentation more effectively and judge the relevance of examples and usage scenarios for specific problems. Similarly, some of the interviewees proposed presenting conceptual knowledge independent of examples, e.g., at the beginning of the documentation (I1) or in separate documentation resources like books (I13).

Summarizing, literature and interview experts agreed on the importance of conceptual knowledge as part of API documentation. However, while Meng et al. (2018) and Meng et al. (2019) argue that conceptual knowledge should accompany examples, Ko and Riche (2011), Jeong et al. (2009), and the interviewees I1 and I13 advocate for presenting conceptual knowledge separately of examples, e.g., at the beginning of the documentation.

K02: Examples need to present correct request syntax and semantics, including all necessary HTTP headers, valid parameters, valid data types, and data formats.

First of all, examples have to demonstrate required and optional HTTP headers and their valid values, otherwise, consumers struggle to include the correct HTTP headers into API requests (Sohan et al., 2017). Moreover, examples should demonstrate valid data formats for each data element in the request body (Sohan et al., 2017). For some data types it might not be clear what data format the API expects, e.g., the expected format of a `Date` data type. The `Date` data type could expect the format `yyyy-mm-dd`, `yyyy:dd:mm`, or `yy-dd-mm,hh-mm-ss` to name just a few options. Finally, examples need to demonstrate the use of expected data types, e.g., Integer or Array, for API elements in request bodies (Sohan et al., 2017). Information on the expected data types in response messages prevents consumers from using a trial and error approach to identify the data types (Sohan et al., 2017).

The interviews confirmed the importance of information on the syntax and semantics as part of examples:

> *"So that's certainly the most important thing - syntax: "what do I do", and semantics: "how to use it"."* (I9)

Also, examples should clarify what parameters (I7, I8, I11) and data types (I4, I13) are valid in an API request.

K03: Example explanations should provide information on pre- and postconditions of API interactions.

Preconditions are conditions that must be true before executing an API request. On the other hand, postconditions are conditions that must be true after the execution of the API request. Explanations as part of examples in API documentation should provide information on such pre- and postconditions (Hoffman and Strooper, 2000, 2003).

The interviewees agreed that it is important to provide information on requirements to be fulfilled before interacting with an API (I11, I13). Moreover, answering the question of what information the interviewee expects when using examples and usage, I11 stated:

> *"Also, what are important conditions so that the endpoint can be called in the first place? Maybe also something like postconditions, i.e., what does the endpoint do and what you can do afterward."* (I11)

K04: Example explanations should provide information on non-deterministic API behavior.

Nondeterminism describes the behavior of an API that returns different responses for identical requests. API consumers should know which API requests are nondeterministic (Hoffman and Strooper, 2000, 2003).

K05: An example should present valid test data.

The API provider must explain what test data is available for each example (I13). When asked what information consumers require to learn an API using examples, I13 stated:

> *"The necessary test data you need to use the API. It doesn't help to create a tutorial with predefined data if every little deviation leads to errors. Therefore, the test data should be provided and explained."* (I13)

<u>K06:</u> The example explanation should describe shortcomings of the API itself and potential workarounds.

Potential shortcomings of an API are, for example, the lack of certain functionality or a bug. The documentation should disclose such flaws (Nasehi et al., 2012).

<u>K07:</u> Explanations should describe error messages, the reasons for error messages, and measures to solve these errors.

The API provider should provide explanations of the meaning and potential reasons for specific errors (I1, I6, I7, I8, I11), as I8 described:

> *"For me, it would be important to know the root of the problem. Most error codes are not very expressive. At least [provide] further information on where the error comes from because the [error] code itself is of no use to me as a consumer. I only know that something went wrong. Then I can start to guess and play through different scenarios until I understand the error."* (I8)

The provider should aim to cover the most common errors since it is not realistic to explain all potential errors (I8). Therefore, the provider should also present information on how consumers can proceed to discover further details on unknown errors (I8). This explanation could be part of the error message (I13). At the same time, however, the explanation of an error message should not reveal any internal or security-relevant information (I1, I6, I13).

<u>K08:</u> Examples should cover all common usage scenarios of an API.

The example section of API documentation should provide examples for all basic usage scenarios of an API (Nasehi and Maurer, 2010; Ko et al., 2004). Examples of common usage scenarios enable consumers to identify what an API is intended to be used for (Ko et al., 2004).

Several experts agreed that documentation cannot cover all usage scenarios but should cover the most important ones that the provider intends the API to serve (I5, I7, I8, I9). For instance, I9 stated:

> *"It would definitely be helpful to do this [provide test cases] for projects simply for the most important and critical use cases. This way, you can at least find out how the developers perceive the approach based on the important examples. Either way, you won't get one hundred percent coverage."* (I9)

<u>K09:</u> An example should describe the intended usages of the API.

Consumers want short explanations of the use cases for which the API provider intends them to use the API (I9):

"What I personally need in a tutorial are 10-15 sentences explaining what the tutorial is about and what [problem] it solves. What many tutorials do, is start right away with code snippets. I actually prefer something like a product description. So why and for what did we design this, and, especially, what can it do. [...] Because then I still have in mind what the use case for it [the API] is actually." (I9)

K10: Documentation should entail a sample app that demonstrates the primary usage(s) of the API.

The API provider should present a sample app that implements the main usage or usages of an API (I1, I6). A sample app enables consumers to understand relevant flows more easily (I1, I6). The provider can use GitHub[8] or a similar repository to publish the sample app (I6). I1 described the need as follows:

"There should be a central code example for the entire API that represents not only the main problem but also the main flow through the program once." (I1)

However, a sample is written in only one programming language, limiting its usability (I6). Also, sample apps often only present one simple happy path and are sometimes not ready for production (I6). Thus, a sample is useful, but consumers require additional information to learn an API easily.

K11: The API provider should describe an API's capabilities at the beginning of the example.

The description of API capabilities enables consumers to form hypotheses about the APIs functionality and recall knowledge related to similar APIs and the domain (McLellan et al., 1998). As a result, the developers can form a better understanding of the API (McLellan et al., 1998).

Even though pointing more to documentation in general, I12 also emphasized the importance of explicitly mentioning API capabilities:

"But usually there is a certain set of capabilities for an API; so this is what my product can do, this is what it cannot do. [...] Because often it is also not really clear when you start learning the API if it can solve what you actually want to solve." (I12)

K12: Examples should demonstrate how to combine sequential requests to a single API to implement (more complex) functionality.

The implementation of more complex functionality often requires the combination of several requests to an API (Ko et al., 2004). Examples are especially useful for complex API interactions since they allow developers to comprehend how to combine requests to achieve a specific goal

[8] `https://github.com/`

(Robillard, 2009; Nasehi and Maurer, 2010). Hence, examples need to demonstrate how to implement features that require a sequence of API requests, i.e., how to realize the prerequisites for making a specific API call (Sohan et al., 2017).

The interviewees confirmed that it is essential for documentation to explain how to string together requests to an API (I7, I8) since it is difficult for the consumers to figure out complex sequences without help (I12). Also, suppose the documentation does not provide any information about sequences, the consumers might need to employ a trial and error approach (I4) or contact the provider to learn about them (I1). Therefore, I12 stated:

> *"Oh, it [a usage scenario] helps a lot because when you have complex flows, it really helps you to see what entities need to be created and what steps are needed before you can actually do what you want to do. [...] In cases where you expect multiple API calls and where it is not that simple, there, it helps a lot. Otherwise, it would not be obvious what to do."* (I12)

A tool that providers can use to present information about API call sequences is Postman script[9] (I2).

K13: Examples should demonstrate how to coordinate requests to multiple APIs to implement (more complex) functionality.

The implementation of functionality often requires combining calls to different APIs (Thayer et al., 2021). Therefore, developers expect examples that show how to realize functionality that requires the combination of requests to several APIs (Robillard and DeLine, 2011).

The interviewees stated that it is challenging to document API interactions involving more than one API (I2, I7). One reason for this difficulty is that if there are changes to one of the APIs, the example might not work anymore (I8). However, examples can be an approach to documenting such dependencies (I13). In summary, the documentation of usages comprising interactions with several APIs using examples can be helpful but might go beyond what is necessary, as I2 described:

> *"I think this is very good, and I can also imagine that this could even go beyond what is necessary. For example, other previously undocumented APIs could be included in these scenarios and documented as well, so to speak."* (I2)

K14: Examples should entail or be accompanied by all information necessary for successful authentication.

Several interviewees claimed that it is important for API providers to present information on authentication (I2, I7, I13). Authentication can be difficult since the consumers need to make sequential requests that require a handover of previously received data into new requests (I2), as I11 emphasized:

[9] `https://learning.postman.com/docs/writing-scripts/intro-to-scripts/`

> *"What I often find difficult and doesn't always work is handling authentication tokens and so on. These are the parts you have to do completely by yourself, and you can't just take an example."* (I11)

K15: Explanations accompanying examples should describe the limitations of the solution that the example presents.

Consumers want to understand the potential downsides of a solution concerning performance and security risks, and potential usability limitations. Hence, the documentation should make the limitations of an example transparent (Nasehi et al., 2012).

K16: Examples should implement "best practices" specific to an API.

Best practices demonstrate how an API consumer should use an API, e.g., how to best combine different elements to unleash the API's full potential (Robillard, 2009; Robillard and DeLine, 2011). Furthermore, accompanying descriptions should explain these best practices and their rationale (Nasehi et al., 2012).

Similarly, the interviewees stated that they would like to learn about best practices of using an API (I8, I9):

> *"Examples also show very well what best practices or standards the providers follow. If the provider gives you a guideline on how to do it, it's a big help. But, of course, you don't find that very often."* (I8)

K17: Examples should adhere to general and community-specific programming conventions.

Since developers expect examples to follow common and community-specific programming conventions (Meng et al., 2018), violations of these rules can lead to developers making erroneous assumptions about the APIs' behavior (McLellan et al., 1998). Also, developers perceive examples as sloppy if they do not follow conventions (McLellan et al., 1998).

K18: Explanations of code examples should reference related or alternative solutions.

Related or alternative solutions for a problem might use different classes of an API, different APIs, or different API versions. Therefore, the code examples should provide links to these related or alternative solutions (Nasehi et al., 2012).

K19: Examples should demonstrate alternative solution approaches, including corner cases.

Alternatives show different paths to solve a specific problem, with a corner case being a particular type of alternative that demonstrates an unusual approach to reach the solution. If alternative solutions exist, the API provider should describe them (I1, I8, I11, I12), e.g., using examples. Similarly, the provider should make corner cases explicit (I2, I12). However, alternative solutions are only relevant for more complex API interactions, as I8 explained:

> *"Yes, I am interested in alternative ways to do this [solve a problem]. But it depends on what I want to implement. If it's something straightforward, then it's probably not that important. For complex tasks, when you also need to optimize the implementation for efficiency, this is probably more important. [...] So first of all, the provider should, of course, represent the normal flow, but at the appropriate place, it can also refer to: "if you want to do that, then you have to read on here." Branches basically. Without further complicating the simple flow, you refer to an alternative."* (I8)

Yet, one expert also doubted the helpfulness of showing alternative approaches to solve a problem (I1). Instead, API providers should only demonstrate the best solution approach, thus relieving the API consumers from choosing a path.

K20: Examples should also demonstrate "unhappy paths".

Developers make mistakes when learning APIs. Hence, documentation should also present examples of common errors or "unhappy paths" (I3, I8), how these errors come about (I11), and how to handle them [I8, I11]. As I8 described:

> *"This is actually rather common that you set up everything and try to run it, but nothing works at first. [Then,] It is very helpful to perhaps also have an overview showing: "okay, these are the usual errors". If this [your error] is not listed, you still need to know how I get more information. This can be something like a forum to which you refer or a FAQ. Something like a knowledge base."* (I8)

K21: Tutorials should demonstrate the API's basic functionality and common use cases.

An introduction of supported use cases enables consumers unfamiliar with an API to understand what the API does (Meng et al., 2018; Nykaza et al., 2002). The tutorials should be short and demonstrate the basic functionality of the API (Nykaza et al., 2002). However, it is likely that only developers following a concept-oriented learning strategy use such learning tutorials (Meng et al., 2018). Developers with a code-oriented learning strategy will use such tutorials only if they exactly match the problem they want to solve (Meng et al., 2018).

The interviewees also expected a set of tutorials that lead consumers through implementing the APIs' basic functionality (I1, I2). These tutorials should entail all steps to get started with an API but still need to be concise since the consumers simply want to understand the basics of the API and dive into details later (I1). Furthermore, a good tutorial can positively influence the likelihood of choosing an API (I6). Hence, when asked when to use examples and tutorials, I1 answered:

> *"At the very beginning, so essentially at the very beginning of learning. I use it, especially when I've never used the library before, and I'm wondering how to get started with it. Once I understand it and have the concept sorted in my head, I move on to the detailed technical specifications."* (I1)

K22: Documentation should include advanced tutorials and applications that transfer information on complex API interactions.

A mismatch between the consumers' complex goals and presented simple examples can become a hindrance. Robillard (2009) observed these hindrances, especially in situations in which consumers wanted to realize more complex solutions. Hence, documentation should include advanced tutorials with more complex examples (Robillard, 2009).

K23: Examples should convey information on the API design rationale.

According to Robillard (2009) and Nykaza et al. (2002), examples should transfer information on the APIs design rationale. The documentation can convey this information, for example, using annotations in the example code (Nykaza et al., 2002).

K24: Explanations should describe the rationale for each part of an example request.

Explanations should describe what an element of a call does and what it triggers in the system (I11, I12, I13). If the provider does not add rationales to examples, the examples are less relevant for consumers (I3, I7):

> *"The background information is important, of course; Why is something the way it is? Ultimately also the results. So, not only, "do this and that and then you're done"."* (I13)

K25: Example explanations need to explain how the example code relates to low-level API elements.

Consumers need to be able to connect knowledge about API elements and examples. Hence, example explanations need to relate the example code to low-level API elements (Thayer et al., 2021).

The interviewee I9 agreed with this best practice, stating:

> *"But I want to have the possibility to dig deeper in the tutorials. So each simple case should link to more detailed case variants or documentation."* (I9)

K26: The documentation should not indicate that an example is old if it still works.

API consumers are less likely to use examples if the documentation suggests that they are old (Robillard, 2009; Robillard and DeLine, 2011).

K27: Examples should present indicators that provide information on code example quality.

Glassman et al. (2018) identified the need for indicators of example quality in the context of examples retrieved from GitHub repositories. However, we assume that official documentation can also profit from quality indicators. Such quality indicators could be, for instance, the aggregated rating of consumers using the example.

4.3.2. Best Practice Candidates Concerning the Form of Code Examples

In addition to best practice candidates that define knowledge that examples should convey to API consumers, we identified 19 best practice candidates that describe some characteristic of the presentation or form that a code example should have to be useful. We present an overview of these best practice candidates in Tab. 4.6 and Tab. 4.7. Again, we assign a unique ID to each best practices candidate. The unique ID consists of the letter F followed by an increasing number, e.g., F02 for the second best practice candidate. Also, we indicate from which literature sources and/or interview partner(s) we derived these best practice candidates. Again, the sequence of best practice candidates in the table is of no particular meaning but we list best practice candidates concerning similar topics next to each other.

In the following, we describe each of the best practice candidates concerning the form of code examples in more detail.

F01: The documentation should enable the execution of examples within the documentation or common tools.

Consumers want to use the API as quickly as possible to learn about its behavior (Meng et al., 2019). Hence, the API provider should include try-out functions into the documentation that enable consumers to execute examples with different parameters and inspect the responses (Meng et al., 2019; Jeong et al., 2009). Such try-out functions allow API consumers to understand the API's functionality and verify their assumptions about it (Jeong et al., 2009).

The interviewees confirmed the findings in the literature by stating that they like to use try-out functions when learning an API. While some interviewees said that embedded try-out functions are nice-to-haves (I8), others considered them integral to good documentation (I12). Furthermore, the try-out features enable the consumers to gain an initial feeling for the APIs functionality (I2, I9, I12) and thus lower the barrier to using an API (I12, I13), as I13 explained:

> *"Often, you have this try-out where you can simply upload something and try it. That is always helpful because you see the request and the response. Of course, errors often*

Table 4.6.: Best practice candidates for code examples concerning the form of the examples.

ID	Best Practice Candidate	Literature Sources	Interview Partner
F01	The documentation should enable the execution of examples within the documentation or common tools.	(Meng et al., 2019; Jeong et al., 2009)	I2, I3, I6, I8, I9, I10, I11, I12
F02	Examples should provide client code in different programming languages.		I4
F03	Examples should be freely accessible.		I8
F04	Examples need to be correct and up-to-date not to frustrate consumers.	(Meng et al., 2018; Robillard and DeLine, 2011)	I1, I2, I8, I9, I11, I12, I13
F05	Examples should be copyable and thus complete.	(Meng et al., 2018; Meng et al., 2019; Hoffman and Strooper, 2000; Hoffman and Strooper, 2003)	I1, I8, I9, I11
F06	Code snippets, tutorials, and their explanations should be as concise and problem-oriented as possible.	(Nasehi et al., 2012)	I1, I7, I10, I12, I13
F07	Tutorials should present usages following a storyline.		I2, I9, I11
F08	The documentation should highlight how explanations relate to the code.	(Meng et al., 2019; Meng et al., 2020)	
F09	The documentation should make it easy to separate example code from textual descriptions.	(Meng et al., 2019)	
F10	The documentation should highlight crucial elements in the code examples and explanations.	(Nasehi et al., 2012)	I13
F11	Tutorials should be accompanied by suitable visualizations.		I8
F12	Tutorials should have a consistent level of detail across all steps.		I13
F13	Tutorials should be structured to show some intermediate result after each step.	(Inzunza et al., 2018)	I13
F14	Documentation should reuse compact and readable test cases as code examples.	(Hoffman and Strooper, 2000; Hoffman and Strooper, 2003; Nasehi and Maurer, 2010)	I3, I4, I8, I9

Table 4.7.: Best practice candidates for code examples concerning the form of the examples. (continued).

ID	Best Practice Candidate	Literature Sources	Interview Partner
F15	Each example should comprise a concise unit of functionality that can be combined into more complex functionality.	(Meng et al., 2018; Robillard and DeLine, 2011; Nasehi et al., 2012)	I8
F16	Documentation should present examples with different, e.g., increasing, levels of complexity to meet the needs of consumers with varying expertise.	(Nasehi et al., 2012; Nykaza et al., 2002)	I9, I11, I12
F17	API providers should design examples set in familiar domain contexts.	(Nasehi et al., 2012)	
F18	Examples need to relate to the overall domain of the API.		I4
F19	Examples should evolve according to the consumers' needs.	(Nasehi et al., 2012)	

> *occur when you make minor changes to the request. You work with a real system."* (I13)

However, the try-out functions do not necessarily need to access the productive API but can also use a sandbox environment (I3, I8, I9, I11, I12). For instance, I9 reported:

> *"I can think of a programming language that offered something like a sandbox. I think it was Go*[10]*. I, personally, find that very helpful for the process of learning."* (I9)

Also, the interviewees agreed that the applicability of try-out functions depends on the complexity of the API (I8). Generally, consumers expect executable examples for small tasks and simple services but not for complex interactions (I3, I8).

Moreover, the interviewees repeatedly mentioned specific tools that implement executable examples. One frequently mentioned tool is Swagger UI[11], which generates an interactive user interface for APIs from a standardized OpenAPI Specification (OAS)[12] (I2, I10). The interviewees stated that Swagger UI is useful for implementing executable examples for simple API requests (I2). However, the tool has limitations regarding implementing examples that require several steps and passing along data between these steps (I2). Therefore, another toolset that consumers like is Postman collections[13] together with Postman scripts[14] (I6, I10). Postman collections are executable API descriptions, and Postman scripts enable a consumer to concatenate

[10] `https://go.dev/`
[11] `https://swagger.io/tools/swagger-ui/`
[12] `https://swagger.io/specification/`
[13] `https://www.postman.com/collection/`
[14] `https://learning.postman.com/docs/writing-scripts/intro-to-scripts/`

requests, taking into account dynamic behavior. Finally, one interviewee mentioned the RAML console[15] as a useful tool (I10).

However, I9 and I10 stated that it is confusing for consumers if examples in a try-out function are erroneous. Hence, examples in try-out functions need to be correct.

F02: Examples should provide client code in different programming languages.

Developers expect client code in different programming languages (A4). Tools exist that automatically generate such client code, as A4 reported:

> *"In a previous project, it was possible to create client code in a console with a click. We offered different languages, which I would definitely recommend."* (A4)

F03: Examples should be freely accessible.

In some cases, examples and other learning materials are only accessible after registration, which frustrates developers:

> *"Something that always annoys me is when you have to register to access important learning materials. The product doesn't necessarily have to be free, but at least the descriptions should be publicly accessible."* (I8)

F04: Examples need to be correct and up-to-date not to frustrate consumers.

A barrier that consumers regularly face is out-of-date or erroneous example documentation (Meng et al., 2018; Robillard and DeLine, 2011; Hosono et al., 2019; Uddin and Robillard, 2015). Hence, examples should be correct and up-to-date (Meng et al., 2018; Robillard and DeLine, 2011).

The expert interviews confirmed that consumers are often confronted with outdated API documentation (I1, I8, I9, I13). The reason for outdated API documentation is that an API changes over time, but the provider does not update the documentation in time (I1, I2, I8). Providers neglect to update documentation because of the effort of constantly keeping APIs and their documentation in sync (I8, I12). However, faulty examples are frustrating for consumers (I1, I9, I11, I13), especially since it is challenging to fix out-of-date examples if the consumer does not have the necessary domain knowledge (I12). However, in line with the findings of Lethbridge et al. (2003), I9 stated that even outdated examples can be helpful:

> *"What strikes me most negatively is when the examples do not work. This happens quite often. I suspect they were written for one version and then not updated in subsequent versions. Something has changed, and then, [the provider] did not follow it up. But it really upsets me. But that's not quite as bad as having nothing at all because at least you get a hint of how it might work."* (I9)

[15] `https://raml.org/blogs/raml-console-20`

However, outdatedness is not the only reason for erroneous documentation. For instance, the API provider sometimes does not invest enough effort into documentation, leading to API documentation that neglects to describe necessary steps (I2). Such incomplete documentation makes it difficult for consumers to understand an API, especially when learning a new API (I2). Hence, I2 explained:

> *"Another point is the number of errors in the examples. And if you go through a tutorial and you don't get what you expect or what was promised, that is also very bad. Let's say we have a description saying we will do that and get this; in the end, then I expect that the outcome will exactly be this. So if you follow all the steps and you still do not get there, that's definitely a problem. It could be because [...] some steps are missing. Or let's say the order of the request has changed."* (I2)

F05: Examples should be copyable and thus complete.

API consumers tend to copy examples to use them as they are or adapt them to their needs (Nykaza et al., 2002; Meng et al., 2019, 2018; Hoffman and Strooper, 2000, 2003; Nasehi and Maurer, 2010). Thus, examples need to be complete in the sense that they are ready to execute after being copied (Meng et al., 2019, 2018; Hoffman and Strooper, 2000, 2003). For instance, examples should not use placeholders pointing to other snippets that have to be included for a copied example to work (Meng et al., 2019).

The interviewees agreed that consumers like to copy example code as a starting point and to later adapt it to their needs (I1, I8, I11). Also, examples let consumers explore the APIs' functionality and limitations (I9, I11). For instance, I1 explained:

> *"Well, at least you have a starting point. Even if you just copy and paste it [the example], you already understand a bit more through the interactivity. Then, if you have a complete example and it works, you can start from that and make changes and extend it. [...] I think we just learn better when we have something to copy instead of making it all up ourselves."* (I1)

F06: Code snippets, tutorials, and their explanations should be as concise and problem-oriented as possible.

Nasehi et al. (2012) discovered that consumers prefer code snippets that are short and that have a reduced code complexity (Nasehi et al., 2012).

Similarly, the interviewees mentioned that they do not like too detailed documentation that consists of lots of text (I1, I7, I10). Thorough documentation can make it difficult for consumers to keep the overall picture in mind (I1). In addition, the consumers want to solve a specific problem, and lengthy explanations lead them to scan the documentation in search of the aspects relevant to their concern (I1, I7, I10). Hence, providers should explain examples with the problem solution in mind, neglect too specific details (I12), and use short sentences that reduce the mental load for consumers (I1). Moreover, the explanation can reference more detailed information on specific topics (I10). In summary, I1 explained:

> *"Often, they [the documentations] also have too many details. The provider then tries to show which additional problem can be solved in full detail. That doesn't interest me, and I only scan the documentation."* (I1)

Finally, I13 stated that tutorials should be short.

F07: Tutorials should present usages following a storyline.

The interviewees agreed that tutorials should describe end-to-end use usages (I2, I9, I11) and that a story should guide the consumer through the tutorial (I11). A missing story can annoy consumers, as I9 explained:

> *"The second thing that bothers me is when it's totally mixed-up. Often, the tutorials simply lack a central theme. Sure, you still have the reference documentation, but it lacks the story of how to call something and when."* (I9)

F08: The documentation should highlight how explanations relate to the code.

Documentation should support developers to relate the explanation to the example code to enable easy switching between text and code (Meng et al., 2019). For example, the documentation can highlight method or parameter names by using the same color in the text and the code (Meng et al., 2019).

F09: The documentation should make it easy to separate example code from textual descriptions.

Documentation should clearly delineate example code and related explanations (Meng et al., 2019, 2020). For example, the API provider can split the documentation into a column with text and a second column with related examples (Meng et al., 2019, 2020).

F10: The documentation should highlight crucial aspects in code examples and explanations.

According to (Nasehi et al., 2012), API providers should highlight essential segments, e.g., the names of relevant API elements or design patterns, in the API documentation. In addition, the explanations should provide links to external resources with more information on these elements or patterns (Nasehi et al., 2012).

Similarly, I13 emphasized the importance of highlighting solution steps:

> *"What can also be an issue is if certain important things are not properly highlighted. [For example,] If you need to set a checkmark somewhere and that is not highlighted with a bold font."* (I13)

F11: Tutorials should be accompanied by suitable visualizations.

API providers should add visualizations to tutorials since text alone can be confusing, as I8 explained:

> *"It can't hurt to have a diagram for a simple scenario. Because a desert of text alone is often deterrent and incomprehensible."* (I8)

F12: Tutorials should have a consistent level of detail across all steps.

Inconsistencies in the level of detail of the description between tutorial steps show the consumers that the provider ran out of motivation to write the tutorial, as I13 described:

> *"I mean yes clearly [...] the quality of tutorials can also be a problem. If it is detailed, it should be consistently detailed. If it is initially very detailed and then abbreviated at some point, you notice that. Or if the complexity is not consistent because the writer did not feel like it. Such tutorials have no professional claim."* (I13)

F13: Tutorials should be structured to show the intermediate result after each step.

Intermediate results make it easier for consumers to follow a tutorial. Hence the documentation should provide intermediate results after each step in a tutorial (Inzunza et al., 2018).

Similarly, I13 stated:

> *"What we often use is that in the middle of the tutorial, you have a piece of text with which allows you to check whether you have done everything right so far."* (I13)

F14: Documentation should reuse compact and readable test cases as code examples.

Some research proposes to reuse existing test cases as code examples in API documentation (Hoffman and Strooper, 2000, 2003; Nasehi and Maurer, 2010). Test cases transfer information by describing valid and invalid requests to an API and corresponding expected responses (Hoffman and Strooper, 2000). An advantage of using test cases as code examples is that they guarantee consistency between code and documentation (Hoffman and Strooper, 2000). Also, the API provider saves effort by reusing existing test cases instead of manually developing examples exclusively for documentation purposes (Nasehi and Maurer, 2010).

Some interviewees agreed that using existing test cases as part of the documentation creates little effort for the API provider (I3, I4, I9). The reuse of test cases is especially suitable for Web APIs since the existing test cases present end-to-end scenarios from a consumers perspective (I4, I8). For instance, I8 argued:

> *"For me, this would not be a problem. I like to test a lot, so the only difference for me would be that you have to make these scenarios available to others. I don't even*

> *expect much extra work for this approach since most of the material and knowledge is already available."* (I8)

However, some experts are critical of publishing test cases as examples in the documentation. The interviewees feared that the provider inadvertently exposes security-relevant internal information (I7, I8). Hence, providers should carefully choose and only publish a subset of test cases to consumers.

Finally, I4 stated that using test cases for documentation purposes is a promising idea, but organizations will not adopt the approach unless an industry standard emerges. Similarly, I6 stated that a provider organization needs to define a governance process for publishing test cases as part of the documentation.

F15: Each example should comprise a concise unit of functionality that can be combined into more complex functionality.

Meng et al. (2018) and Nasehi et al. (2012) state that each example should be a small but detailed chunk that describes a piece of functionality. Similarly, Robillard and DeLine (2011) argue that each example should cover an independent functional intent. Hence, examples should not be too simple, i.e., presenting a simple API request involving just one method (Robillard and DeLine, 2011), since too simple examples might not be helpful for consumers (Nasehi and Maurer, 2010). At the same time, examples should not be too complex, making it difficult for a developer to understand them (Robillard and DeLine, 2011; Nykaza et al., 2002). Consumers can string together individual examples to realize more complex functionality (Nasehi et al., 2012).

One interviewee agreed that documentation should break complex functionality into smaller samples:

> *"Yes, it can't hurt to have a diagram for a simple scenario. Because a text desert alone is often daunting and incomprehensible. But of course, that depends on how complex the scenario is. With complex processes, for example, I think that you should divide it into smaller flows."* (I8)

F16: Documentation should present examples with different, e.g., increasing, levels of complexity to meet the needs of consumers with varying expertise.

Consumers learning a new API are likely to look for simple examples, whereas more experienced consumers might expect to find examples of higher complexity (Nasehi et al., 2012; Nykaza et al., 2002). I11 argued that the expected example complexity depends on the consumer's task:

> *"On the other hand, you can also download all the code right at the beginning and play around with it. [...] It depends on the context in which I use it. If I have to learn everything from scratch, then I use it [a sample application] quite often. But probably, in everyday life, if one wants to solve something specific, then I want only a*

> *certain part, and it would be perhaps even hindering if everything is firmly integrated and one cannot simply look up only a small thing."* (I11)

An option is to organize examples and tutorials so that they increase in complexity. For example, documentation can start with low complexity examples, e.g., examples only providing limited input options (I9, I12). Later on, the documentation should introduce more complex examples, going more into detail (I9). Hence, I12 explained:

> *"And speaking of tutorials, they should go from a low level of complexity to a higher level. Also, they should be full of examples."* (I12)

F17: API providers should design examples set in familiar domain contexts.

If examples use domains that most users are familiar with, it helps novices understand the functionality (Nasehi et al., 2012). A familiar domain context relieves API consumers from having to first learn a domain before being able to learn the API (Nasehi et al., 2012). Therefore, API providers should design examples set in domain contexts familiar to most consumers (Nasehi et al., 2012).

F18: Examples need to relate to the overall domain of the API.

Consumers need to understand the relationship between examples and the overall domain of the API, as I4 described:

> *"But the critical question about the scenarios is how they play together. By that, I mean, in particular, how they relate to the domain in the overall context. It may be that this [the example] is designed very well, but still, no one understands the context because the connections to the domain are simply missing. It also gives the impression that the API is incomplete."* (I4)

F19: Examples should evolve according to the consumers' needs.

API providers should not only update the documentation to reflect changes to the API, but they should also elicit what problems consumers face when using the API and its documentation and address potential issues in new versions of the documentation (Nasehi et al., 2012). Hence, the API documentation should also evolve to meet consumers' needs (Nasehi et al., 2012).

4.4. Discussion

First, we discuss findings derived from separating the best practice candidates into the categories knowledge and form. Generally, API documentation seeks to transfer knowledge about an API to consumers. Hence, the best practice candidates concerning form could potentially also seek to share some knowledge, even though they do not explicitly mention the type of knowledge.

For example, F08 states that *"The documentation should highlight how explanations relate to the code."*, which could aim at conveying knowledge on execution facts and API usage patterns. However, it is more challenging to map other best practice candidates concerning example form to components of API knowledge, e.g., the best practice candidate F19 stating that *"Examples should evolve according to the consumers' needs."* As a result, we present the following implication:

Implication 1: The effect of code examples on API consumer productivity and satisfaction depends not only on the knowledge that a code example transfers but also on its form.

Next, we look at the best practice candidates that describe knowledge that code examples should convey to consumers. We categorize the knowledge into knowledge related to *Domain Concepts*, *Execution Facts*, and *API Usage Patterns* according to the theory of *robust API knowledge* presented by Thayer et al. (2021).

Domain concepts describe concepts of a domain that exist outside of an API and which an API models. Domain knowledge is vital for API consumers since it enables them to understand what an API can do and how to manipulate API abstractions to achieve specific results (Thayer et al., 2021). Furthermore, domain knowledge improves API consumers' ability to search and recognize relevant code examples (Thayer et al., 2021; Ko and Riche, 2011). In the context of code examples, we identified only one best practice candidate concerned with transferring domain knowledge. The best practice candidate K01 states that required conceptual knowledge should accompany each code example. Hence, we derive the following implication:

Implication 2: Domain knowledge is not expected to be transferred to API consumers via code examples and therefore needs to be transferred in other documentation sections.

Execution facts are the simple rules that describe an API's runtime behavior, e.g., an API call's output and side effects given a specific input. These rules enable API consumers to write, test, debug and repair code, handle error messages, and reason about unexpected behavior (Thayer et al., 2021). We identified six best practice candidates that describe knowledge on execution facts that code examples should transfer. In some cases, the knowledge is embedded into the code example itself, e.g., HTTP headers, valid parameters, valid data types, and data formats (K02). In addition, knowledge of some execution facts should be part of the examples' descriptions, e.g., information on non-deterministic API behavior (K04) or reasons for error messages (K07). As a result, we present the following implication supporting the theory of robust API knowledge (Thayer et al., 2021):

Implication 3: Code examples should transfer knowledge on execution facts to API consumers.

API usage patterns describe how to combine and modify API interactions to realize specific outcomes and a rationale that allows consumers to modify the pattern to achieve different goals (Thayer et al., 2021). Since we focus on code examples, it is of little surprise that most knowledge-related best practice candidates that we identified capture knowledge about usage patterns (18 out of 48). Therefore, we present the following implication:

Implication 4: Code examples should transfer knowledge on usage patterns to API consumers.

In addition to best practice candidates aiming to transfer knowledge on the three categories of robust API knowledge, we identified two best practice candidates seeking to transfer meta-information about the example to the consumer. More precisely, the information enables the consumer to judge an example's reliability and manage their expectations. As a result, we derive the following implication:

Implication 5: API consumers look not only for API-related knowledge but also for information about the documentation quality to judge an example's reliability and manage their expectations.

4.5. Summary

For public, partner, and group Web APIs, the API provider has to transfer knowledge about an API to potentially unknown, heterogeneous, and distributed API consumers with different goals. Therefore, the success of a Web API can depend on the API documentation's ability to meet the consumers' information needs (Meng et al., 2018). Code examples are a vital element of API documentation (Nykaza et al., 2002; Meng et al., 2018, 2019; Robillard, 2009; Ko et al., 2007; Nasehi and Maurer, 2010; McLellan et al., 1998; Meng et al., 2020; Jeong et al., 2009; McLellan et al., 1998). Still, they have to meet specific quality criteria to unlock their potential (Meng et al., 2018; Robillard and DeLine, 2011; Robillard, 2009; Nykaza et al., 2002). Therefore, the goal of this chapter was to identify best practice candidates for code examples in public, partner, and group Web API documentation, i.e., in settings where the consumers do not have direct access to the Web API developers.

We reached this goal by identifying best practice candidates for code examples from literature and expert interviews. As part of the literature review, we identified 17 papers that present implications, principles, or observations concerning examples in general API documentation, from which we derived 32 best practice candidates. In addition, we conducted 13 expert interviews to enrich the existing and identify further potential best practice candidates. As a result, we presented a total of 48 best practice candidates. Furthermore, we categorized these best practice candidates into 27 best practice candidates that describe knowledge that the API provider should convey to API consumers following and extending the theory of robust API knowledge of Thayer et al. (2021). The remaining 19 best practice candidates describe characteristics of the presentation or the form the code examples should have to convey knowledge efficiently.

Finally, we analyzed the categorization of the best practice candidates and derive several implications. In general, we realized that the effect of code examples on API consumer productivity and satisfaction depends not only on the knowledge that a code example transfers but also on its form. In addition, code examples in Web API documentation are expected to transfer knowledge about execution facts and usage patterns, but not so much about domain concepts. Also, API consumers look for information about the quality of documentation to judge an example's reliability and manage their own expectations.

CHAPTER 5

Evaluation of Best Practices for Code Examples in Web API Documentation

This chapter aims to evaluate if a subset of the best practice candidates positively affects API consumers' productivity and perceived usability and are thus actual best practices. We do so by building on the results presented in Chapter 4 and extending the results presented in Bondel et al. (2022) and in the student thesis Cerit (2019).

In the remainder, we first present the research approach, followed by the quantitative and qualitative analysis of the results. Afterward, we discuss the analysis results and summarize the chapter.

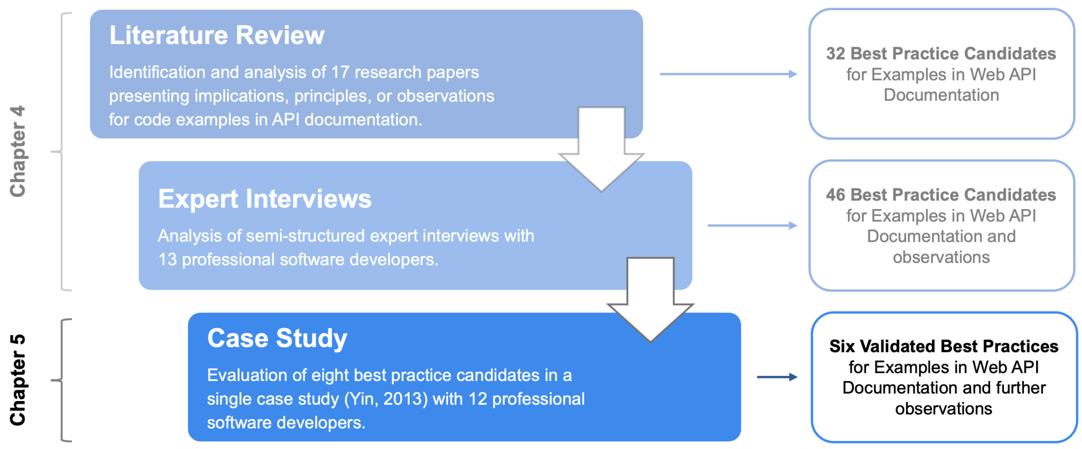

Chapter 4
Literature Review
Identification and analysis of 17 research papers presenting implications, principles, or observations for code examples in API documentation.
32 Best Practice Candidates for Examples in Web API Documentation
Expert Interviews
Analysis of semi-structured expert interviews with 13 professional software developers.
46 Best Practice Candidates for Examples in Web API Documentation and observations
Chapter 5
Case Study
Evaluation of eight best practice candidates in a single case study (Yin, 2013) with 12 professional software developers.
Six Validated Best Practices for Examples in Web API Documentation and further observations

Figure 5.1.: Steps and results of the research approach applied to evaluate best practice for Web API documentation. This chapter builds on the results presented in Chapter 4.

5.1. Research Approach

We present an overview of our research approach and results in Fig. 5.1. In this chapter, we aim to evaluate if a subset of best practice candidates for code examples previously identified in Chapter 4 are actually best practices in the context of public, partner, and group Web API documentation. A best practice candidate is validated as an actual best practice if it positively affects the API consumers' productivity or perceived usability. To reach this goal, we conducted an embedded single case study (Yin, 2013) with a large software vendor. The following describes the case study preparation, participants, and data collection.

Case Study Preparation

> "As a basis we create two different versions of documentation for the existing open-source system Compass[1]. The system enables the integration and monitoring of application landscapes consisting of internal applications running on specific computing clusters and external applications (SAP America, 2023). A user interacts with the system through a GraphQL[2] API."
>
> – (Bondel et al., 2022)

We visualize the overall structure of the documentation in Fig. 5.2.

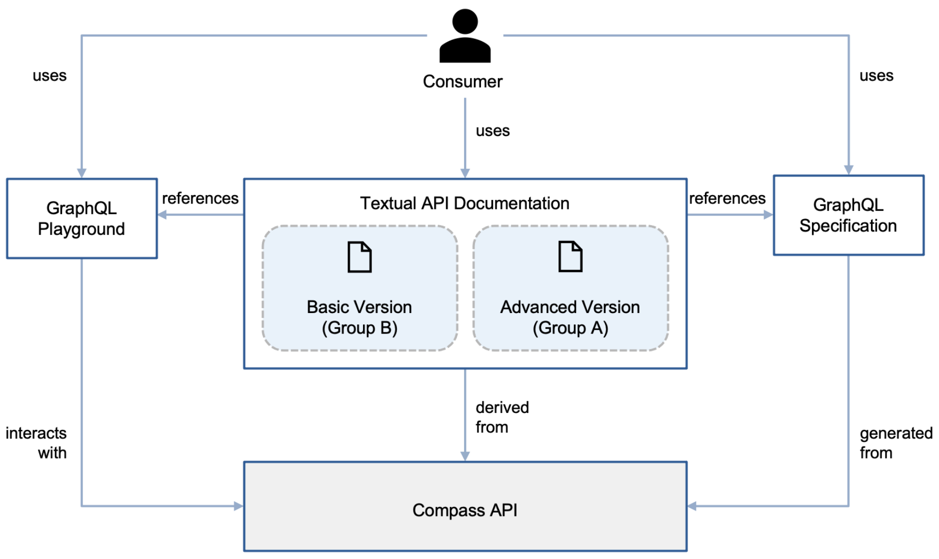


Figure 5.2.: Structure of the Web API documentation adapted from Cerit (2019).

"Overall, the documentation consists of three components, which are a textual de-

[1]https://github.com/kyma-incubator/compass
[2]https://graphql.org/

scription of the API including examples, a GraphQL playground[3], and an interactive specification generated with graphdoc[4]. The textual description references the GraphQL playground and the specification.

We focus on the textual description of the API since it entails the examples, and design two significantly different versions of it; a basic version and an advanced version. However, there is no consensus on the best structure of textual API documentation in literature or practice. Therefore, we derive the structure of the basic textual documentation from leading API management tool providers, e.g., MuleSoft[5] and Apigee[6] (Pillai et al., 2021), and practice-driven guides for technical writers (Johnson, 2023). As a result, we structure the textual documentation into an overview, getting started, tutorial, samples[7], and glossary part.

Since this research aims to analyze best practice candidates for code examples, the tutorial and sample section of the textual descriptions differ between the two versions. More specifically, we realize a set of eight best practice candidates in the tutorial and sample sections of the advanced version of the documentation, which we do not implement in the basic version. We chose the specific set of eight best practice candidates for one of three reasons. Either, previous literature does not mention or only weakly support a best practice candidate described by the interviewees. In addition, we chose best practices candidates for which we identified contradicting statements about their impact on the API consumers' productivity and perception of APIs. Finally, we selected best practices candidate, which we derived from literature that is not specific to public Web APIs."

– (Bondel et al., 2022)

We provide an overview of the eight best practice candidates that we evaluate in the case study in Tab. 5.1. The table also presents the reasoning for choosing each of these best practice candidates.

"The basic documentation enables API consumers to use the API by providing only necessary and common documentation sections. The tutorial of the basic documentation describes an exemplary usage scenario consisting of three code snippets of low complexity within a specific context. The samples are four independent code snippets covering essential ways to query Compass elements without context.

The advanced textual documentation builds on the basic documentation by implementing the best practices candidates described in the following.

First of all, the tutorial in the advanced documentation presents a "main scenario" that covers the typical use of the system (K21). Moreover, the tutorial integrates

[3]https://github.com/graphql/graphql-playground
[4]https://github.com/2fd/graphdoc
[5]https://www.mulesoft.com/
[6]https://cloud.google.com/apigee?hl=en
[7]We named the section "samples" since it is common to name a section containing code snippets "samples" or "examples" and we did not want to confuse the case study participants. However, following the definition of Robillard and DeLine (2011), the section contains code snippets.

Table 5.1.: Best practice candidates evaluated in the case study.

ID	Best Practice Candidate	Reason for Inclusion into the Case Study
K08	Examples should cover all common usage scenarios of an API.	This best practice candidate is supported by literature not specific to Web APIs (Nasehi and Maurer, 2010; Ko et al., 2004).
K19	Examples should demonstrate alternative solution approaches, including corner cases.	No literature supports this best practice candidate and interview partner I1 was critical of it.
K21	Tutorials should demonstrate the APIs basic functionality and common use cases.	This best practice candidate is supported by two research papers (Meng et al., 2018; Nykaza et al., 2002) of which only one is specific to Web API documentation (Meng et al., 2018). In comparison, several interview partners mentioned the importance of the best practice candidate (I5, I7, I8, I9).
K25	Example explanations need to explain how the example code relates to low-level API elements.	Only one Web API documentation specific research paper (Thayer et al., 2021) supports the best practice candidate.
F01	The documentation should enable the execution of examples within the documentation or common tools.	This best practice candidate is supported by two research papers (Meng et al., 2019; Jeong et al., 2009) of which only one is specific to Web API documentation (Meng et al., 2019). In comparison, several interview partners mentioned the importance of the best practice candidate (I2, I3, I6, I8, I9, I10, I11, I12).
F06	Code snippets, tutorials, and their explanations should be as concise and problem-oriented as possible.	Only literature not specific to Web APIs (Nasehi et al., 2012) supports this best practice candidate.
F07	Tutorials should present usages following a storyline.	Literature does not support this best practice candidate.
F16	Documentation should present examples with different, e.g., increasing, levels of complexity to meet the needs of consumers with varying expertise.	Only literature not specific to Web APIs (Nasehi et al., 2012; Nykaza et al., 2002) supports this best practice candidate.

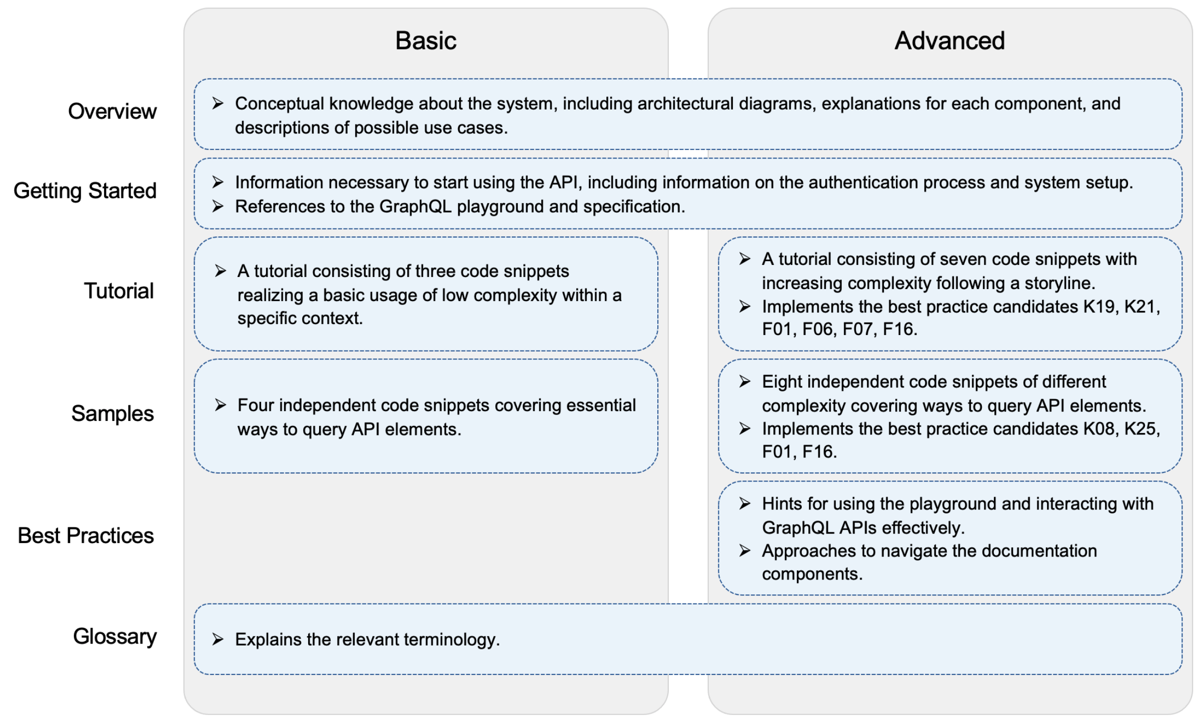

Basic
Advanced
Overview
➢ Conceptual knowledge about the system, including architectural diagrams, explanations for each component, and descriptions of possible use cases.
Getting Started
➢ Information necessary to start using the API, including information on the authentication process and system setup.
➢ References to the GraphQL playground and specification.
Tutorial
➢ A tutorial consisting of three code snippets realizing a basic usage of low complexity within a specific context.
➢ A tutorial consisting of seven code snippets with increasing complexity following a storyline.
➢ Implements the best practice candidates K19, K21, F01, F06, F07, F16.
Samples
➢ Four independent code snippets covering essential ways to query API elements.
➢ Eight independent code snippets of different complexity covering ways to query API elements.
➢ Implements the best practice candidates K08, K25, F01, F16.
Best Practices
➢ Hints for using the playground and interacting with GraphQL APIs effectively.
➢ Approaches to navigate the documentation components.
Glossary
➢ Explains the relevant terminology.

Figure 5.3.: Overview of the differences between the basic and the advanced textual documentation versions adapted from Cerit (2019).

a button that imports the complete tutorial into the Postman[8] application, thus enabling to easily try-out the examples (F01). The tutorial description is very concise and problem-oriented (F06). Furthermore, the tutorial entails seven code snippets with increasing complexity (F16). The tutorial also follows a storyline with a clear outcome (F07). Lastly, the tutorial presents alternatives to solve a problem, i.e., includes junctions in the solution path (K19).

Focusing on the samples, the code snippets in the advanced textual documentation frequently reference the specification (K25). Again, the samples provide integrated Postman tool support (F01). Another significant difference is that the samples cover a higher amount of usages by providing eight code snippets, including snippets with higher complexity (K08, F16).

In addition, the advanced textual description entails a "best practices" section. These best practices make it easier for users to use the playground, present beneficial approaches to navigate the documentation components, and provide hints on how to interact with GraphQL APIs."

– (Bondel et al., 2022)

[8]`https://www.postman.com/`

Table 5.2.: Overview of case study participants adopted from Bondel et al. (2022).

Group	ID	API Experience	Total Experience
Group A	A1	10	14
	A2	3	4
	A3	10	15
	A4	4	4
	A5	3	9
	A6	9	10
	Mean	**6.50**	**9.33**
Group B	B1	7	9
	B2	7	7
	B3	10	15
	B4	2	4
	B5	6	10
	B6	3	4
	Mean	**5.83**	**8.17**

We present an overview of the differences between the basic and the advanced textual documentation versions in Fig. 5.3.

Case Study Participants

> "Overall, 12 professional software developers from the industry partners organization participated in the evaluation. We admitted only participants with at least four years of professional experience into the study. We split the participants into two groups with the aim to balance the mean regarding years of experience across the groups and assigned each group one version of the documentation. An overview of the case study participants is provided in Tab. 5.2."
>
> – (Bondel et al., 2022)

Data Collection

Next, we present the steps of the case study which each participant passed through as presented in Fig. 5.4.

> "[...] We used multiple approaches to collect case study data, comprising observations, a SUS questionnaire, and open questions to collect quantitative and qualitative data.
>
> After starting the documentation application, setting up the GraphQL playground, and answering organizational questions, the case study setting allowed participants to first read and get acquainted with the assigned version of the API documentation. Next, we confronted the participant with three tasks. The tasks resemble real API

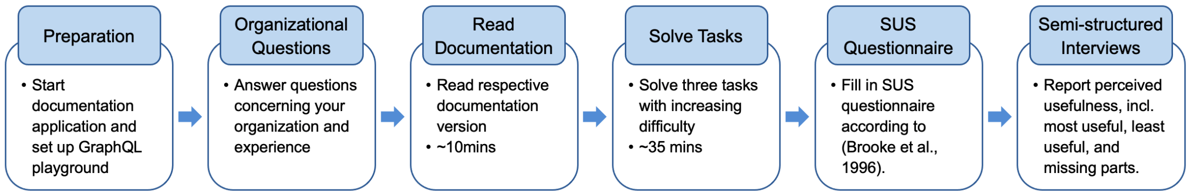

Preparation
• Start documentation application and set up GraphQL playground
Organizational Questions
• Answer questions concerning your organization and experience
Read Documentation
• Read respective documentation version
• ~10mins
Solve Tasks
• Solve three tasks with increasing difficulty
• ~35 mins
SUS Questionnaire
• Fill in SUS questionnaire according to (Brooke et al., 1996).
Semi-structured Interviews
• Report perceived usefulness, incl. most useful, least useful, and missing parts.

Figure 5.4.: Steps of the case study adapted from Cerit (2019).

usages, have increasing difficulty levels, and are solvable in a limited time. We encouraged the participants to express think-aloud comments. One researcher observed each participant and used a field protocol to capture the time, the number of issued API requests, and the usage of documentation sections during the task solution.

After solving or trying to solve the tasks, we asked the participant to fill in a SUS questionnaire according to Brooke (1996). Finally, we used open questions to inquire about the perceived usefulness of the documentation, including the most useful, least useful, and missing parts of the documentation. The whole evaluation process was audio recorded."

– (Bondel et al., 2022)

5.2. Evaluation of Best Practices for Examples in Web API Documentation

Next, we analyzed the case study data to evaluate the effect of applying the eight best practice candidates K08, K19, K21, K25, F01, F06, F07, and F16 to the code examples in public, partner, and group Web API documentation, i.e., in settings in which the consumers usually do not have access to the Web API developer team. Specifically, we focused on the best practice candidates' impact on the developers' productivity and perceived usability. First, we describe the findings of the quantitative analysis followed by the results of the qualitative analysis. In addition, we summarize the quantitative and qualitative results specific to each best practice candidate to evaluate which of the candidates are actually best practices.

5.2.1. Quantitative Analysis

"The quantitative analysis comprises metrics on the needed time, success, and documentation feature usage observed during the evaluation. In addition, we present the results of the SUS survey. The sample size for all presented statistics is n = 12.

First, we analyze the time that participants took to initially read the documentation and to solve all tasks as presented in Fig. 5.5. The analysis shows that, overall, participants of group A needed on average 43:10 minutes whereas group B required

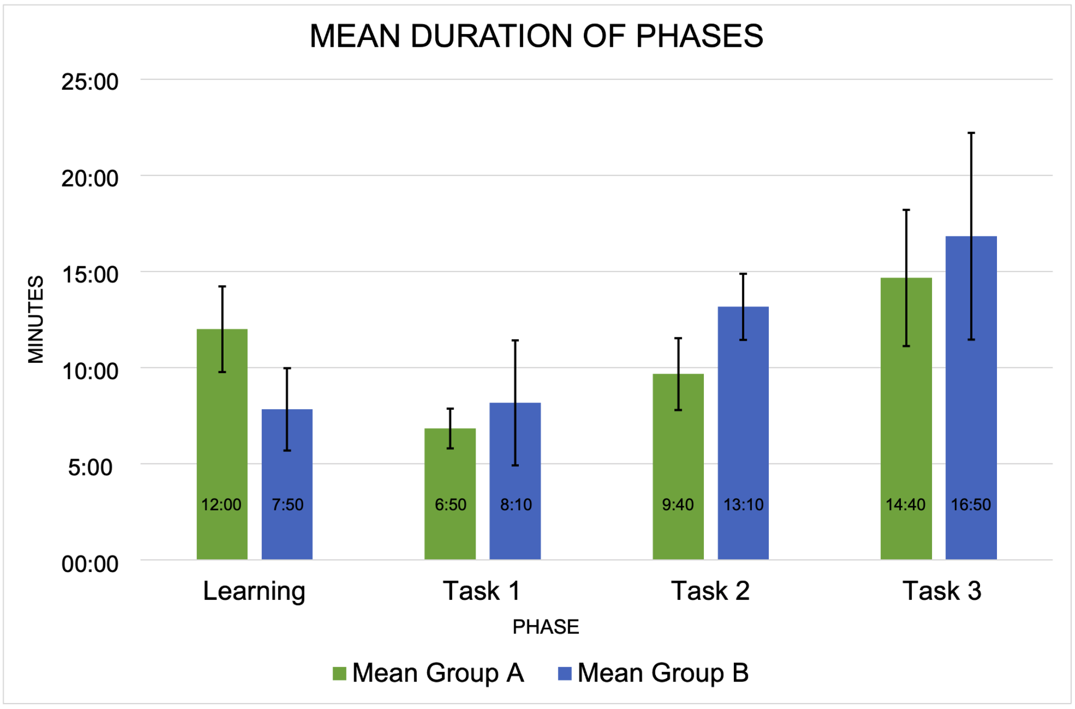


Figure 5.5.: Time that participants took for learning and solving the three tasks adapted from Bondel et al. (2022).

46:00 minutes. Moreover, we observe that while group A spend more time learning about the API, they solved all tasks faster than group B. However, the improvement in time is only statistically significant for the more complex tasks 2 (t=2.29, df=10, p=0.05, one-tailed) and 3 (t=1.91, df=10, p=0.05, one-tailed).

Next, we analyzed the success of the participants solving the tasks. We assigned points to the level of achievement of solutions to do so. A participant receives one point for a solved task, a half-point for an unfinished task with the right approach, and no points for wrong solutions. Group A reached on average 2.92 points and group B reaches 2.83 points. Hence, the assessment yields, that both groups are very successful in solving all three tasks. As a result, we cannot derive any significant differences with regards to the success of solving the tasks between the groups.

We also observed the number of API requests that the participants issue to Compass and their success. Overall, the participants made between 12 and 16 requests to the application during the study. On average, the rate of successful requests is 61% for group A and 39% for group B. Hence, the request success rate for members of group A is significantly higher compared to group B (t=5.66, df=10, p=0.05, one-tailed). Also, the standard deviation differs between the groups with group A yielding a standard deviation of 0.019 and group B with 0.084. Hence, the success rate of members of the group B varied more.

Furthermore, we noted down the approximate fractions of time that each participant spent on the documentation's different sections. In Fig. 5.6, we visualize the relative usage of each documentation section for each group. We observe that group B used the getting started guide more often than group A. However, the members of group A spend more time in the tutorial and best practices sections. Group A also shows higher variations across most sections, indicating that participants value the documentation sections differently.

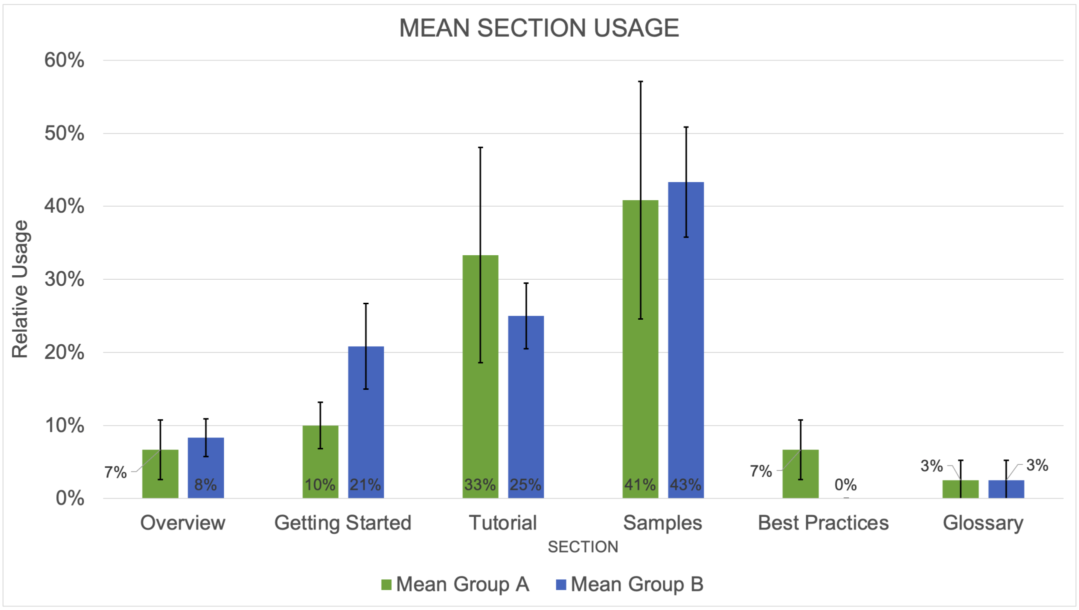


Figure 5.6.: Relative amount of times the participants consulted specific documentation sections adapted from Bondel et al. (2022).

Moreover, the participants rated the advanced documentation with an average SUS score of 85.8 points compared to 75 points for the basic documentation. Hence, the software developers perceive the usability of the advanced documentation as better.

– (Bondel et al., 2022)

We present an overview of the results of the SUS analysis in Tab. 5.3.

"Finally, we analyzed the data for correlation between variables. First, we identified a moderately strong correlation (r = 0.79) between the request success rates and the duration of the learning phase, indicating that investment into learning the API results in a higher probability of making successful API requests. In addition, we see a moderate correlation (r = 0.66) between the success rate and the perceived usability measured with the SUS questionnaire. Interestingly, the participants' API and software engineering experience does not seem to correlate with the API request success rate (r = 0.25, r = 0.32)."

– (Bondel et al., 2022)

Overall, the productivity of developers with the advanced documentation version was better compared to developers with access to the basic documentation. Developers of group A solved more complex tasks faster and had a higher API request success rate. Also, they perceived the usability of the documentation as better compared to group B. Nonetheless, both groups were equally successful in solving all tasks.

Table 5.3.: SUS score (Brooke, 1996) of the different documentation versions.

Particiant	**SUS Score**	**Participant**	**SUS Score**
A1	75	B1	85
A2	90	B2	75
A3	87,5	B3	80
A4	92,5	B4	77,5
A5	77,5	B5	80
A6	92,5	B6	52,5
Mean	**85,83**	**Mean**	**75**

5.2.2. Qualitative Analysis

"We derive the qualitative results from the think-aloud protocol and semi-structured interviews with the case study participants. We describe our observations along the questions we asked the participants. Since the participants discussed the documentation in general, we also report observations not related exclusively to examples.

First, we asked the participants if they perceived the API documentation as helpful in solving the tasks. Five participants answered that they do not think of the overview section as beneficial for the case study tasks but it might be helpful for very complex cases or tasks aiming at integration with 3rd party applications (A2, A4; B3, B4, B5). In addition, the API specification is useful for looking up more detailed information and targeted problem solving (A2; B5, B6). Also, the execution environment was considered "vital" (B5) and helped participants gain confidence and experience using the API (A2, A3). Both groups used the samples as a lookup point, but group B saw the samples as the main entry point (B1, B2, B5, B6). The tutorial section was perceived as helpful by participants of group A because they could solve tasks by merely adjusting the code examples (A1, A2). Moreover, the advanced documentation puts the code snippets in the tutorial into a context, and therefore the participants expect it to help with real implementation tasks (A4, A5). However, group B did rarely comment on the tutorials. Finally, participants of group A commented that the tool support for Postman might only be helpful if the API consumers are Postman "power users" (A3, A5).

Next, we asked what parts of the documentation the participants liked the most and why. All respondents stated that the specification and playground were among their favorite parts of the documentation (A1, A2, A3, A4, A5, A6; B1, B2, B3, B4, B5, B6). The specification supported the participants with clickable elements (A1, A2), a search function (B3, B4), and a structured presentation (B5). The reason for the positive perception of the playground was that it allowed consumers to try out code snippets and tutorials (A1, A2, A4; B4, B6). Group A also liked the tutorials, primarily because of the many references to other resources included in the advanced version (A1, A2, A5). Moreover, participants of group A stated that they liked the coverage of usages with examples (A3, A4). In comparison, participants of group B appreciated the samples the most (B1, B2, B4) since they provided an entry point into solving the tasks (B1, B2). At last, two participants of group A positively mentioned the best practices section, explaining that such information is often missing in documentation (A4, A5). The best practices were only available to group A.

The third question we asked the case study participants was which parts of the documentation they did not like at all and why. All participants agreed that the overview and glossary were the most useless sections because they presented too much information that was unnecessary for solving the tasks (A2, A3, B2). However, the provided conceptual information might be necessary for more complex scenarios or non-technical stakeholders (A4; B4). Also, the participants criticized that we did not highlight important aspects of the text in the documentation enough (A5; B2, B5). A complaint specific to group B was that the descriptions of the samples and the tutorial were too long (B1, B6). In general, the participants of group B perceived the documentation as not "developer-centric" enough (B3, B4).

Lastly, we asked the participants which features they missed the most in the documentation. Several participants mentioned the lack of "helper buttons", e.g., buttons that automatically copy a code snippet or directly transfer snippets into the playground for execution (A5; B2, B5). Also, one participant complained about missing information on prerequisites (A4). Most statements about missing features came from group B. Most importantly, they requested more samples with higher complexity (B1, B2, B6). Also, group B participants criticized that they lacked links between resources, which required them to conduct more searches (B2, B6). Finally, one participant requested more concise and practical descriptions of the tutorial and the samples (B4)."

– (Bondel et al., 2022)

5.2.3. Evaluation of the Best Practice Candidates

This section reviews the quantitative and qualitative case study results to evaluate which of the investigated eight best practices candidates can be confirmed to be actual best practices for code examples in official public Web API documentation. A best practice candidate is validated as an actual best practice if it positively affects the API consumers' productivity or perceived usability. However, we cannot determine any direct correlation, let alone causality, between the implementation of specific best practice candidates and the improvement of the API consumers' productivity or perceived usability since we apply all best practice candidates simultaneously.

Therefore, the following results are based on indications derived from observations, not hard evidence.

K08: Examples should cover all common usage scenarios of an API.

The best practice candidate K08 is only mentioned in literature not specific to Web APIs (Nasehi and Maurer, 2010; Ko et al., 2004). In the advanced textual documentation version, the sample section provides eight code snippets covering all usages required to solve the use case tasks. In comparison, the basic documentation version only provides four code snippets covering essential ways to query the API elements. The qualitative case study evaluation revealed that the participants of group A explicitly stated that they liked the coverage of usages with examples (A3, A4). Nevertheless, at the same time, several participants of group B declared the reduced number of samples as the most useful part of the documentation (B1, B2, B4) as they provide an entry point into solving the tasks (B1, B2).

In summary, both participant groups perceived the code snippets in the sample section as useful. A reason for group B to value the sample section highly could be the lack of other resources providing entry points for task solutions. An alternative explanation could be that code snippets that do not cover a consumer's exact use case still transfer enough knowledge to be helpful to a consumer. Hence, it is not clear if the implementation of the best practice candidate improved the productivity or perceived usability of group A compared to group B. Therefore, we cannot confirm the best practice candidate K08 to be a best practice for code examples in official public, partner, and group Web API documentation.

K19: Examples should demonstrate alternative solution approaches, including corner cases.

No literature source mentions the best practice candidate K19, but the interviewees I2, I8, I11, and I12 emphasized its importance. However, the interviewee I1 was critical of its added value. I1 argued that API documentation should only present the best solution to realize a specific use case to relieve the consumer from choosing between approaches. The tutorial in the advanced documentation version showed junctions in the solution path. However, no group A participant mentioned the junctions' usefulness, and no participant in group B complained about missing alternative paths.

A potential reason why the case study participants did not mention the best practice candidate could be the simplicity of the tasks. Since the tasks did not provide any context that would make it necessary to choose between different solution approaches, the participants were not required to select between solutions. Hence, we have no indication that the best practice candidate K19 improved the productivity or perceived usability of members of group A. Therefore, we cannot confirm the best practice candidate to be a best practice.

K21: Tutorials should demonstrate the API's basic functionality and common use cases.

The best practice candidate is mentioned in one Web API specific (Meng et al., 2018) and one general API documentation related (Nykaza et al., 2002) literature source. In addition, several interviewees (I5, I7, I8, I9) agreed on the importance of the best practice candidate K21. While the basic documentation presented one exemplary usage scenario consisting of three code snippets within a specific context, the advanced tutorial guided the consumers through the typical use of the system. Two participants of group A (A1, A2) explained that the tutorial was handy since they could solve the case study tasks by merely adjusting the code examples. On the other hand, participants of group B did not comment on the tutorials but used the sample section as the main entry point to solve tasks (B1, B2, B5, B6).

As a result, we see that members of group A valued the tutorial more compared to group B members. Therefore, our case study confirms the best practice candidate K21 to be a best practice for examples in Web API documentation as previously described in Meng et al. (2018).

K25: Example explanations need to explain how the example code relates to low-level API elements.

The best practice candidate K25 is previously described in one Web API specific literature resource (Thayer et al., 2021). We realized this best practice candidate in the advanced documentation version by adding frequent links to the sample and tutorial descriptions pointing at the specification. Several group A participants reported that they valued these references (A1, A2, A5). On the other hand, group B participants criticized the lack of links between resources, which required them to conduct more searches (B2, B6).

Hence, we confirm the best practice candidate K25 to be a best practice for examples in Web API documentation in accordance with Thayer et al. (2021).

F01: The documentation should enable the execution of examples within the documentation or common tools.

One Web API specific (Meng et al., 2018) and one general API documentation (Jeong et al., 2009) literature source mention the best practice candidate F01, while many interviewees (I2, I3, I6, I8, I9, I10, I11, I12) emphasized its importance. Both the basic and advanced documentation versions enabled access to the GraphQL playground. Members of both groups perceived the execution environment as one of the favorite features of the documentation since it allowed them to quickly try out code snippets to gain experience with the API (A1, A2, A3, A4; B4, B5, B6). Nevertheless, some participants also mentioned that they missed "helper buttons," which automatically copy code snippets into the playground (A5; B2, B5).

In addition, the advanced documentation version enabled automated imports of code snippets from the sample and tutorial section into the Postman[9] application. However, not all members of group A noticed the option of using Postman and no one used it to solve a task. In addition,

[9] `https://www.postman.com/`

some participants of group A commented that Postman tool support might only be helpful if the API consumers are Postman "power users" (A3, A5).

Even though we intended to evaluate F01 with the automated import of code snippets into Postman, we realized we already provided the capability to test code examples in both documentation versions with the GraphQL playground. Due to this circumstance, we could not observe a difference between the groups with access to different documentation versions. However, the overwhelmingly positive feedback of case study participants regarding the GraphQL playground confirms the best practice candidate C01 to be a best practice for examples in Web API documentation as previously described in Meng et al. (2018). However, we must note that even the need to copy code from the documentation to the playground was perceived negatively. Thus, consumers prefer try-out functions that allow them to execute the code with as few hurdles as possible. Additionally, an API provider should only provide specific tool support, e.g., for Postman, if the particular API consumer community commonly uses the tool.

F06: Code snippets, tutorials, and their explanations should be as concise and problem-oriented as possible.

The best practice candidate F06 is supported only by literature not specific to Web APIs (Nasehi et al., 2012). While the tutorial description in the basic documentation version was rather long, the tutorial description in the advanced documentation was concise and problem-oriented. As a result, several group B members complained about long and not developer-centric tutorial descriptions (B1, B3, B4, B6).

Therefore, we confirm the best practice candidate C06 to be a best practice for examples in public, partner, and group Web API documentation.

F07: Tutorials should present usages following a storyline.

No literature supports the best practice candidate F07. We adopted the tutorial in the advanced documentation version to follow a storyline with a clear outcome. On the other hand, the basic documentation version did not explicitly mention the outcome of the tutorial, and the link between the code snippets was somewhat blurred. Consequently, two members of group A stated that the storyline of the tutorial adds value since they expect it to help with actual implementation tasks (A4, A5). Also, participants of group B reported that the tutorial descriptions were not developer-centric enough (B3, B4).

As a result, we confirm the best practice candidate F07 to be a best practice for examples in Web in public, partner, and group Web API documentation.

F16: Documentation should present examples with different, e.g., increasing, levels of complexity to meet the needs of consumers with varying expertise.

The best practice candidate F16 is supported only by literature not specific to Web API documentation (Nasehi et al., 2012; Nykaza et al., 2002). The tutorial in the basic documentation version entailed three code snippets with a relatively low level of complexity. Similarly, the sample section consisted of four independent code snippets of similar, low complexity. In comparison, in the advanced documentation version, the tutorial comprised seven and the sample section eight code snippets with increasing complexity. While group A members did not positively mention the different levels of complexity of the code snippets, several participants of group B complained about the lack of snippets with higher complexity (B1, B2, B6).

Therefore, we confirm the best practice candidate F16 to be a best practice for examples in Web API documentation.

In summary, we confirm the best practice candidates K21, K25, F01, F06, F07, and F16 to be actual best practices for examples in public, partner, and group Web API documentation. In addition, the best practice candidates K08 and K19 need further evaluation and cannot be confirmed as best practices for examples in public, partner, and group Web API documentation based on our case study results.

5.3. Discussion

First, we pick up on implication 2 presented in Sec. 4.4. Implication 2 states, *"Domain knowledge is not expected to be transferred to API consumers via code examples and therefore needs to be transferred via other documentation sections."* These other documentation sections could be the documentation introduction, system overviews, or glossaries. However, the case study results show that the participants did not think of the overview section and glossary as helpful either (A2, A4; B3, B4, B5) because they presented too much information that was unnecessary for solving the case study tasks (A2, A3; B2). A potential reason could be that all participants were already acquainted with the API's domain. If a developer already knows a domain, they could perceive additional domain information in code example descriptions as clutter. In this case, the best practice candidate K01, stating that conceptual knowledge should accompany code examples, would conflict with the best practice C06, which says that examples and their explanations should be concise and problem-oriented (F06). Moreover, due to time constraints, the case study tasks were of low and medium complexity. Hence, we assume that domain knowledge as part of examples or in other documentation sections is only relevant for API consumers if they do not know the domain or the domain or task is of high complexity. This assumption is supported by case study participants who stated that the overview in the case study documentation might be helpful for very complex cases or tasks aiming at integration with 3rd party applications (A2, A4; B3, B4, B5). Hence, we revise the existing implication 2:

Implication 2 (revised): The perceived value of conceptual knowledge as part of code examples or documentation depends on the API consumers' pre-existing knowledge of the domain and the task complexity.

In addition to the best practice candidate K01 and the best practice F06, we realized that the applicability of other best practices and best practice candidates might also depend on contextual factors. For example, the best practice candidate K19 stating that "Examples should demonstrate alternative solution approaches, including corner cases" might only be relevant for more complex tasks. Also, an API provider should only present a Postman integration if the particular API consumer community commonly uses the tool. Thus, we present a more general implication:

Implication 6: The usefulness of specific best practices or best practice candidates for code examples in public, partner, and group Web API documentation depends on the context in which they are applied.

Next, we look at the effect of different documentation version on different developer personas and learning styles. Stylos and Clarke (2007)[10] present three archetypical software engineering *personas*. A persona captures and categorizes work styles, characteristics and motivations of developers approaching a task. The first persona is the *systematic* developer, who works in a top-down manner, i.e., wants to understand a system as a whole before focusing on a specific task. On the other side of the spectrum are *opportunistic* developers, who approach a task bottom-up and are primarily concerned with solving a task as fast as possible. Finally, there is the *pragmatic* developer, who leverages a mix of the systematic and the opportunistic approach. Pragmatic developers first approach a task bottom-up but switch to a top-down approach in case they get stuck.

Meng et al. (2019) and Meng et al. (2018) take up the concept of different user personas, and tailor it to API consumers. Meng et al. (2018) identifies two learning strategies that resemble the systematic and the opportunistic developer personas. First, the systematic developer persona uses a *concepts-oriented* learning strategy. The concepts-oriented learning strategy is characterized by systematic searches and regular consultation of official documentation. These developers invest time to create a deeper understanding of an API before starting to solve a task. Systematic developers like to use technical architecture overviews, working through getting started sections, tutorials, and implementing small projects before solving a specific task.

On the other hand, the opportunistic developer persona employs a *code-oriented* learning strategy. Such opportunistic developers like to solve tasks quickly in a bottom-up manner. They are task-driven and like to start coding as fast as possible, often using a trial-and-error approach. These developers do not spent time emerging themselves in conceptual knowledge and stop learning about an API as soon as they solve their task (Meng et al., 2019, 2018).

Our case study identified hints of developers using both learning strategies. For example, A1 stated that conceptual information provides you with a *"third eye"* onto the whole API, and it should be part of the documentation even if it does not directly support the solution of your task. In comparison, A4 explained that the overview section was irrelevant and too long. Nevertheless, this kind of information might be relevant for more complex systems or debugging (A4). Hence, conceptual information is only relevant if the information is required to solve a specific task. However, we did not collect data to categorize preferred learning strategies systematically.

Since these two learning strategies for APIs exist, documentation should address the information

[10]Stylos and Clarke (2007) expands the concept of personas citing Clarke (2004). However, Clarke (2004) is not accessible to us.

needs of both (Meng et al., 2020, 2019, 2018). The quantitative results of our case study show higher variations in the usage of documentation sections for group A. In comparison, most of group B valued the sample section most and used it as the entry point to solving the case study tasks (B1, B2, B5, B6). Furthermore, group A members showed a consistently higher rate of successful requests to the API compared to members of group B. Hence, we assume that the advanced documentation accommodates more learning styles compared to the basic version. Moreover, we assume that insufficient documentation forces all developers to use a code-oriented learning approach. An observation supporting this hypothesis is the higher standard deviation of the rate of successful requests of group B, which could indicate that these developers had to rely more on experimentation to solve a task. Hence, we present the following implication:

Implication 7: Low quality code examples in public, partner, and group Web API documentation force developers to use a trial-and-error approach, thus hampering productivity and perceived satisfaction of developers with a systemic learning type.

While this study focuses on code examples specifically, other parts of the documentation also have major influence on the learnability of APIs. First of all, developers need to have knowledge of an API beyond examples, otherwise they might have issues with identifying relevant examples or modifying them (Thayer et al., 2021). Furthermore, all participants of the case study emphasized the value of the specification and playground as part of the documentation. However, similar to code examples, these documentation parts also have to meet certain quality criteria to be of value for API consumers, e.g., specifications should have a structured presentation and be searchable. Hence, we present the following implication:

Implication 8: The specification and playground are essential elements of public, partner, and group Web API documentation, but they need to meet certain quality criteria to be of value for API consumers.

5.4. Summary

In the previous Chapter 4, we presented 46 best practice candidates for code examples in public, partner, and group Web API documentation. However, these best practice candidates have not yet been evaluated in the context of public, partner, and group Web APIs, i.e., settings where consumers usually do not have direct access to the Web API developers. Therefore, this chapter aimed to evaluate a subset of best practice candidates to see if they are actually best practices.

To reach this goal, we performed a case study. We chose eight best practice candidates, K08, K19, K21, K25, F01, F06, F07, and F16, for evaluation since they have little or no support in Web API-specific literature or there are contradicting statements about their impact on the API consumers' productivity and perceived usability. As a basis for the case study, we chose an existing public GraphQL API and created two versions of documentation for the API: a basic version and an advanced version implementing the chosen best practice candidates. Afterward, we split a group of 12 professional developers into two equally sized groups with access to either documentation versions and observed them while solving three tasks using the Web API. In addition, we collected data using a SUS questionnaire and semi-structured interviews.

Overall, we observed that participants with access to the advanced documentation spent more time initially reading the documentation but solved all tasks faster. In addition, members of this group had a higher rate of successful requests and perceived the documentation as more usable than participants with access to the basic version. Also, the usage of different documentation sections varied more for participants using the advanced documentation. However, the participants of both groups had similar overall success in solving the tasks.

We additionally analyzed indications of a positive effect of each best practice candidate on the developers' productivity and perceived usability in the quantitative and qualitative analysis. The results showed that we can confirm the six best practice candidates, K21, K25, C01, C06, C07, and C16, to be actual best practices. In comparison, the observations for K08 and K19 were not clear or contradicting.

Finally, we analyzed the case study observations, leading to the presentation of three new implications and the revision of a previously presented implication. First, one new implication states that low-quality code examples in open Web API documentation force developers to use a trial-and-error approach, which is typical for opportunistic developer personas. Thus, such low-quality code examples can hamper systematic developer personas' productivity and perceived satisfaction. Secondly, not only the code examples are essential aspects of Web API documentation, but also the API specification and the API playground. Nevertheless, the specification and playground must also meet specific quality criteria to be of value for API consumers.

In addition, we revised implication 2 (see Sec.4.4), which states that developers do not expect code examples to transfer domain knowledge. Instead, we realized that the perceived value of conceptual knowledge, as part of code examples or other documentation sections, depends on contextual factors. Such contextual factors are, e.g., the API consumers' pre-existing domain knowledge and a task's complexity. In addition, the effect of other best practices or best practice candidates for code examples in public, partner, or group Web API documentation also depends on the context in which they are applied. Consequently, future work should evaluate the impact of best practices and best practice candidates under different contextual circumstances.

These findings show that best practices should explicitly describe the circumstances under which they are applicable. Therefore, in the next chapter, we will present patterns.

CHAPTER 6

Design of the API Management Pattern Catalog (AMPC)

As indicated in the previous Chapter 5, Web API management best practices need to consider the context in which they are applied. Therefore, in this chapter, we describe the design of Web API management patterns focusing on collaboration for public, partner, and group Web API initiatives. More precisely, we describe the design of the *API Management Pattern Catalog for Public, Partner, and Group Web APIs with a Focus on Collaboration* (Bondel and Matthes, 2023), which we abbreviate as *API Management Pattern Catalog (AMPC).* We do so by extending the results previously presented in Bondel et al. (2021b) and the student thesis Landgraf (2021). In the following, we describe the research approach. Afterward, we detail the iterative improvement of the AMPC. Also, we outline the major contents of the AMPC. Finally, we conclude the chapter with a summary.

6.1. Research Approach

Hevner et al. (2004) and Hevner (2007) present a conceptual framework for design science research in IS. The basic principle of design science is to create new knowledge about an unsolved problem and its solution by iteratively building and evaluating a solution artifact (Hevner et al., 2004). Artifacts can be constructs, models, instantiations, or methods (Hevner et al., 2004). Methods comprise best practice approaches (Hevner et al., 2004), e.g., patterns. Hence, we apply a design science research approach according to Hevner et al. (2004) and Hevner (2007) to guide the creation of the AMPC as a solution artifact as visualized in Fig. 6.1. Hevner et al. (2004) defines seven guidelines that ensure that a research endeavor meets the requirements of design science. We discuss the realization of these guidelines in the following.

Problem Relevance. The artifact needs to present or support the solution to a relevant and unsolved business problem in an application domain (Hevner et al., 2004). As discussed in

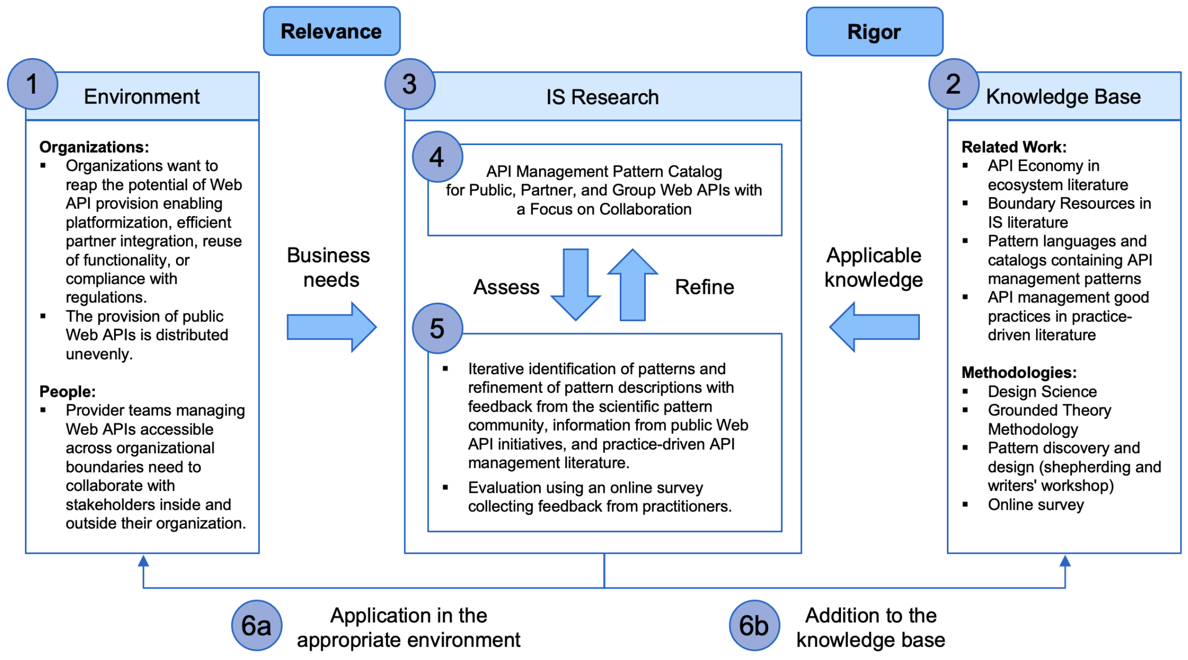

Relevance
Rigor
1
Environment
Organizations:
Organizations want to reap the potential of Web API provision enabling platformization, efficient partner integration, reuse of functionality, or compliance with regulations.
The provision of public Web APIs is distributed unevenly.
People:
Provider teams managing Web APIs accessible across organizational boundaries need to collaborate with stakeholders inside and outside their organization.
Business needs
3
IS Research
4
API Management Pattern Catalog for Public, Partner, and Group Web APIs with a Focus on Collaboration
Assess
Refine
5
Iterative identification of patterns and refinement of pattern descriptions with feedback from the scientific pattern community, information from public Web API initiatives, and practice-driven API management literature.
Evaluation using an online survey collecting feedback from practitioners.
Applicable knowledge
2
Knowledge Base
Related Work:
API Economy in ecosystem literature
Boundary Resources in IS literature
Pattern languages and catalogs containing API management patterns
API management good practices in practice-driven literature
Methodologies:
Design Science
Grounded Theory Methodology
Pattern discovery and design (shepherding and writers' workshop)
Online survey
6a
Application in the appropriate environment
6b
Addition to the knowledge base

Figure 6.1.: Design science research framework (Hevner et al., 2004) adapted to the creation of the AMPC (Bondel and Matthes, 2023) adapted from Bondel et al. (2021b).

Chapter 1 and summarized in Fig. 6.1 (#1), providing public, partner, and group Web APIs creates profits through the realization of new business models (Evans and Basole, 2016; Basole, 2016, 2019), platform creation (Ghazawneh and Henfridsson, 2010, 2013; Eaton et al., 2015; Karhu et al., 2018; de Reuver et al., 2018), efficient partner integration (Hagel III and Brown, 2001), or compliance (Bondel et al., 2021a; ISO 20077-1; ISO 20078-1; ISO 20080). However, relatively young digital organizations in the US currently dominate the provision of successful Web APIs (Evans and Basole, 2016; Basole, 2019; Huhtamäki et al., 2017). Moreover, Web API provision requires careful design (Yoo et al., 2010) and management, taking into account the collaboration with stakeholders inside and outside (Islind et al., 2016) of the provider organization. Nevertheless, knowledge about API management is distributed across mostly practitioner- and vendor-driven literature. Hence, the AMPC aims to provide a holistic approach to API management from an API provider perspective. To successfully contribute to practice and research, the AMPC should provide patterns applicable to real-world settings, be comprehensible and usable for practitioners, present correct information, and capture relevant practices completely.

Research Rigor. Design science research needs to be rigorous, i.e., the researchers need to apply existing foundational knowledge and methodologies during the design and evaluation of the artifact (Hevner et al., 2004; Hevner, 2007). As visualized in Fig. 6.1 (#2), we reviewed IS literature to identify relevant foundational knowledge on the management of Web APIs across organizational boundaries. First, we identified and examined trends in the API Economy (see Section 2.2) and IS literature on platforms focusing on boundary resources (see Section 2.3). Moreover, we inspected existing pattern languages, catalogs, and collections concerned with APIs and interfaces (see Chapter 3) as well as practice-driven literature on API management best practices, i.e., De (2017), Medjaoui et al. (2018), Jacobson et al. (2012) and Spichale (2017).

Regarding methodologies, we thoroughly applied design science (Hevner et al., 2004; Hevner, 2007) as an overall approach to the AMPC design. Additionally, we used GTM (Wiesche et al., 2017) to identify the pattern candidates and stakeholders in the case base. Also, we reviewed literature concerning pattern discovery and design (Alexander, 1973; Alexander et al., 1977; Gamma et al., 1995; Coplien, 1996; Buschmann et al., 1996; Buckl et al., 2013). As part of the iterative improvement of the design artifact, we participated in shepherding and a pattern writers workshop, a standard and long-standing approach for pattern improvement in the pattern community (Coplien, 1996). Finally, we used an online survey to evaluate the AMPC from a practitioner's point of view.

Design as a Search Process. Design science prescribes the iterative discovery and improvement of an artifact until it presents a satisfactory solution to a problem (Hevner et al., 2004). As indicated in Fig. 6.1 (#3) and detailed in Fig. 6.2, we iteratively evolved the AMPC. We build on the data collection, analysis, and initial pattern collection presented in the student thesis Landgraf (2021) (see Section 6.2). We conducted a second analysis after restricting the data basis (see Section 6.3.1). In addition, we enriched the pattern descriptions with information derived from successful public API initiatives and practice-driven API management literature (see Section 6.3.2). Simultaneously, we evaluated a subset of pattern descriptions from the perspective of the scientific pattern community by participating in shepherding and a writers workshop (Coplien, 1996) and continuously incorporated the feedback into the creation of the AMPC (see Section 6.3.3). These activities were executed iteratively, resulting in the publication of intermediate results in Bondel et al. (2021b)

Finally, we evaluated the applicability, comprehensibility and usability, correctness, and completeness of the AMPC from a practitioner's viewpoint using an online survey (see Chapter 7). After the evaluation but before its publication we incorporated further minor changes as detailed in Appendix D.

Furthermore, we list measures to further evolve the AMPC derived from the feedback of the pattern community, the survey responses, and pattern literature (see Section 8.3).

Design as an Artifact. The goal of design science is to create a purposeful artifact (Hevner et al., 2004). The artifact can be an instantiated information system, a construct, a model, or a method supporting the implementation of information systems (Hevner et al., 2004). Methods comprise best practices (Hevner et al., 2004). As indicated in Fig. 6.1 (#4), the artifact resulting from this research endeavor is the *API Management Pattern Catalog for Public, Partner, and Group Web APIs with a Focus on Collaboration (AMPC)* (Bondel and Matthes, 2023). The AMPC is a pattern catalog comprising of 22 related patterns, 37 pattern candidates, nine stakeholders, and five observations related to collaboration between stakeholders in public, partner, and group Web API initiatives.

Design Evaluation. The design evaluation aims to demonstrate an artifact's utility in solving a specific problem (Hevner et al., 2004). Researchers must rigorously execute the evaluation method (Hevner et al., 2004). The primary goal of the AMPC is to support practitioners in managing the provision of API initiatives used across organizational boundaries. To achieve this

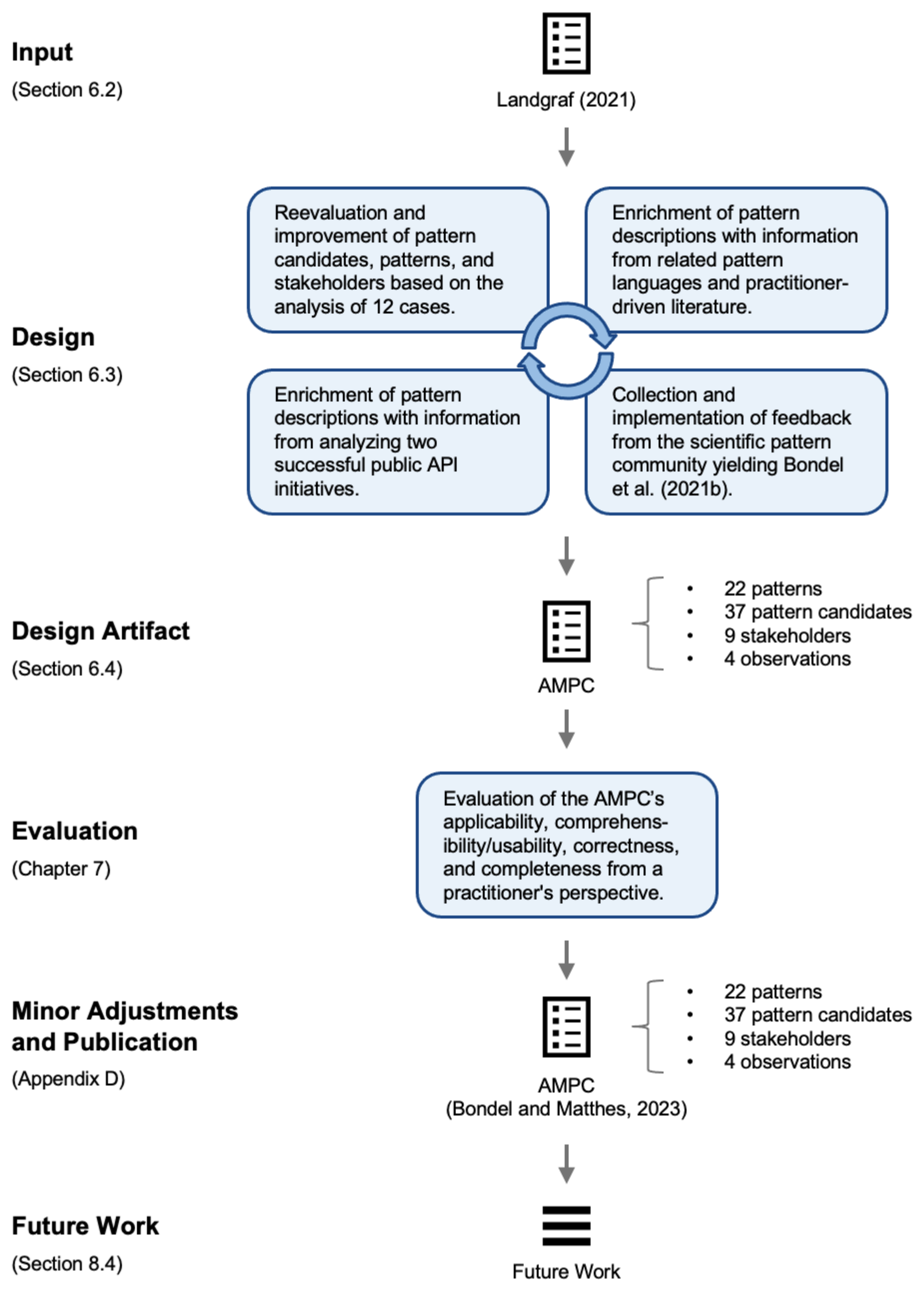

Input
(Section 6.2)
Landgraf (2021)
Reevaluation and improvement of pattern candidates, patterns, and stakeholders based on the analysis of 12 cases.
Enrichment of pattern descriptions with information from related pattern languages and practitioner-driven literature.
Enrichment of pattern descriptions with information from analyzing two successful public API initiatives.
Collection and implementation of feedback from the scientific pattern community yielding Bondel et al. (2021b).
Design
(Section 6.3)
Design Artifact
(Section 6.4)
AMPC
22 patterns
37 pattern candidates
9 stakeholders
4 observations
Evaluation
(Chapter 7)
Evaluation of the AMPC's applicability, comprehensibility/usability, correctness, and completeness from a practitioner's perspective.
Minor Adjustments and Publication
(Appendix D)
AMPC
(Bondel and Matthes, 2023)
22 patterns
37 pattern candidates
9 stakeholders
4 observations
Future Work
(Section 8.4)
Future Work

Figure 6.2.: Iterative improvement and publication of the AMPC (Bondel and Matthes, 2023).

goal, the AMPC must capture relevant but obscure knowledge (Coplien, 1996) in an accessible way. In addition, it aims to create knowledge for research to enable theorizing.

Thus, we evaluated the AMPC twice, as illustrated in Fig. 6.1 (#5). We first participated in shepherding and a writers workshop to ensure that the pattern structure captures all relevant information a reader requires to truly understand and apply a pattern. Moreover, we ensured that the AMPC implements pattern writing best practices to maximize comprehensibility and usability. Afterward, we evaluated the applicability, comprehensibility and usability, correctness, and completeness of the AMPC from a practitioner's point of view using an online survey.

Research Contributions. Design science research must contribute to the knowledge base and solve a previously unsolved business problem (Hevner et al., 2004; Hevner, 2007). Contributions can be the design artifact itself, knowledge about the artifact construction, or design evaluation methodologies (Hevner et al., 2004).

The contribution of this research endeavor is the artifact itself, i.e., the AMPC (Bondel and Matthes, 2023). The AMPC contributes to practice in several ways. First, the AMPC documents proven best practices for managing Web APIs used across organizational boundaries focusing on stakeholder collaboration. This operational knowledge supports practitioners in designing new API initiatives (see Section 7.4). In addition, practitioners can use the AMPC to benchmark current API initiatives with state-of-the-art practices. Also, the AMPC presents a consistent taxonomy that stakeholders can use to communicate. Finally, the AMPC documents knowledge that can help educate developers on Web API management. Hence, it supports organizations in reaping the potential of public, partner, and group API initiatives by addressing the challenge of API provider teams. Therefore, the generated knowledge feeds into the relevance cycle (Hevner, 2007) as visualized in Fig. 6.1 (#6a).

In addition, as Hevner et al. (2004) states, *"[...] utility informs theory."* (Hevner et al., 2004, p. 80). Hence, the AMPC contributes to research by providing a basis for theory building in the domain of Web API management for pubic, partner, and group API initiatives. Also, future changes to the pattern catalog will allow researchers to create knowledge on the evolution of the discipline of Web API management. As a result, new knowledge is added to the knowledge base (Hevner, 2007) as indicated in Fig. 6.1 (#6b).

Research Communication. Finally, an essential aspect of design science is its suitable communication to research and management audiences (Hevner et al., 2004). Hence, we published the final AMPC online, free of charge in Bondel and Matthes (2023). The AMPC motivates and describes its goal, creation approach, and relevant foundational knowledge. Moreover, the AMPC describes its structure to support practitioners in its use. The core contribution is a set of patterns and their relations, which the AMPC documents in a comprehensible and structured manner.

6.2. Previous Work

The AMPC builds on previous work presented in the student thesis Landgraf (2021) advised by the author of this dissertation. The student thesis reports on the data collection, initial analysis, and resulting API management pattern collection. In creating the AMPC, we reused the case base assembled in the course of Landgraf (2021) as the data basis but excluded some cases (see below). Moreover, we analyzed the remaining cases applying a similar data analysis approach and utilized some previous results as input to the analysis. Nevertheless, these activities resulted in a change in the pattern catalog's structure, each pattern's structure, and the instantiation of the patterns. Appendix B makes differences in the essence of the patterns presented in Landgraf (2021) and the AMPC transparent.

Hence, we summarize the case base creation, analysis, and relevant results of Landgraf (2021) in the following.

6.2.1. Case Base

The author of Landgraf (2021) and the author of this dissertation collaborated closely during the data collection. The first step of creating the case base was to collect data on API initiatives during 16 semi-structured interviews with 15 API management team members between August 2020 and January 2021. Of these interviews, 13 were initial, and three were follow-up, allowing more information on specific pattern candidates to be elicited. Since the goal was to capture best practices, the interviews focused on the interviewees' daily API management tasks, including past and current tasks, issues, and solutions to these issues. An overview of the interviews is provided in Tab. 6.1. All interviews were transcribed.

Next, the API initiatives described in the interviews were categorized into distinct cases to allow the identification of patterns using the *rule of three* (Coplien, 1996). Since some interviewees reported information about several API management initiatives in one interview, the API initiatives were categorized into different cases if they used different API portals, as described in Bondel et al. (2021b):

> "[...] The rationale for choosing this granularity level is that we observed that the APIs managed in one API portal all share the same general context regarding collaboration with the API consumers. For example, the provider of a public API portal with several endpoints and more than 10,000 external consumers could provide a contact form for API consumer inquiries. However, if the same organization also provides an API portal for few endpoints and less than five contractual partners, direct communication via email or phone could be more suitable. Thus, characteristics influencing the suitability of patterns like the type of users, the number of users, or the API initiative's maturity level can differ for different API portals."
>
> – (Bondel et al., 2021b)

Overall, the case base comprises 15 cases. We present an overview of the case base in Tab. 6.2. However, the first interview did not entail enough information to allow for the identification of API management patterns (Landgraf, 2021). Consequently, the 14 cases C2-C15 were analyzed as the basis for the pattern collection presented in Landgraf (2021).

6.2.2. Data Analysis

In parallel, the data was analyzed using GTM, an approach for qualitative content analysis in Information Systems (IS) research (Wiesche et al., 2017). The seed categories for open coding were *stakeholders* and *patterns*. In addition to creating subcategories for the seed categories, new categories were added if deemed suitable. Also, new codes were continuously compared to existing codes.

Next, the *rule of three* (Coplien, 1996) was applied to identify patterns (also applied by Uludağ

Table 6.1.: Overview of the interviews informing the case base adopted from Bondel et al. (2021b).

#	Industry	Role	# Employees	Duration (hh:mm:ss)	Participants
1	Financial services	Backend Developer	11 - 50	00:22:52	IV1
2	Industrial manufacturing	Internal Consulting	>100.000	00:44:09	IV2
3	Automotive	Product Owner, Product Owner	>100.000	00:48:49	IV3, IV4
4	Financial services	Software Architect	1001 - 5000	00:42:25	IV5
5	Mobility	Portfolio Manager	1001 - 5000	00:51:12	IV6
6	Insurance	Software Architect	51 - 250	00:59:28	IV7
7	Industrial manufacturing	Product Owner	>100.000	00:46:34	IV8
8	Industrial manufacturing	Software Architect	>100.000	00:47:03	IV9
9	Financial services	Software Developer	10.001 - 50.000	00:35:25	IV10
10	Financial Services	Internal Consulting	5001 - 10.000	00:50:49	IV11
11	Insurance	Integration Architect	51 - 250	00:56:29	IV12
12	Automotive	Product Owner, Product Owner	>100.000	00:51:48	IV3, IV4
13	Financial Services	Technical Lead, Product Owner	5001 - 10.000	00:55:25	IV13, IV14
14	Financial Services	Software Architect	1001 - 5000	00:50:49	IV5
15	Mobility	Portfolio Manager	1001 - 5000	00:31:58	IV6
16	Mobility	Internal Consulting	1001 - 5000	00:45:44	IV15

et al. (2019); Buckl et al. (2008); Khosroshahi et al. (2015)). The rule of three states that *"[...] a good pattern should have three examples that show three insightfully different implementations"* (Coplien, 1996, p. 35).

Overall, this initial API management pattern collection presented in (Landgraf, 2021) comprises 23 patterns, 35 pattern candidates, 32 concerns, 20 influence factors, and ten stakeholders.

Table 6.2.: Overview of the case base derived from expert interviews adapted from Bondel et al. (2021b).

Case ID	# Interview	Architectural Openness	Maturity	Number of API Consumers
C1 (excluded)	1	Private	Development	<20
C2	2	Partner	Pilot	<20
C3	3, 12	Public & Partner	Production	>20
C4	4, 14	Public	Production	>10000
C5	4, 14	Partner	Production	>20
C6	5, 15, 16	Group	Production	na
C7	6	Group	Development	<20
C8	7	Private	Development	>20
C9	8	Public & Partner	Production	na
C10	9	Partner	Production	na
C11	9	Public & Partner	Production	>10000
C12	10	Partner	Pilot	<20
C13	11	Public & Partner	Production	<20
C14	13	Public & Partner	Production	>10000
C15	13	Private	Development	<20

6.3. AMPC Design

This section describes the design of the AMPC building on Landgraf (2021). First, we improved the data analysis, yielding the stakeholders and patterns in the AMPC. Moreover, we enriched the pattern descriptions using information from chosen, successful public Web API initiatives and practice-driven API management literature. In parallel, we collected and realized requirements on the pattern form from the scientific pattern community in the course of publishing intermediary results in Bondel et al. (2021b).

6.3.1. Improvements to the Data Analysis

Using the case base created as part of Landgraf (2021) as a basis, we analyzed the data to identify pattern candidates. Since the AMPC aims to describe patterns for public, partner, and group API initiatives, we excluded the cases describing private API initiatives from the data analysis. More precisely, in addition to the already excluded case C1, we excluded C8 and C15. Hence, we analyzed 12 cases describing public, partner, and group API initiatives as presented in Tab. 6.3.

Given the new data basis, the author of this dissertation reviewed all collected data, i.e., the interview transcripts, a second time. First, we used GTM (Wiesche et al., 2017) to identify pattern candidates and stakeholders in the data basis. We used the pattern candidates, patterns, and stakeholder names previously presented in Landgraf (2021) as seed categories and applied open coding to assign codes to these seed categories. Also, we added new categories if we

Table 6.3.: Adopted case base used as a basis for the creation of the AMPC adapted from Bondel et al. (2021b).

Case ID	**# Interview**	**Architectural Openness**	**Maturity**	**Number of API Consumers**
C1 (excluded)	1	Private	Development	<20
C2	2	Partner	Pilot	<20
C3	3, 12	Public & Partner	Production	>20
C4	4, 14	Public	Production	>10000
C5	4, 14	Partner	Production	>20
C6	5, 15, 16	Group	Production	na
C7	6	Group	Development	<20
C8 (excluded)	7	Private	Development	>20
C9	8	Public & Partner	Production	na
C10	9	Partner	Production	na
C11	9	Public & Partner	Production	>10000
C12	10	Partner	Pilot	<20
C13	11	Public & Partner	Production	<20
C14	13	Public & Partner	Production	>10000
C15 (excluded)	13	Private	Development	<20
C16 (Stripe)		Public	Production	<>10,000
C17 (Twilio)		Public	Production	<>10,000

identified practices or stakeholders not belonging to any of the seed categories. During the whole process, we continuously compared new codes to existing codes. As a result, we identified 56 pattern candidates to which we applied the *rule of three* (Coplien, 1996). Of these 56 pattern candidates, 22 were validated as actual patterns for API management with three or more known usages.

Furthermore, in addition to the data collected through interviews, we included publicly available information on these API initiatives if possible, i.e., on public API initiatives. Such public information included information published online via the API provider's developer portal or on other provider owned websites or, in one case, via a podcast.

6.3.2. Enrichment of Pattern Descriptions

We enriched the pattern descriptions with additional information derived from successful public API initiatives and practice-driven API management literature.

The case base mostly held cases that describe the API initiatives of established, traditional organizations with primary business activities independent of their API initiative. Hence, we added Stripe[1] and Twilio[2] to the case base as visible in Tab. 6.3 and systematically reviewed and

[1]https://stripe.com/

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

added public information on these API initiatives. However, we did not identify new patterns but endeavored to validate and add further information to the patterns previously identified in cases C2-C7 and C9-C14. Hence, the inclusion of the API initiatives of Stripe and Twilio added an additional perspective to the patterns documented in the AMPC.

In addition, we systematically included information from API management literature into the pattern descriptions. First, we reviewed the pattern languages and catalogs presented in Chapter 3 and related them to the patterns in the AMPC. In addition, we systematically analyzed practice-driven literature on best practices for API management to enrich the pattern descriptions, e.g., the ‘Forces’ or ‘Implementation Hints’ sections. The reviewed literature comprises Jacobson et al. (2012), Spichale (2017), De (2017) and Medjaoui et al. (2018). The inclusion of these literature sources created richer pattern descriptions.

6.3.3. Evolution of the Pattern Form

Finally, we aimed to refine the AMPC to meet the requirements and best practices of the scientific software engineering pattern community. We collected and implemented feedback from the scientific pattern community by participating in the submission and publication process of the European Conference on Pattern Languages of Programs 2021 (EuroPLoP’21)[3]. The EuroPLoP belongs to a series of conferences organized by the Hillside Group, i.e., a non-profit educational group founded in 1993 by the pioneers of software engineering patterns to improve and promote pattern and pattern language design and usage (Hillside Group).

In comparison to other scientific conferences, the EuroPLoP *"[...] focuses on improving papers instead of only presenting them"* (Hillside Europe e. V., a). Since the goal of participating in the conference was to validate and improve the research approach, the resulting structure of patterns, and the overall pattern catalog design of the AMPC, we summarized these aspects in a research paper. The paper also presented two example patterns, i.e., the patterns `Frontend venture` and `Role-based marketing`. We submitted the paper to the EuroPLoP 2021 in February 2021. The research paper was accepted, allowing us to participate in the submission and publication process.

The EuroPLoP submission and publication process comprises a shepherding phase and a pattern writers workshop (Coplien, 1996), which take about seven months and require continuous improvement of the submitted contents (Hillside Europe e. V., b). During the shepherding phase, an experienced pattern author (shepherd) partners with the pattern author of a submitted paper and provides continuous feedback (Hillside Europe e. V., b). In our case, the shepherding phase lasted from mid-March to mid-June 2021 and comprised several rounds of iterative improvements to the pattern and overall pattern catalog structure.

Afterward, the already improved paper version was reviewed at a writers workshop at the EuroPLoP conference in July 2021. The writers workshop follows an approach derived from poetry review workshops (Coplien, 1996). The approach dictates that a peer group of five pattern authors with varying experience review each other’s papers and discuss their findings and feedback during a one-hour session. During that session, the author can voice their goals for the workshop

[3] `https://www.europlop.net/content/conference`

in the first couple of minutes. Hence, previous to the workshop, we sent a questionnaire detailing the goal of the workshop participation to the participants (see Appendix C). However, the author cannot participate or steer the discussion afterward. Instead, the author collects feedback from the peer group without interfering. We recorded and transcribed the writers workshop to ensure we captured all feedback. In addition, some writers workshop participants provided further feedback via email after the workshop.

After incorporating all feedback from shepherding and the writers workshop, we published the results in Bondel et al. (2021b). Hence, Bondel et al. (2021b) presents intermediary results.

In the following, we summarize the feedback collected during the EuroPLoP submission and publication process, including the shepherding phase, the writers workshop, and further feedback received after the workshop. First, we present positive feedback. Afterward, we present requirements for the AMPC design derived from the feedback. Also, we describe the realization of these requirements in the AMPC. If applicable, we used the proven pattern writing patterns presented by Meszaros and Doble (1997) to realize the requirements.

We received the following positive feedback from the shepherd and writers workshop participants:

- **Contribution.** Creating an API management pattern language presents a valuable contribution to practice. Moreover, addressing the collaboration in API management seems to fulfill a relevant research gap.
- **Methodology/Approach.** The authors developed the patterns based on input from industry experts, thus capturing real-world experiences. Hence, the pattern catalog links the patterns derived from real-world cases to scientific literature very well.
- **Pattern structure.** The pattern descriptions comprise all relevant sections, making it easy to identify the most relevant information. Also, the 'Problem' and 'Solution' sections of each pattern nicely explain the essence of a pattern when read in isolation. Finally, the 'Example' and 'Implementation Hints' are very readable and tangible.

The scientific pattern community named several requirements to the pattern and pattern catalog structure in their feedback. In the following, we list these requirements and describe their realization in the AMPC:

- **Requirements for the Pattern Descriptions:**
  - **Pattern section structure.** The sections of a pattern need to follow a logical order, e.g., the order presented in `B.1 Mandatory Elements Present` (Meszaros and Doble, 1997). Hence, the pattern section sequence in the AMPC follows a logical structure by sequentially describing the 'Context', 'Concern', 'Forces', 'Solution', 'Stakeholders', 'Implementation Hints', 'Consequences', 'Related Patterns within this Pattern Catalog', 'Other Related Patterns', and 'Known Uses'.
  - **Pattern naming.** Pattern names should be consistent, short, and easy to use in everyday language. In the AMPC, all pattern names are `C.3.1 Noun Phrase Names` (Meszaros and Doble, 1997), i.e., short noun strings that describe the solution.

– **Sketch.** A pattern description should provide a sketch of the solution. Thus, we included an abstract illustration of the solution in each pattern description in the AMPC.

– **Stakeholders.** Stakeholder lists need to be self-explanatory. Hence, the AMPC describes each stakeholder's role in realizing a respective pattern.

– **Concern.** Each pattern needs a clear problem description free of forces that influence it. Therefore, the AMPC formulates the problem statement as one question in the section 'Concern'. In addition, we captured potential forces and made them explicit in the 'Forces' section.

– **Forces.** Forces should explain why it is hard and non-trivial to find and apply a solution (Coplien, 1996). The AMPC clearly describes these forces.

– **Consequences.** Consequences describe the the result of applying a pattern (Gamma et al., 1995). The AMPC describes how the instantiation of a pattern resolves each force.

– **Related Patterns.** Patterns should refer to other patterns within and outside the respective pattern language or catalog (Meszaros and Doble, 1997). There should be a clear distinction between related patterns within the same pattern catalog and external sources, e.g., existing pattern languages. As a result, in the AMPC, we separated the section 'Related Patterns' into a section 'Related Patterns within this Pattern Catalog' and a section 'Other Related Patterns'.

– **Relationships between Patterns.** Patterns can have several types of relationships besides "lead-to" relationships (Meszaros and Doble, 1997). We described the relationships between patterns in the 'Related Patterns within this Pattern Catalog' section of the AMPC. Patterns can be complementary, conflicting, or one pattern specializes another pattern.

– **Description of Known Uses.** Known uses should help a reader to understand how to apply a pattern in different settings. Therefore, the AMPC describes each case and how it applies the respective pattern in the 'Known Uses' section. In addition, we summarized commonalities and differences between the cases and presented assumptions as to why these characteristics foster the applicability of the respective pattern.

– **Cross-references.** Each pattern should be readable as a document under closure, and cross-references make it difficult for readers to understand a pattern (Meszaros and Doble, 1997). Therefore, in the AMPC, we aimed to realize the pattern `B.4 Single-Pass Readable` (Meszaros and Doble, 1997). By referencing related patterns with their name according to `C.2 Readable References to Patterns` (Meszaros and Doble, 1997). Furthermore, we described each case in the 'Known Uses'. Also, we incorporated short explanations of terms potentially unknown to the target audience within the pattern descriptions.

– **"How-to" questions.** One workshop participant proposed avoiding the use of "how-to" questions in the ‘Concerns’ section. However, in some cases, "how-to" questions best capture the essence of a problem. Therefore, we kept "how-to" questions to express the concerns where applicable.

 – **Real-wold examples.** Each known use should provide as much information as possible. In the best case, each known use presents a real-world example. While we aimed to realize the goal of providing as much information on each known use as possible, we were limited by the confidentiality agreements in place with the interview partners. We could describe real-world examples only in detail for publicly observable applications of patterns.
 – **Example resolved.** It is possible to include the sections ‘Example’ and ‘Example Resolved’ as presented in Buschmann et al. (1996). However, we decided to report examples in the ‘Known Uses’ section of the AMPC.

- **Requirements for the Pattern Catalog Structure**:
  – **Pattern relation visualization.** A pattern catalog should visualize the relation between patterns. Hence, the AMPC visualizes the relationships between patterns as described in their respective ‘Related Patterns within this Pattern Catalog’ sections. In addition, the visualization also shows entry points into the pattern language for API providers and consumers.
  – **Patlets:** A pattern catalog should provide an overview of all patterns using a `E.1.1 Problem/Solution Summary` (Meszaros and Doble, 1997) or a patlets table. The AMPC presents a patlets table comprising short descriptions of each pattern.

Hence, the AMPC realizes all requirements except for "How-to" questions, Real-wold examples, and Example resolved.

6.4. The API Management Pattern Catalog (AMPC)

In the previous sections, we described the AMPC design. This section summarizes the resulting AMPC’s contents as published in Bondel and Matthes (2023).

First, we present the structure of the AMPC based on its meta-model. Afterward, we provide an overview of the identified stakeholders. Moreover, we show an overview of the patterns, their relations to each other, and their relations to patterns from other pattern languages and catalogs. Finally, we discuss the pattern candidates. The AMPC comprises nine stakeholders, 22 patterns, and 37 pattern candidates.

6.4.1. Structure of the AMPC

The AMPC documents the entities and their relations as visualized in the meta-model in Fig. 6.3. These entities comprise the *API management software artifacts*, *stakeholders*, *patterns*, and *pattern candidates.*

The API management software artifacts potentially exchange data. The stakeholders are responsible for or use API management software artifacts. The stakeholders apply or support the application of API management patterns. In all cases, different stakeholders must collaborate to implement a pattern, thus creating relations between the stakeholders. API management patterns were pattern candidates before they were validated to be patterns according to the *rule of three* (Coplien, 1996).

We do not discuss the API management software artifacts in this section, as they have been previously presented in Section 2.1.1.

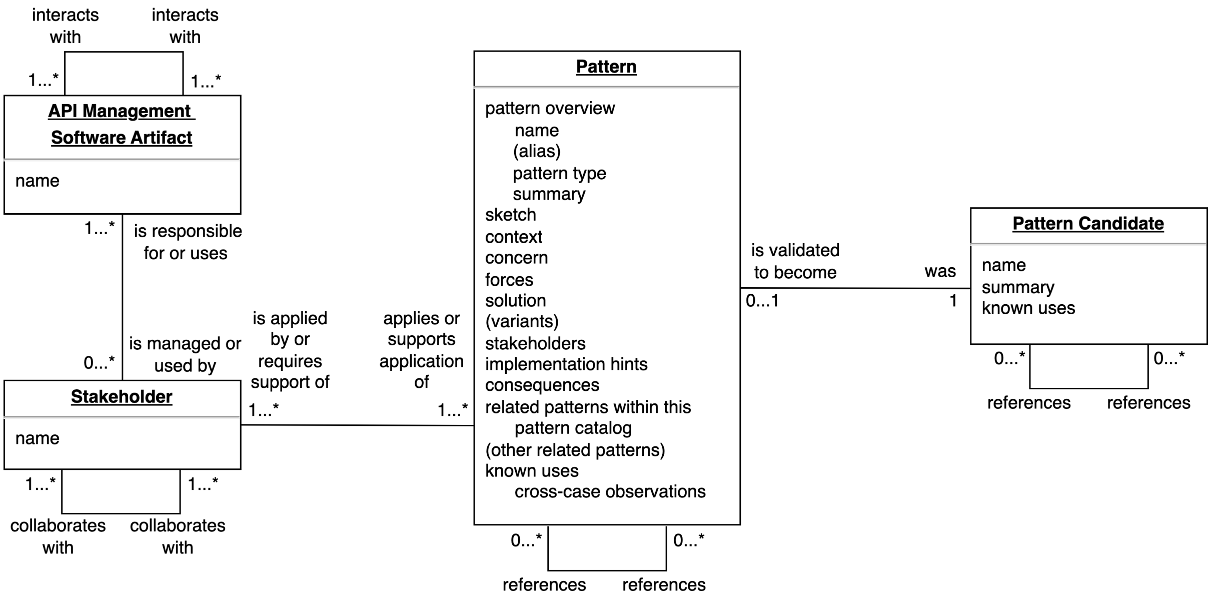


Figure 6.3.: Meta-model of the AMPC.

6.4.2. Stakeholders

We identified nine stakeholders involved in interactions using Web APIs as illustrated in the upper part of Fig. 6.4 from the interview data and literature (Bondel et al., 2021b; Bondel and Matthes, 2023). We identified the stakeholders involved in a basic Web API interaction as presented in Section 2.1.2, i.e., the *backend provider*, *API provider*, *API consumer*, and *end user*. In addition, we identified the *upper management*, *legal*, *sales & marketing*, *customer support*, and *integration partners* to be further stakeholders involved in API management. A more detailed description of the stakeholders is presented in the AMPC (Bondel and Matthes, 2023).

6.4.3. Patterns

The patterns are the core contribution of the AMPC (Bondel and Matthes, 2023). Overall, the AMPC presents 22 patterns. Fig. 6.5 provides an overview all patterns and their relations. Also, we present short summaries of all patterns in Appendix E. Moreover, as examples, we present the

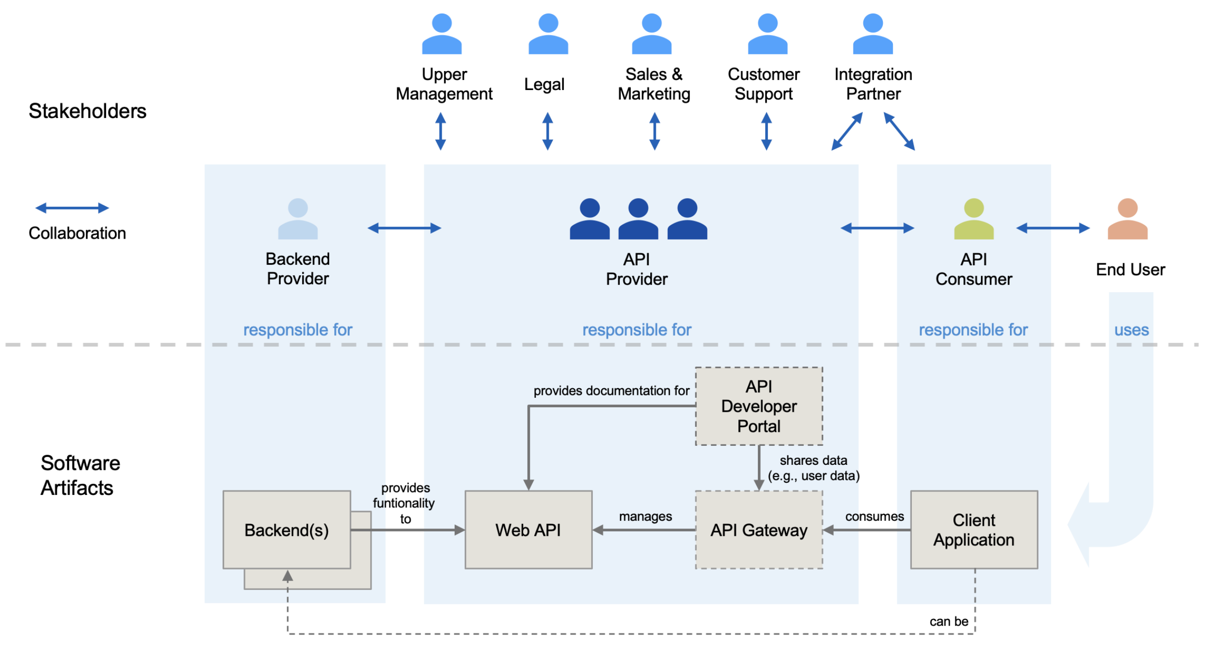


Figure 6.4.: Overview and relation of software artifacts and stakeholders involved in API management adapted from Bondel et al. (2021b); Bondel and Matthes (2023).

complete descriptions of the two patterns `Collaborative Pilot Project` and `Frontend Venture` in Appendix F.

In addition to relating the patterns, Fig. 6.5 presents entry points for API providers and consumers who want to explore the AMPC. Also, some patterns are categorized into product management domains.

Each pattern description comprises the same set of mandatory or optional elements. In the following, we describe these elements (adopted from Bondel and Matthes (2023)).

- "Each pattern has a 'Name' and potentially one or more 'Aliases'. A name allows readers to find relevant patterns quickly and can become part of an API management team's vocabulary (Coplien, 1996).
- Furthermore, each pattern belongs to one of the *pattern categories Interface Type Pattern*, *API Provider Internal Patterns*, and *API Consumer-facing Patterns*. Interface Type Patterns capture different approaches to making functionality and data provided via API available to API consumers. API Provider Patterns describe patterns that require the API provider team to collaborate mainly with API provider organization internal stakeholders. In contrast, API Consumer-facing Patterns are concerned with the interaction between the API provider and consumer.
- A short 'Summary' of each pattern allows the reader to grasp the essence of a pattern quickly.

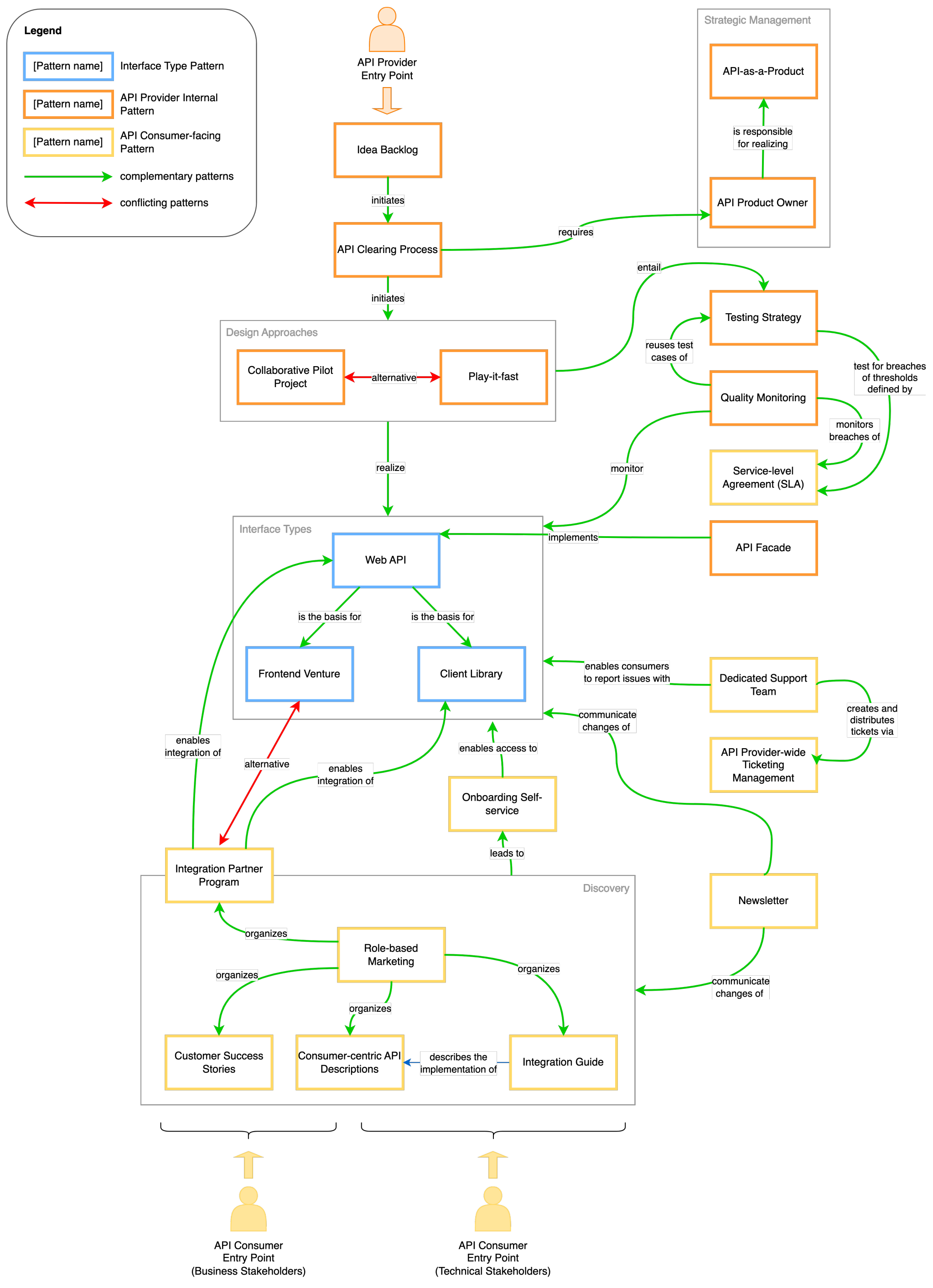


Figure 6.5.: Overview of the patterns in the AMPC and their relations to each other adopted from Bondel and Matthes (2023).

- A ‘Sketch’ visualizes the patterns basic concept (Coplien, 1996).
- The ‘Context’ describes the situation in which the reader can apply a pattern (Meszaros and Doble, 1997). The situation imposes constraints that the solution needs to address (Meszaros and Doble, 1997).
- A ‘Concern’ captures the (design) problem that the pattern addresses (Meszaros and Doble, 1997; Coplien, 1996). Concerns usually represent the interests and goals of the stakeholders applying a pattern (Uludağ et al., 2019; Buckl et al., 2008; Khosroshahi et al., 2015), i.e., the API provider team.
- Patterns address concerns that are difficult to solve due to contradictory goals and considerations of stakeholders (Meszaros and Doble, 1997; Coplien, 1996). The ‘Forces’ describe these trade-offs (Meszaros and Doble, 1997; Coplien, 1996). The context usually indicates which forces the pattern should optimize (Meszaros and Doble, 1997). Moreover, understanding the forces allows the reader to better understand the concern and the solution (Coplien, 1996).
- The ‘Solution’ captures the core approach to solving the concern in a given context (Meszaros and Doble, 1997). The solution provides enough detail to enable the API provider team to apply the pattern. Nevertheless, simultaneously, the solution is generic enough to apply to many contexts (Coplien, 1996). Furthermore, the solution dictates how the forces are resolved (Meszaros and Doble, 1997). In some cases, we observed ‘Variants’ of a solution.
- The ‘Stakeholders’ list all roles involved, affected, or influenced by API management. Stakeholders can be internal or external to the API provider team’s organization. This pattern catalog focuses on the API provider team, including the Web API provider, the API gateway provider, the API developer portal provider, and the API governance role.
- The ‘Implementation Hints’ provide additional information supporting the successful implementation of the pattern.
- The ‘Consequences’ pick up on the forces and explain which considerations the solution optimizes at the expense of others.
- The ‘Related Patterns within this Pattern Catalog’ section relates a pattern to other patterns presented in this pattern catalog. [Patterns can complement each other, a pattern can specialize another pattern, or patterns can be alternatives. For example, the patterns `API-as-a-Product` and `API Product Owner` complement each other. Similarly, a `Consumer Success Story` can be part of realizing `Role-based Marketing`. In comparison, the patterns `Collaborative Pilot Project` and `Play-it-fast Approach` are alternatives to solving the same problem, that prioritize either time-to-market or fit between API and stability. The relations between the patterns are presented in Fig. 6.5.]
- Similarly, ‘Other Related Patterns’ point to patterns already published by other

authors. [...] This section is only instantiated if the pattern at hand relates to any of these other patterns.

- 'Known uses' describe the cases in which we observed the pattern and, if possible, provide some details on its implementation. Since we apply the *rule of three* (Coplien, 1996), each pattern has at least three known uses. [Moreover, the subsection 'Cross-case observations' summarizes similarities and differences between these cases.]"

– (Bondel and Matthes, 2023)

6.4.4. Relations to Other Pattern Collections

As mentioned above, the AMPC's patterns relate to patterns of other pattern collections as described in the section 'Other Related Patterns' of each pattern description. Fig. 6.6 presents an overview of these relations. The pattern relations can be complementary, conflicting, or one pattern specializes another pattern. In the following, we briefly describe these relations.

First, the patterns `Pricing Plan` and `Rate Limit` presented by (Zimmermann et al., 2022), Zimmermann et al., and Stocker et al. (2018) can be part of the AMPC's pattern `API-as-a-Product`. In addition, both pattern collections present a pattern `Service Level Agreement`. The essence of the two pattern descriptions is very similar, but the scope and level of detail differ. Therefore, we view the two pattern descriptions as confirmation and extension of each other. Next, an `API Key` (Zimmermann et al., 2022) can enable an `Onboarding Self-service`. Also, the patterns `API Description` (Zimmermann et al., 2022; Zimmermann et al.; Lübke et al., 2019) can be part of the AMPC's patterns `Consumer-centric API Description` and `Integration Guide`.

Geewax (2021) puts forward a collection of *API design patterns* concerned with, e.g., resource naming, the use of standard or custom methods, or cross-references. Therefore, *API design patterns* can be used to guide the technical implementation of the `Web API` pattern presented in the AMPC.

The *Patterns for RESTful Conversations* by Pautasso et al. (2016) can be used to implement a `Web API` as described in the AMPC.

Similarly, Bellido et al. (2013) presents *Control-Flow Patterns for Decentralized RESTful Service Composition* documenting four types of control-flows. These patterns can be used in the implementation of `Web APIs` according to the AMPC.

Next, Daigneau (2011) presents a set of *Service Design Patterns* that can support the implementation of the `Web API` pattern as presented in the AMPC. Daigneau (2011) also details the patterns `RPC API` and `Message API` which are also described in other pattern collections.

Erl (2008) presents `Service Messaging` which enables the realization of the AMPC's pattern `Web API`. Also, Erl (2008) presents the pattern `Service Façade`, which aims to reduce the coupling of clients to services. Decoupling the service contract from the underlying implementation is one aspect of the `API Façade` pattern in the AMPC, which focuses on orchestrating functionality provided by several backends.

The pattern `Service Watchdog` presented by Rotem-Gal-Oz (2012) can support the pattern `API Quality Monitoring` in the AMPC.

Richardson presentes the microservices patterns `Service Component Test` and `Consumer-driven Contract Test` that can be part of the AMPC's pattern `Test Strategy`. Similarly, the microservices patterns `Health Check API` (Richardson) and `Application Metrics` Richardson support the AMPC's pattern `Quality Monitoring`. Moreover, Richardson describes the pattern `Messaging` and `Remote Procedure Invocation (RPI)`, which document different approaches to implementing a `Web API` as described in the AMPC. Finally, the microservices pattern `Access Token` (Richardson) can enable the AMPC's pattern `Onboarding self-service`.

The enterprise integration pattern language and AMPC have a different focus, but some patterns are complementary. First, the `Test Massage` pattern (Hohpe and Woolf, 2003) can support the AMPC's `Quality Monitoring` pattern. In addition, we can relate the AMPC's pattern `Web API` with the root pattern `Messaging` (Hohpe and Woolf, 2003).

Völter et al. (2004) presents the `Interface Description` pattern which should be part of the AMPC's patterns `Consumer-centric API Descriptions` and `Integration Guide`.

Some of Fowler (2003)'s Enterprise Application Patterns complement some patterns in the AMPC. For once, the AMPC's `API Facade` is a specialization of the `Remote Facade` pattern presented by Fowler (2003). Furthermore, a `Service Stub` (Fowler, 2003) can be part of the AMPC's `Testing Strategy`.

Gamma et al. (1995) presents design patterns concerned with the creation, composition, and interaction between classes and objects in object-oriented systems. As a result, some design patterns for object-oriented systems enable the implementation of the patterns in the AMPC. More specifically, the AMPC's `Client Library` is an implementation of an `Adapter` (Gamma et al., 1995). Furthermore, the AMPC's `API Facade` is a domain-specific implementation of the `Facade` pattern presented in Gamma et al. (1995).

Buschmann et al. (1996) and Buschmann et al. (2007b) present the `Facade` pattern, previously published by Gamma et al. (1995). The `Facade` pattern relates to the AMPC's pattern `API Facade` in that the latter is a specialization of the former.

Finally, Dyson and Longshaw (2004) present the patterns `Continual Status Reporting` and `Operational Monitoring and Alerting` that can support the AMPC's pattern `Quality Monitoring`.

6.4.5. Pattern Candidates

Finally, we could not validate 37 pattern candidates as patterns according to the rule of three (Coplien, 1996). Instead, the AMPC presents each pattern candidate with a name, a solution summary, and known uses. Pattern candidates can complement, specialize, or provide alternatives to patterns.

6.5. Discussion

During the design and refinement of the AMPC, we made five general observations related to API management as previously reported in Bondel et al. (2021b). We list these observations in the following:

"**[...] Most initial collaboration between the API provider and the API consumer happens through software artifacts controlled by the API management team.**

API consumers use the developer portal's features to discover, initially inform themselves, and contact the API provider team. Additionally, self-service options foster easy first interactions and testing of APIs. Thus, the successful collaboration between API provider and API consumers heavily depends on resources controlled by the API management team.

[...] API consumers want personal contact with the API provider before and during integrating an API.

Even though API consumers often discover an API via its developer portal, in most cases, the API consumer negotiates contracts with the API provider before actually integrating the API. These contracts contain agreed-upon quality levels in the form of service-level agreements (SLAs). Thus, the API consumer can hold the API provider accountable to provide functionality that meets specific non-functional requirements, e.g., availability or performance levels. Also, the API provider can include terms of use, e.g., number of allowed calls within a time period. The use of APIs initiated solely via self-service and without a separate contract between API consumers and API provider is rare. Furthermore, after signing the contract, the API provider often supports the API consumer with integration activities. [This observation is in line with Islind et al. (2016), who argue that co-creating boundary resources and intimate knowledge communication are essential success factors for creating small-scale platforms.]

[...]

[...] The collaboration between the API provider team and all other [internal] stakeholders is challenging.

Collaboration between the API provider and the backend provider and [other] internal stakeholders mostly focuses on quality, defect, and incident management across team, business unit, or company boundaries. The interviewees stressed the challenges of collaboration between these stakeholders, especially if APIs are not the main distribution channel of a product. In these cases, the API management team often feels the pressure of fixing issues and defects since they are the first point of consumer contact. The backend also has other tasks and only feels indirect pressure through the API provider team. Thus, the backend provider might prioritize API-related issues differently. Also, only a few approaches to standardize the collaboration between these stakeholders exist, and most collaboration relies on ad-hoc communication channels such as email. Hence, the API provider organization should adapt processes and organizational structures when starting an API initiative.

[...] The API provider has to treat the API as a product with a lifecycle.

An API makes resources like functionality, data, or software products with lifecycles accessible to API consumers. However, the API itself changes due to consumer wishes, technology developments, etc., and the API changes can impact the consumer business. Therefore, the API provider needs to actively manage the API lifecycle in coordination with the backend provider and the API consumer.

[...] Strategic relevance and the structure of the consumers' organization are potential influence factors for the suitability of patterns.

The API initiative cases used as a basis for eliciting the API management patterns have different characteristics that can influence the applicability of an API pattern. For example, a candidate for those factors is the question if an API provider offers a commodity functionality or data for a broad audience or if the usage of the API has a strategic impact on the consumers business and is thus probably relevant to a smaller group of consumers, i.e., within one branch. Furthermore, the organizational structure and the technical capabilities of the target consumers organization are relevant. Future research should aim at making such potential influence factors of API initiatives explicit."

– (Bondel et al., 2021b)

6.6. Summary

We applied a design science research approach to create the AMPC (Bondel and Matthes, 2023), which we described along the seven guidelines for conducting and presenting design science according to Hevner et al. (2004). We built on the data collection and analysis of Landgraf (2021).

Based on these previous works, we restricted the data basis to match the scope of the AMPC and reran the data analysis. Each pattern description contains information derived from the expert interviews, enriched with potentially publicly available information on the API initiative. In addition, we added information extracted from two successful public API initiatives Stripe[4] and Twilio[5]. Also, we systematically added information from related pattern languages

[4]https://stripe.com/
[5]https://www.twilio.com/

and practitioner-driven API management literature. In parallel, we collected and implemented requirements for the pattern descriptions from the scientific pattern community through shepherding and a writers workshop, a standard and long-standing approach for pattern improvement in the pattern community (Coplien, 1996). The participation in shepherding and the writers workshop resulted in the publication of intermediary results in (Bondel et al., 2021b).

As a result, we published the AMPC (Bondel and Matthes, 2023) comprising 22 patterns, 37 pattern candidates, and nine stakeholders. The patterns are categorized into *interface type patterns*, *API provider internal activity patterns*, and *API consumer-facing support activity patterns*. Also, the AMPC relates the patterns to the API management pattern collections presented in Chapter 3. Finally, we reported the minor improvements we made to the AMPC after its evaluation but before its publication.

In addition, we reported on five general observations about API management. Accordingly, the initial discovery and collaboration between the API consumer and provider happen through software artifacts controlled by the API provider team. Nevertheless, API consumers want personal contact with API providers before and during API integration. Also, collaboration between the API provider team and internal stakeholders can be challenging. Furthermore, the API provider has to treat the API as a product with a lifecycle. Finally, the strategic relevance of provided functionality and the structure of a consumer organization seem to influence the applicability of patterns significantly.

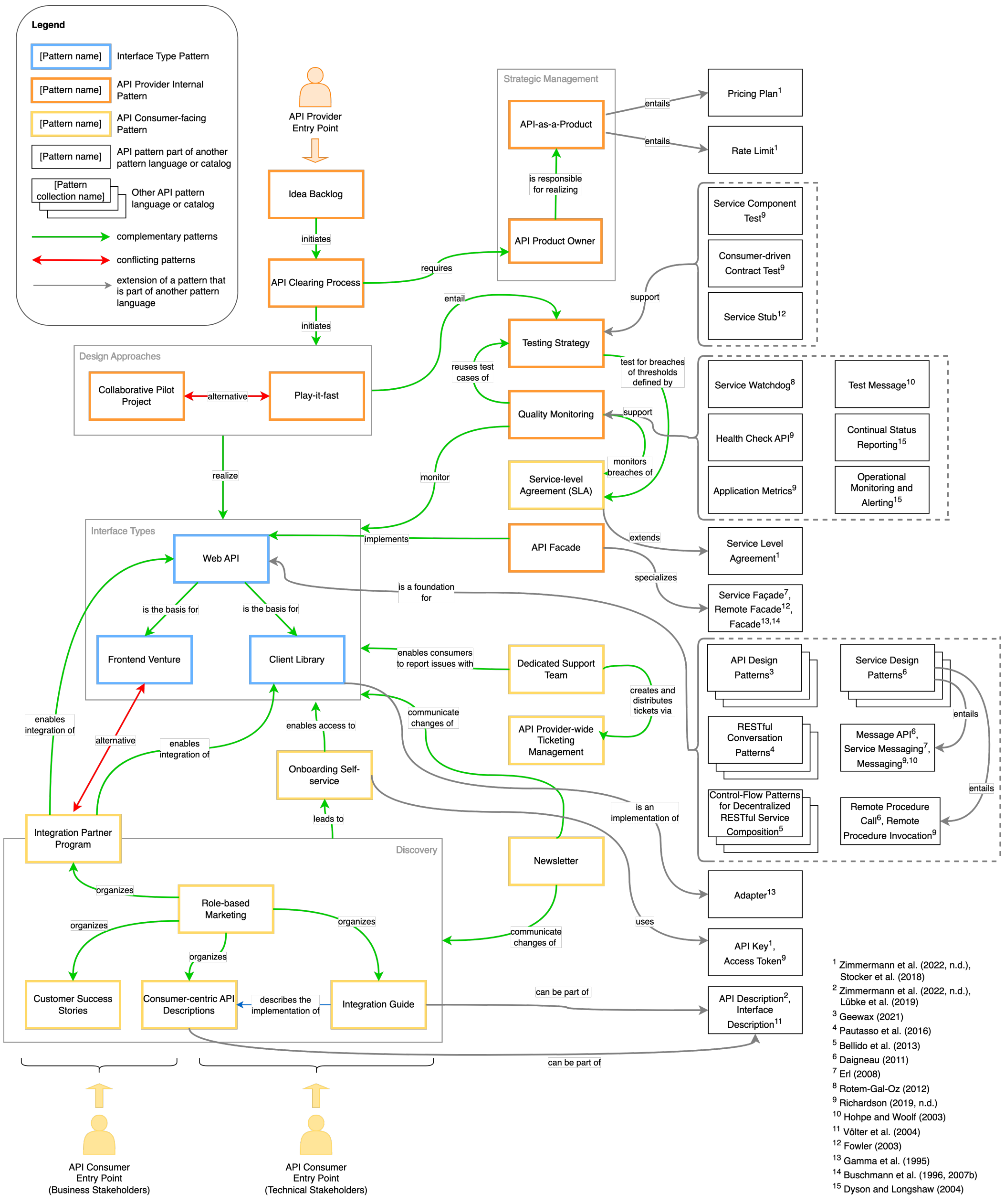

Legend
[Pattern name] Interface Type Pattern
[Pattern name] API Provider Internal Pattern
[Pattern name] API Consumer-facing Pattern
[Pattern name] API pattern part of another pattern language or catalog
[Pattern collection name] Other API pattern language or catalog
complementary patterns
conflicting patterns
extension of a pattern that is part of another pattern language
API Provider Entry Point
Idea Backlog
initiates
API Clearing Process
requires
initiates
Strategic Management
API-as-a-Product
is responsible for realizing
API Product Owner
entails
entails
Pricing Plan[1]
Rate Limit[1]
Service Component Test[9]
Consumer-driven Contract Test[9]
Service Stub[12]
support
entail
Testing Strategy
reuses test cases of
test for breaches of thresholds defined by
Design Approaches
Collaborative Pilot Project
alternative
Play-it-fast
Quality Monitoring
support
monitors breaches of
Service Watchdog[8]
Health Check API[9]
Application Metrics[9]
Test Message[10]
Continual Status Reporting[15]
Operational Monitoring and Alerting[15]
realize
monitor
Service-level Agreement (SLA)
Interface Types
Web API
implements
API Facade
extends
Service Level Agreement[1]
specializes
Service Façade[7], Remote Facade[12], Facade[13,14]
is a foundation for
is the basis for
is the basis for
Frontend Venture
Client Library
enables consumers to report issues with
Dedicated Support Team
creates and distributes tickets via
API Provider-wide Ticketing Management
communicate changes of
API Design Patterns[3]
Service Design Patterns[6]
RESTful Conversation Patterns[4]
Message API[6], Service Messaging[7], Messaging[9,10]
Control-Flow Patterns for Decentralized RESTful Service Composition[5]
Remote Procedure Call[6], Remote Procedure Invocation[9]
entails
entails
enables integration of
alternative
enables integration of
enables access to
Onboarding Self-service
leads to
is an implementation of
Integration Partner Program
Discovery
Newsletter
Adapter[13]
organizes
Role-based Marketing
organizes
organizes
organizes
communicate changes of
uses
API Key[1], Access Token[9]
Customer Success Stories
Consumer-centric API Descriptions
describes the implementation of
Integration Guide
can be part of
API Description[2], Interface Description[11]
can be part of
API Consumer Entry Point (Business Stakeholders)
API Consumer Entry Point (Technical Stakeholders)
1 Zimmermann et al. (2022, n.d.), Stocker et al. (2018)
2 Zimmermann et al. (2022, n.d.), Lübke et al. (2019)
3 Geewax (2021)
4 Pautasso et al. (2016)
5 Bellido et al. (2013)
6 Daigneau (2011)
7 Erl (2008)
8 Rotem-Gal-Oz (2012)
9 Richardson (2019, n.d.)
10 Hohpe and Woolf (2003)
11 Völter et al. (2004)
12 Fowler (2003)
13 Gamma et al. (1995)
14 Buschmann et al. (1996, 2007b)
15 Dyson and Longshaw (2004)

Figure 6.6.: Overview of the patterns in the AMPC, their relations to each other, and their relations to other patterns or pattern languages adopted from Bondel and Matthes (2023).

CHAPTER 7

Evaluation of the API Management Pattern Catalog (AMPC)

This chapter presents the evaluation of the AMPC (Bondel and Matthes, 2023) using a survey. We first detail the survey approach. Afterward, we present an overview of the survey participants' characteristics and the qualitative and quantitative feedback received. In the discussion, we summarize the survey results to evaluate the applicability, comprehensibility/usability, completeness, and correctness of the AMPC. To conclude, we summarize this chapter's contents.

In the following, we refer to IT professionals using the AMPC to gain additional knowledge on API management or guide API initiatives as *readers* of the AMPC.

7.1. Survey Approach

In the following, we detail the survey goal, structure, and participant acquisition as part of the survey approach.

7.1.1. Survey Goal

The survey aimed to evaluate the AMPC concerning its applicability, comprehensibility, usability, completeness, and correctness. We used a survey to collect feedback from various IT roles employed at organizations active in different industry sectors.

7.1.2. Survey Structure

The complete survey is appended in Appendix G. The following summarizes the survey design, including the rationale for some design decisions.

The survey started with a data privacy statement to which the participants had to consent actively. Afterward, the survey presented six categories of questions aiming to evaluate the AMPC. Each category of questions was presented on a separate site.

The first site consisted of eight closed questions that evaluated the applicability, comprehensibility/usability, completeness, and correctness of the AMPC. Each question presented a statement. The survey instructed the participants to use a 5-point Likert scale ranging from "strongly disagree" to "strongly agree" to indicate their level of agreement with the statement. The statement aiming to evaluate the applicability of the AMPC was:

- *"The pattern catalog provides useful patterns that I can apply in my organization."*

The following statements were aimed at evaluating the comprehensibility and usability of the AMPC:

- *"The structure of the pattern is easy to understand."*
- *"I would need the support of an author to be able to use this pattern catalog."*
- *"The visual design of the pattern catalog meets my expectations."*

The second question, asking about the need for author support, was inspired by the system usability scale pioneered by Brooke (1996).

The survey also asked about the perceived completeness of the patterns covered in the AMPC and the completeness of pattern descriptions:

- *"The level of detail of pattern and pattern candidate descriptions is adequate for the problem at hand."*
- *"There is no unnecessary information in the pattern catalog."*
- *"The pattern catalog covers all major topics of API management."*

The last statement, aiming to evaluate the completeness of the patterns covered in the AMPC, additionally allowed for text so that participants could list missing topics.

Also, we included a statement to evaluate the correctness of the AMPC:

- *"The patterns correctly capture the essence of API management problems and solutions."*

The second site asked the participants how likely they would recommend the AMPC to colleagues to calculate the *NPS*. Reichheld (2003) presents the NPS as a measure to analyze customer loyalty, which has been shown to correlate with a company's growth. A customer has to rate the question *"How likely is it that you would recommend (company X) to a friend or colleague?"* on a scale from 0 (very unlikely) to 10 (very likely). The respondents are then grouped into three categories. First, respondents rating the question 9-10 are *promoters*. Promoters are customers who are satisfied to a degree that they vouch for the company with their reputation by referring it to others. Next, respondents rating the question 7-8 are *passively satisfied*. Finally, the *detractors* are the respondents that answer the question with 0-6. The NPS is calculated by subtracting the percentage of detractors from the percentage of promoters. Companies can use the NPS to track changes in consumer loyalty over time, between regions, or between themselves and competitors. We used the NPS as an indicator of reader-perceived comprehensibility of the AMPC.

Next, the survey used two open questions to identify positive aspects and improvement potentials for the AMPC. We formulated the questions as open as possible to allow for broad feedback.

The fourth site dived deeper into the applicability of the AMPC. On the one hand, the survey asked the participants how they would use the AMPC in practice. Additionally, it asked the participants if they use or know any other structured approaches for API management. The later question aimed to identify additional approaches we could use to enrich the AMPC. Also, in chapter 1.1, we claimed that currently, there are no holistic API management approaches in research or practice. Consequently, the question also aimed at validating this statement.

The next page aimed to collect participant information to determine the competency and heterogeneity of participants. The questions asked about the participant's role and industry affiliation of their organization[1]. In addition, we inquired about the participants' professional experiences in IT in general as well as with integrating and providing public Web APIs.

Finally, on the last page of the survey, the participants could provide any final remarks they wished to communicate before submitting the survey.

None of the questions in the survey were mandatory to answer to advance to the next question. Hence, participants could leave a question unanswered if they were unsure or did not want to provide any information on a specific topic. The rationale behind this decision was that we did not want participants to abandon the survey if they did not wish to answer a question. Moreover, the participants were able to terminate the survey at any time. However, we only analyzed the responses of participants who finished the survey.

The survey used the Unipark[2] survey software.

7.1.3. Survey Participant Acquisition

We contacted 52 IT professionals comprising mainly industry contacts of the Software Engineering for Business Information Systems (sebis) chair that previously participated in research on API management. In addition, we contacted professionals that the researcher and author of this dissertation met during her professional life, especially Enterprise Architects.

We used emails to contact the professionals. The emails contained the AMPC as an attachment and the link to the survey. Since the AMPC is very long, we told the professionals that it is sufficient to check the overview of pattern relations and skim one pattern before answering the

[1]We had to exclude a question about the organizations' size due to a mistake in the drop-down that did not allow the participants to select a specific range of employees.

[2]`https://www.unipark.com/umfragesoftware-bestellen/`

survey. The goal of this measure was to increase the likelihood of participation. Also, we asked the participants to forward the request to colleagues interested in the AMPC and participation in the survey.

We contacted the first professionals starting on 08.07.2023. We sent a reminder after three weeks to most participants, excluding those who confirmed that they already answered. The survey closed after eight weeks on 03.09.2023.

7.2. Survey Participants

Overall, 18 participants finalized the survey. In the following, we present an overview of the participants' characteristics to evaluate the suitability and relevance of the sample.

First, Fig. 7.1 presents an overview of the survey participants' roles. Six of the participants were Enterprise Architects, thus forming the largest group. In addition, four Software Developers, two Heads of IT, two Technical Architects, and one Solution Architect participated in the evaluation. Moreover, three participants stated that they had roles not listed in the drop-down of the survey. One of these participants is a Product Owner, and one is a Research Associate. Most interestingly, one participant's role is *Business Model Type Officer for Services*[3], a role created specifically to push service development in the respective organization. As a result, overall, there is a bias towards architecture roles. However, architects are likely involved in API design and management, thus forming a critical stakeholder group.

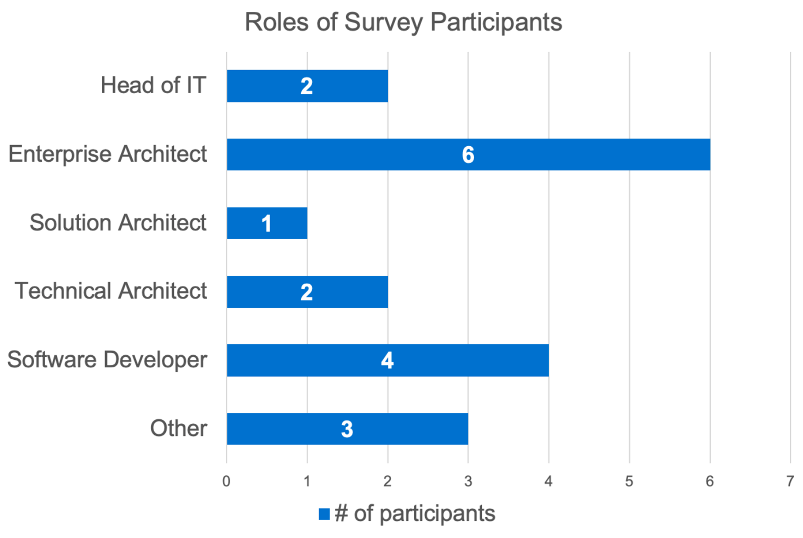


Figure 7.1.: Survey participants' roles.

Another goal of the AMPC is to present patterns for organizations that aren't relatively young tech giants. Therefore, we present an overview of the industry affiliation of the participants

[3]The participant provided the role name in German as "Geschäftsmodelltypverantwortlicher Services".

at the time of the survey in Fig. 7.2. As illustrated, the participants were primarily employed in the financial/insurance and the software services industry, with six and five participants, respectively. Furthermore, two participants worked each in the electronics and automotive, and one participant each in the transport and education/research sectors. Finally, one participant chose the category "other" but did not specify the sector afterward. As a result, there is a tendency towards more digital industry sectors.

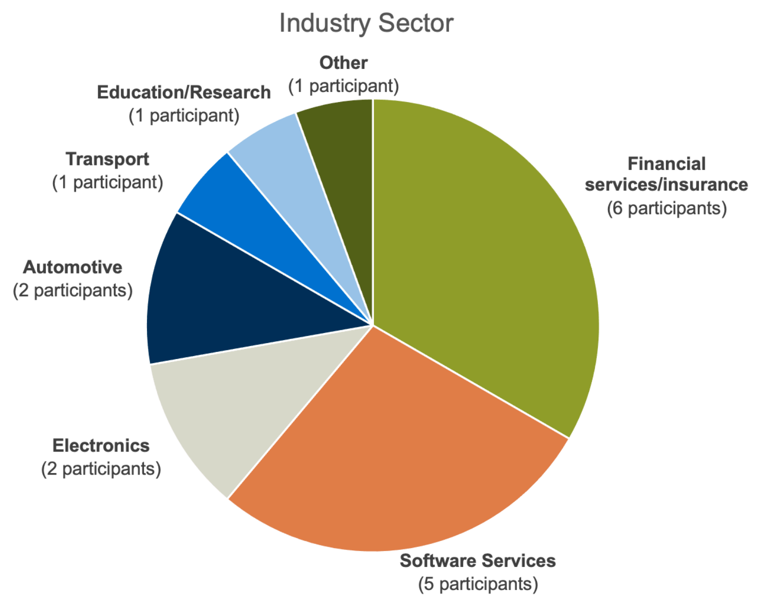


Figure 7.2.: Survey participants' industry sector affiliation.

Next, we analyzed the experience of participants to evaluate their competency. First, we asked the participants how long they professionally worked in IT. As visible in Fig. 7.3, the participant with the least experience worked in IT for five, and the participant with the most experience worked in IT for 27 years. The average professional IT experience across the sample is 14 years, and the median is 13.5 years.

Finally, we asked the participants about their experience consuming or providing Web APIs across organizational boundaries. As visible in Fig. 7.4, 11 participants had experience in both providing and consuming public, partner, and group Web APIs. In addition, four participants stated that they integrated Web APIs but do not have experience proving them. In comparison, one participant had experience providing public, partner, and group Web APIs but not consuming them. Only two participants lacked experience with either using or providing Web APIs across organizational boundaries. While these participants could not give any substantiated feedback on the content of the patterns, they reviewed the AMPC from the perspective of a novice trying to understand the topic of Web API management. Therefore, these participants could provide valuable feedback on the comprehensibility and usability of the AMPC. Consequently, we included all survey responses in the evaluation.

Overall, the sample represents experienced IT practitioners. The sample is biased toward Enterprise Architects and Software Developers, which are roles heavily involved in consuming and

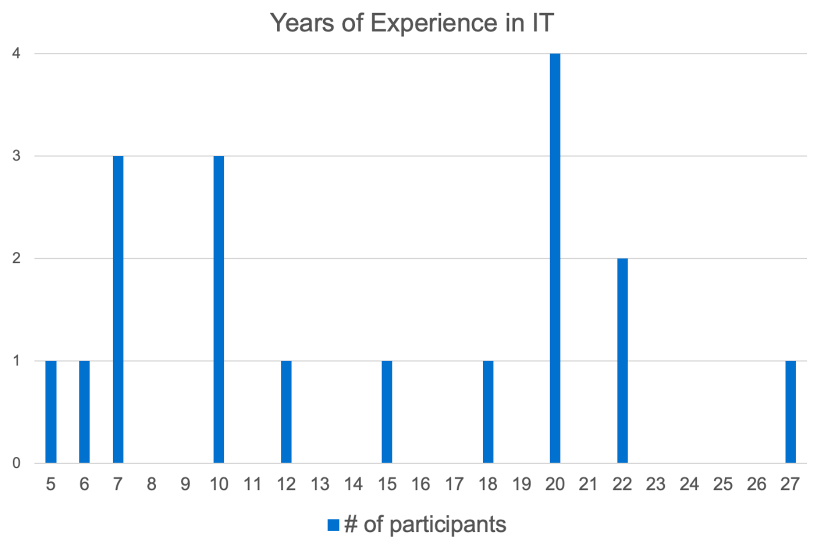


Figure 7.3.: Survey participants' professional IT experience in years.

providing Web APIs. In addition, many participants were employed in the insurance and financial services or software services industry, i.e., digital sectors. Finally, most participants have experience with both or either consuming or providing Web APIs. Hence, the sample is relevant and suitable to evaluate the AMPC.

7.3. Evaluation Results

In this section, we first present quantitative evaluation results, followed by qualitative results.

7.3.1. Quantitative Results

Fig. 7.5 gives an overview of the quantitative evaluation results.

First, we asked the participants to indicate if the structure of the AMPC is easily understandable. As visible in Fig. 7.5, nine participants strongly agreed, and six agreed. However, one participant was neutral, and two participants even disagreed.

The next two questions aimed to evaluate if the level of detail of the AMPC meets the readers' needs. The evaluation results show that eight participants strongly agreed, and another eight agreed that the AMPC has an adequate level of detail. In addition, two participants rated the statement as neutral. Moreover, three participants strongly agreed, and nine agreed that the AMPC does not hold any unnecessary information. Four participants were neutral towards the statement. However, two participants disagreed.

Next, the participants rated their need for the support of an author to use the AMPC. As Fig. 7.5

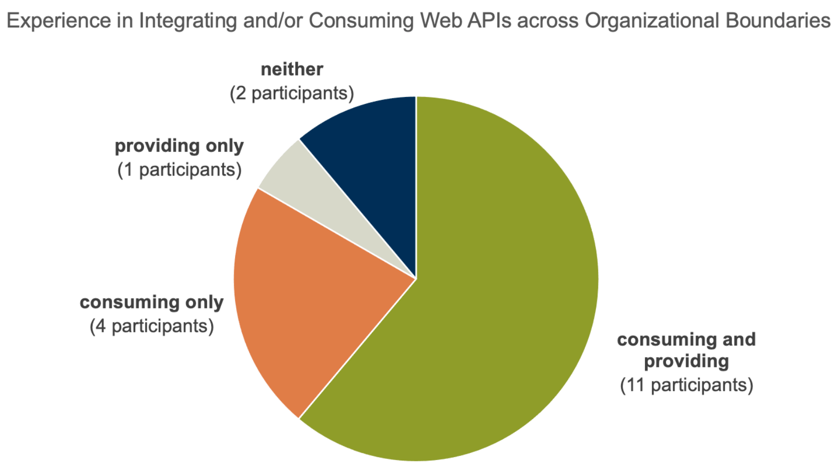


Figure 7.4.: Survey participants' experience consuming and providing Web APIs.

shows, three participants strongly disagreed, while nine disagreed. Three more participants were neutral towards the statement. However, three participants agreed, thus indicating that they desire support from an author.

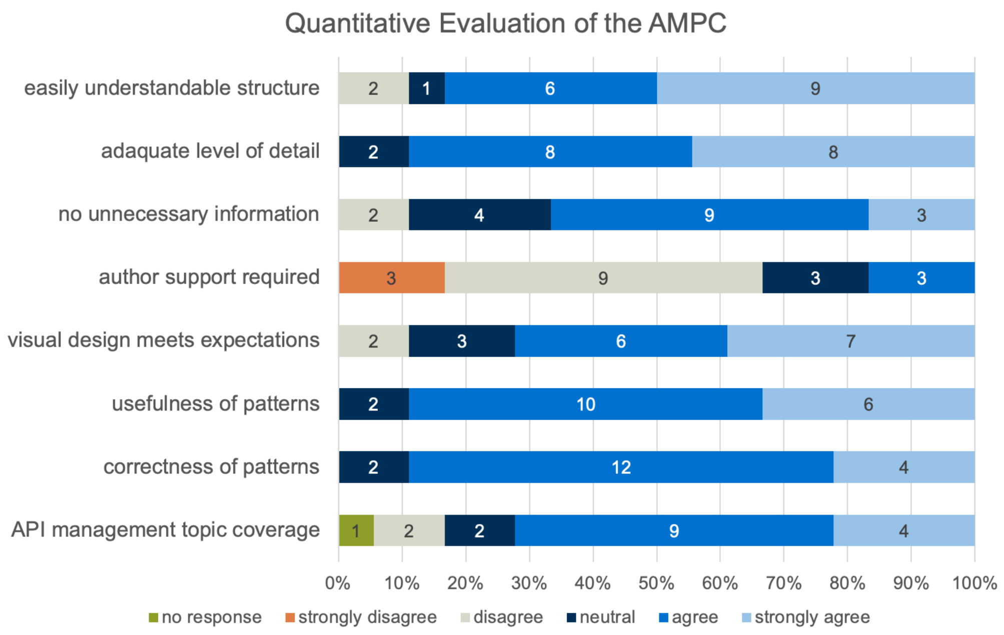
Furthermore, we asked the participants if the visual design of the AMPC meets their expectations. According to Fig. 7.5, seven participants strongly agreed, and six agreed with the statement. Three more participants were neutral towards the visual design of the AMPC, and two stated that the AMPC's visual design did not meet their expectations.

Aiming to evaluate the applicability of the AMPC, the participants had to rate if the AMPC provides useful patterns that they can apply in their organization. As a result, six participants strongly agreed, and ten agreed to the statement. In addition, two participants were neutral toward the statement.

Also, we quantitatively evaluated the correctness of the AMPC by asking the participants if the patterns correctly capture the essence of API management problems and solutions. A large majority of twelve participants agreed with the statement, and four more strongly agreed. Two participants were neutral.

Another goal of the survey was to evaluate the completeness of the AMPC. As visible in Fig. 7.5, four participants strongly agreed, and nine agreed that the AMPC covers all major topics. However, two participants were neutral, and two more disagreed, i.e., were missing topics in the AMPC. In addition, one participant did not rate the statement at all.

Next, we used the NPS (Reichheld, 2003) as an indicator of the comprehensibility and usability of the AMPC. As visible in Fig. 7.6, ten respondents rated the likelihood of recommending the AMPC to colleagues with a nine or ten. Thus, they are promoters. Promoters are customers who are satisfied to a degree that they vouch for a company with their reputation by referring



	no response	strongly disagree	disagree	neutral	agree	strongly agree
easily understandable structure			2	1	6	9
adaquate level of detail				2	8	8
no unnecessary information			2	4	9	3
author support required		3	9	3	3	
visual design meets expectations			2	3	6	7
usefulness of patterns				2	10	6
correctness of patterns				2	12	4
API management topic coverage	1		2	2	9	4

Figure 7.5.: Quantitative results of the AMPC evaluation.

it to others (Reichheld, 2003). In addition, seven participants were passively satisfied since they chose values seven or eight. Finally, one participant rated the question below seven and is, therefore, a detractor. Since the NPS is calculated by subtracting the percentage of detractors from the percentage of promoters, the NPS is 50%.

However, we do not know the NPS for a competing API management approach. Hence, we have no value to which we can compare the AMPC's NPS. Also, we applied a slightly wrong scale[4], further diminishing the meaningfulness of the result. Nevertheless, the author argues that the high NPS hints at the increased interest and perceived value of the AMPC for practitioners.

7.3.2. Qualitative Results

Next, we evaluate the qualitative feedback provided by the survey participants. To enable reproducibility and transparency of the results, we assigned an ID to each survey participant, i.e., P1-P18. We used five open questions to elicit feedback. These open questions were formulated very openly, thus accommodating different perspectives[5]. The following presents the positive

[4]We used a scale from 1-10 in the survey instead of 0-10 as specified in Reichheld (2003).

[5]Due to the openness of the questions, in some cases, participants responded to one question across various response fields. For example, instead of listing missing API management topics when asked for missing topics, several participants mentioned missing API management topics when asked for improvement potentials. Therefore, we analyzed the responses across all response fields for each question.

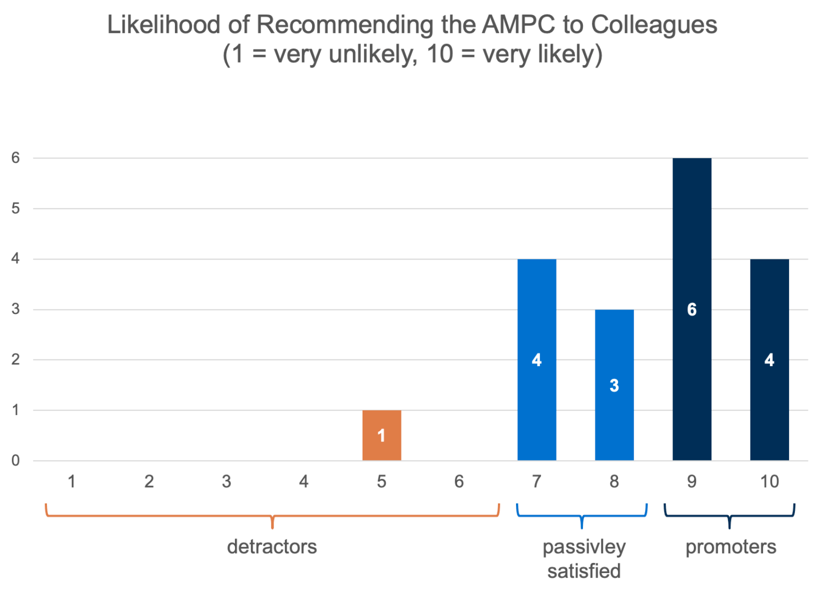


Figure 7.6.: Results to the question if a survey participant would recommend the AMPC to colleagues forming the basis to calculate the NPS (Reichheld, 2003).

aspects and improvement potentials of the AMPC. Also, we present additional topics that the AMPC should cover according to the survey responses. Afterward, we identify activities the AMPC could support in an organizational setting. Finally, we present API management approaches in use in the participants' organizations. If suitable, we include exemplary survey responses[6].

Positive Feedback on the AMPC

We asked the participants to describe what they liked about the AMPC. Fourteen participants provided answers to the question. In addition, four participants left positive feedback as a final remark on the last site of the survey. We grouped the responses into seven categories. We detail each category in the following.

Makes Pattern Relations Explicit. Several participants praised the relation of patterns and the visual presentation of these relations (P1, P4, P8, P9, P13, P10). For example, P8 states:

> *"Clearly laid out sections for each pattern type, excellent visual showing of interrelations between patterns and types."* (P8)

Overall, three participants liked the clear relation of patterns (P4, P8, P9, P13), two participants liked the color coding (P9, P10), one participant highlighted the different entry points

[6]We quote the participants' responses but made minor adjustments that do not change the responses' meaning to improve readability, including fixing spelling errors, adjusting the case sensitivity, and fixing wrong punctuation.

for stakeholders (P9), and one participant positively mentioned the categorization by domains (P16). Also, the overview of related patterns can inspire API providers (P1, P4).

Addresses Real-World API Management Problems. Several participants pointed out that the pattern catalog addresses real-world API management problems (P2, P4, P5, P6). For example, P4 states:

> *"Various patterns we use in our software ecosystem are listed and are described quite applicable with context, concern, and many more, as well as providing relations to similar patterns."* (P4)

The patterns identified in ongoing projects or initiatives were expected to provide a starting point for readers to discover further information on their implementation (P4, P5) and potentially useful related patterns (P4).

Supports Communication and Collaboration. Two participants mentioned the usefulness of patterns in communicating and collaborating with internal (P2, P6) and external stakeholders (P6). P2 argues:

> *"Having patterns already applied in our API gives me more context and helps me to better communicate to internal stakeholders about their relevance"* (P2)

Also, patterns are deemed to help identify responsible or knowledgeable persons to solve API management problems (P16).

Broad Coverage of API Management Topics. P6 and P8 positively mentioned the AMPC covering various API management topics. As P8 states:

> *"Very comprehensive and thorough examination of the field. Truly amazing work in gathering and organizing this pattern catalog. Will certainly be invaluable as reference material."* (P8)

However, P12 argues that it is incomplete and doesn't have to be complete at this point in time. Instead, the AMPC is a starting point for discussing API management topics that needs to evolve over time.

Highlights Organizational and Process Related Aspects. Participants liked that the AMPC focuses on organizational and process-related aspects in addition to technical patterns (P10, P11). For example, P11 states:

> *"It shows, besides the technical pattern, the often overlooked non-technical, especially organization requirements for API management like testing, monitoring but also having a product owner and support team for it."* (P11)

Furthermore, P15 highlighted the importance of monitoring and testing in API management.

Makes Conflicts and Consequences Explicit. Moreover, two participants mentioned the value of the explicitly laid out forces and their resolution in the consequences section (P13, P14). As P14 states:

> *"I found it quite helpful that conflicts are clearly shown. It helps to elaborate the impact when choosing between the pattern options."* (P14)

Provides Implementation Guidance. P5 mentioned the usefulness of the section 'Implementation Hints' providing implementation suggestions. Similarly, P1 highlighted the usefulness of the 'Known Uses' section to guide pattern implementation:

> *"When we search for an implementation for a pattern, we can look at the Known Uses to see how other companies approached this. [...]"* (P1)

Finally, two participants (P1, P4) left encouraging and thanking remarks on the last site of the survey. One such a remark was:

> *"Good luck and thank you, it was an interesting study!"* (P1)

Improvement Potentials

We asked the participants if there are any improvement potentials for the AMPC. Eleven out of eighteen participants submitted responses to this question. The improvement potentials span seven aspects detailed in the following.

Access, Navigation, and Searchability. The usability of the AMPC is limited by its current form as a Portable Document Format (PDF) document (P4). Instead, the pattern catalog would profit from a format that allows for easy access (P10), searchability (P10, P16), hiding and filtering (P9), and links between the pattern overview and the respective pattern descriptions (P4, P8, P9). Such features would reduce the cognitive load for readers of the AMPC (P9). The authors could realize these features by presenting the AMPC as an Hypertext Markup Language (HTML) document (P4).

Completeness. Several participants mentioned topics missing in the AMPC (P11, P12). We will detail these topics in Section 7.3.2.

Also, practice-driven best practices should find their way into the AMPC, as P11 stated:

> *"The realization of APIs with API management software and best practices derived from API management tool providers (like Mulesoft or Axway) is missing"* (P11)

API Maturity. The AMPC does not consider different levels of an API's maturity. Therefore, P15 requested:

> *"For the implementation part, I would like to see more explicit the evaluation of the maturity of the API as a process (i.e., going from chaotic to production)."* (P15)

Tangible Real World Examples. While P1 perceived the 'Known Uses' section as useful, P5 asked for more tangible real-world examples, i.e., states:

> *"Some more tangible real-world examples of the API pattern would make it easier to relate."* (P5)

Regular Updates. The patterns require regular updates (P10). Hence, the authors of the AMPC must design and communicate the approach to regularly updating the AMPC. At the same time, the AMPC's authors need to make changes to patterns over time transparently (P16).

Scoping. P12 argued that the AMPC should delineate different software communication patterns and clarify which ones are in the scope of the AMPC. Such clear scoping would increase the readers' understanding.

Domain assignment. Finally, P16 questioned the definition of domains in the overview of related patterns. Moreover, P16 requested a clear differentiation of domain and standard patterns.

Missing Topics

The survey participants reported the following API management topics as missing in the AMPC:

- Security (P3, P11)
- Payment (P11)
- Realization of APIs with API management software (P11)
- Communication patterns other than request/response, e.g., publish/subscribe (P12)
- Contract-first and consumer-driven contracts (P12)
- API maturity evaluation (P15)

Furthermore, two participants proposed enriching pattern descriptions with additional information. First, P11 proposed to add information derived from best practices put forward by API management tool providers. Secondly, P13 asked to add industry-standard best practices to the pattern descriptions.

Nevertheless, two participants indicated that the AMPC covers the public API management domain well (P6, P8). For example, P8 stated:

> *"Very comprehensive and thorough examination of the field."* (P8)

Finally, P12 commented that the AMPC needs to evolve over time to become complete:

> *"As stated in the introductory chapters: It fills a gap. In this regard, it is not necessary to be complete in the sense of a pattern language. The catalog may serve as a starting point for discussions, and it may be amended over time."* (P12)

Application Situations

We wanted to identify situations in which the readers can apply the AMPC in their organizations. Hence, the survey asked the participants to *"[...] describe concrete situations, issues, or occasions* *in which the pattern catalog would be useful"*. Thirteen of the eighteen participants provided feedback as text. Overall, we derived six organizational activities that the AMPC could support from the responses. We present each of these activities in the following.

Design of a New API. The participants confirmed the usefulness of the AMPC during the design and implementation of new APIs (P2, P6, P10, P13). Several participants focused on the AMPC's value when offering a previously internal API product externally (P11, P14). Also, the AMPC supports the design of APIs as products (P5, P6).

Moreover, the overview of patterns (Fig. 6.5) can inspire the design of APIs (P1). In addition, the AMPC supports prioritizing backlog items during API implementation (P2). Finally, P1 states that the 'Known Uses' section helps review how other companies implement a pattern.

Improvement of an Existing API. The AMPC can improve the management of an already exposed APIs. As P11 stated, the AMPC is useful:

> *"to identify gaps of a given API-based product/approach in order to achieve monetarization (or simply usage/traffic) goals"* (P11)

Design of an Internal API Marketplace. One participant pointed out the applicability of the AMPC for creating an internal API marketplace (P8).

Definition of Roles and Responsibilities. The AMPC can help define clear roles and responsibilities within (P6) and between organizations (P9, P12). As P9 stated:

> *"Once there is an opportunity for collaboration across different organizations, clearly defined roles and patterns to follow can be of support"* (P9)

API Management Training. The AMPC can be used to educate employees about APIs (P2). Beginners will profit from the overview of patterns (P4). Moreover, the link to external patterns is useful to gain further insights into API management (P4).

Vendor and Technology Analysis. The AMPC can support evaluating vendors of new capabilities (P8). Further, the AMPC can help ensure that organizations use only proven technologies (P10, P16).

However, P10 claimed that the usefulness of the AMPC is contingent on training:

> *"I think that it is crucial to train the organization what it means to use API pattern catalog - as well as about the benefit."* (P10)

API Management Approaches in Use

In addition, we asked the participants if they know or use any structured approaches to facilitate API management in their organizations. Ten out of eighteen participants responded to the question. Four participants (P2, P10, P13, P8) stated that they currently do not use a structured approach to API management. Furthermore, two participants indicated they use a proprietary structured approach that is still maturing (P5, P6). Finally, P9 answered that they use technical standards like Swagger[7] and GraphQL[8] (P9). Similarly, P11 used well-known API management software like MuleSoft[9] or Axway[10] to guide their API management. P12 stated that their organization uses API-first design and consumer-driven contract testing. Finally, P15 reported using a Value Stream Management product that discovers existing services' APIs and manually tags them as internal or external.

As a result, while some participants used no structured API management approach, others used technical standards, API management software, or even designed proprietary approaches. However, no participant claimed to apply a mature, holistic approach explicitly covering technical and social aspects of API management along the whole lifecycle of an API. Also, P8 emphasized the usefulness of a structured approach.

7.4. Discussion

The goal of a design science evaluation is to demonstrate the *"utility, quality, and efficacy of a design artifact"* (Hevner et al., 2004, p. 85). Design science researchers can do so by evaluating relevant quality attributes derived from the business environment in which it should be applied (Hevner et al., 2004). Since the AMPC aims to support IT professionals in organizational settings in API management activities, we evaluate the applicability, comprehensibility/usability, completeness, and correctness of the AMPC. Furthermore, we reviewed the validity of the research gap this thesis aims to address.

In the following, we summarize the evaluation results for each of these dimension.

Applicability

First, we summarize the evaluation results that indicate if readers can apply the contents of the AMPC in real-world settings, i.e., if the patterns are useful in supporting their professional day-to-day work.

First, all participants agreed or were at least neutral towards the statement that the AMPC provides useful patterns that they can apply in their organization. Moreover, the participants repeatedly mentioned that a strength of the AMPC is that it addresses real-world API management problems (P2, P4, P5, P6). Nevertheless, providing more real-world examples would increase the AMPC's relevance (P5). Also, the participants deemed the provided implementation guidance (P1, P5) and the explicit conflicts and consequences (P13, P14) as useful. The focus on organizational and process-related aspects besides technical aspects was also positively highlighted (P10, P11). Finally, the patterns were expected to help support communication and collaboration with internal and external stakeholders (P2, P6).

In addition, the evaluation participants named six API management activities that the AMPC

[7]https://swagger.io/
[8]https://graphql.org/
[9]https://www.mulesoft.com/de/
[10]https://www.axway.com/de

could support. These activities span the design of new (P1, P2, P5, P6, P10, P11, P13, P14) and improvement of existing (P11) public, partner, and group Web APIs. Also, even though outside scope, the AMPC can support the creation of internal marketplaces (P8). In addition, the AMPC could support the definition of roles and responsibilities in API initiatives (P6, P9, P12), API management training (P2, P4), and vendor and technology analysis (P8, P10, P16).

Thus, generally, the participants classified the AMPC as useful and applicable in real-world settings. However, we derived these findings from a survey with professionals judging the AMPC instead of actually applying it in real-world settings. Therefore, future work should comprise action research using the AMPC to guide each of the activities mentioned above in real organizational settings.

Comprehensibility/Usability

Next, we summarize the evaluation results on the AMPC's comprehensibility, including its usability. Hence, this section also takes into account the style (Hevner et al., 2004) and form of the AMPC.

First, 15 participants agreed that the structure of the AMPC is easy to understand, while two disagreed. Similarly, 12 participants stated that they do not need the support of an author to use it, while two said they would need an author's help. However, these participants did not indicate why they would require such support. Also, the NPS of 50% suggests an increased usability and perceived value of the AMPC.

Furthermore, 13 participants agreed that the AMPC's visual design did meet their expectations. Especially the visual presentation of the relations between patterns was perceived positively (P1, P4, P8, P9, P10, P13, P16). Nevertheless, two participants were not satisfied with the visual design of the AMPC. This might be due to the AMPC's medium. Several participants (P4, P8, P9, P10, P16) complained about access, navigation, searchability, and filtering of the AMPC due to its current form as a PDF file. More interactive formats, e.g., presenting the AMPC as an HTML document, would improve its usability.

Overall, the participants evaluated the AMPC's usability as primarily positive. However, there are also some critical voices, mainly pertaining to AMPC's current form as PDF.

Completeness

We wanted to evaluate if the AMPC covers all relevant API management topics that a practitioner faces. In addition, completeness pertains to the coverage of information in each pattern description.

As described in Chapter 6, we derived the patterns of the AMPC from 12 cases describing API initiatives. We applied the rule of three (Coplien, 1996) to decide if a pattern candidate is a pattern, i.e., we need to observe a pattern candidate in at least three settings for it to become a pattern. Hence, the data basis to derive patterns was limited.

Therefore, several evaluation participants named topics missing in the AMPC. These topics cover security (P3, P11), payment (P11), use of API management software (P11), other communication patterns besides messaging (P12), contract-first and consumer-driven contracts (P12), and API maturity evaluation processes (P15). These topics span technical as well as collaborative aspects of API management. Moreover, respondents requested to include tool providers and industry-standard best practices in the AMPC (P11, P13). This finding supports the finding of Mathijssen et al. (2020), stating that API management is an organizational function often realized through an API management platform. Finally, one participant requests an additional differentiation between standard and domain patterns (P16).

Nevertheless, the quantitative results also show that 13 participants agreed that the AMPC is complete, while only two participants disagreed. In addition, one participant (P8) highlighted the broad coverage of API management topics in the AMPC. Finally, one participant also argues that the AMPC would profit from more precise scoping (P12). Hence, while the AMPC seems to cover many important topics of API management, it is not a complete collection of API management best practices.

Next, we dive into the completeness of pattern descriptions. Overall, 16 participants agreed that the AMPC presents an adequate level of detail. Similarly, 12 participants agreed that the AMPC does not hold any unnecessary information. Nevertheless, two participants disagreed with this statement. Hence, we assume the level of detail of the pattern descriptions mostly meets the readers' needs, potentially even holding too much information.

In summary, the AMPC does not present a complete pattern language of API management activities but provides a starting point that requires extension, refinement, and regular updates (P10, P12, P16). While the pattern catalog is not complete with regards to the patterns, the individual pattern descriptions meet the readers' information needs and are potentially even a little too detailed.

Correctness

Another goal of the evaluation is to judge if the AMPC's contents are correct. Since the AMPC describes best practices, it is not possible to evaluate the correctness using formal metrics. However, the AMPC derives the patterns from the descriptions of real-world API initiatives. In addition, all survey participants agreed or were neutral toward the correctness of AMPC. Therefore, we assume that most contents of the AMPC capture API management practices and their relations correctly.

Research Gap

In Section 1.1, we stated that to the best of the author's knowledge, currently, there is no holistic approach to API management in research or practice. To ensure that we did not miss an approach, we asked the survey participants if they know or use any such approach in their organization. Most participants did not respond (P1, P3, P4, P7, P14, P16, P17, P18) or responded with "no" (P2, P8, P10, P13). In addition, two participants mentioned applying partial approaches to API management, e.g., API-first design (P12) or automated API discovery

(P15). Two more participants use technical standards (P9) and let API management software guide their API management (P11). Finally, two participants use a proprietary approach, that is still maturing (P5, P6). Sadly, they did not provide any additional information on what the approach entails.

Hence, no participant knew or applied a mature, holistic, and structured approach to API management. Thus, the evaluation results confirm the lack of a recognized holistic API management approach explicitly covering technical and social aspects of API management along the whole lifecycle of an API.

7.5. Summary

In the prior chapter, we detailed the iterative design of the AMPC. This chapter aimed to evaluate the AMPC concerning its applicability, comprehensibility/usability, completeness, and correctness from a practitioner's viewpoint. We contacted 52 IT professionals and asked them to participate in an anonymous online survey to achieve this goal. Overall, 18 practitioners responded to the survey. The sample was dominated by Enterprise Architects and Software Developers as well as participants employed in the insurance and financial services and software services industry. Also, the sample represented experienced IT practitioners.

The analysis of the survey responses showed that, generally, the participants classified the AMPC as applicable in real-world settings. Moreover, the participants named six activities that the AMPC could support. These activities span the design of new (P1, P2, P5, P6, P10, P11, P13, P14) and improvement of existing (P11) public, partner, and group Web APIs. Also, even though outside scope, the AMPC could enable the creation of internal marketplaces (P8). In addition, the AMPC could support the definition of roles and responsibilities in API initiatives (P6, P9, P12), API management training (P2, P4), and vendor and technology analysis (P8, P10, P16).

The survey participants evaluated the AMPC's comprehensibility and usability as primarily positive. However, a downside is the current form as PDF, which limits its searchability and navigability.

Concerning completeness, the survey respondents mentioned several topics missing in the AMPC, for example, best practices concerning security and payment. However, the AMPC is also not meant to be a complete pattern language at this point in time. Instead, it is a starting point for designing a holistic approach to API management that requires extension, refinement, and regular updates in the future. On the other hand, the individual pattern descriptions met the readers' information needs and are potentially even too detailed.

Also, the survey participants perceived the contents of the AMPC as primarily correct.

Finally, we asked the participants if they know or currently use a structured approach to API management in their organization. Two participants stated that they are using a proprietary approach that is still maturing. The other participants did not know or apply a holistic, structured approach to API management. Thus, the evaluation results confirm the lack of a recognized holistic API management approach explicitly covering technical and social aspects of API management along the whole lifecycle of an API.

Overall, a limitation of the survey evaluation is that it relies on expert opinions. Thus, future work should comprise further evaluation of the AMPC based on its actual application in organizational settings.

CHAPTER 8

Conclusion and Future Work

Web Application Programming Interfaces (APIs) are the de facto standard for making data and functionality accessible across organizational boundaries. Using public, partner, and group Web APIs enables API providers to realize new business models (Evans and Basole, 2016; Basole, 2016, 2019), create platforms (Ghazawneh and Henfridsson, 2010, 2013; Eaton et al., 2015; Karhu et al., 2018; de Reuver et al., 2018), integrate partners efficiently (Hagel III and Brown, 2001), and achieve compliance (Bondel et al., 2021a; ISO 20077-1; ISO 20078-1; ISO 20080). However, API provision is currently dominated by relatively young digital organizations in the US (Huhtamäki et al., 2017). Thus, more established organizations in traditional industry sectors located in Europe want to tap into the potential of becoming API providers.

Successful Web API provision requires careful design (Yoo et al., 2010) and management. Since Web APIs used across organizational boundaries are resources at the interface between the API provider team and stakeholders inside and outside the provider organization, API management is an inherently collaborative organizational function. Therefore, this dissertation aimed to identify API management best practices and patterns focusing on knowledge transfer and collaboration for different types of API provider organizations, including established organizations in traditional sectors located in Europe.

To the best of the author's knowledge, no pattern collection explicitly focusing on API management patterns exists. Therefore, we identified API management patterns in API design, service design, middleware design, object-oriented software design, and software architecture pattern collections. The analysis yielded that existing API management patterns focus mainly on the technical solutions to API management challenges. These patterns often concern API testing, security, performance, monitoring, and evolution.

In settings with API providers and consumers belonging to different organizations, the API provider has to transfer knowledge about an API to potentially unknown, heterogeneous, and distributed API consumers with different goals. Hence, this dissertation's first significant contribution is identifying and evaluating best practices for code examples in API provider-generated documentation.

First, we identified 48 best practice candidates for code examples in API documentation. During the process, we realized that the effect of code examples on API consumer productivity and satisfaction depends not only on the knowledge that a code example transfers but also on its form. Moreover, code examples are expected to transfer knowledge about API execution facts and usage patterns. Also, API consumers look for information about documentation quality to judge an example's reliability and manage their expectations.

Next, in a case study, we evaluated a subset of eight best practice candidates for code examples in public, partner, and group Web API documentation. As a result, we validated six best practice candidates as actual best practices as they positively affected the developers' productivity and perceived usability. Moreover, we observed that low-quality code examples in Web API documentation force developers to use a trial-and-error approach. Such a trial-and-error approach is typical for opportunistic developer personas but can hamper systematic developer personas' productivity and perceived satisfaction. Furthermore, the effect of specific best practices or best practice candidates for code examples in public, partner, or group Web API documentation depends on the context in which they are applied.

This dissertation's second and significant contribution is the design and evaluation of the AMPC. The AMPC presents 22 related patterns, 37 pattern candidates, and nine stakeholders derived from 14 case descriptions. We enriched the pattern descriptions with information from scientific and practice-driven literature and improved the pattern form with feedback from the scientific pattern community. Finally, we evaluated it from a practitioner's perspective using a survey. The practitioners generally perceived the AMPC as applicable to real-world settings, comprehensible, usable, and correct. However, it is not complete at this point in time. Instead, it provides a starting point for designing a holistic approach to API management that requires extension, refinement, and regular updates in the future. Hence, future work should identify further patterns and evaluate the application of existing patterns in real-world settings. Also, the documentation of future changes to the pattern catalog will allow researchers to create knowledge on the evolution of the Web API management discipline.

Furthermore, we made several observations during the design of the AMPC. First, API consumers often discover and try out Web APIs through software artifacts controlled by the API provider team, i.e., the developer portal. Nevertheless, API consumers often want personal contact with API providers before and during API integration. Also, collaboration between the API provider team and other internal stakeholders can be challenging, e.g., in some cases, the API provider team and the backend teams prioritize issue resolution affecting Web APIs differently. In addition, the API provider has to treat the API as a product with a lifecycle. Finally, the strategic relevance of provided functionality and the structure of a consumer organization seem to influence the applicability of patterns significantly.

8.1. Answers to Research Questions

In the following, we summarize the answers to the research questions introduced in Chapter 1.

RQ1: What is the current state of research on API management?

RQ 1.1 *How is API management defined in academia?*
As previously stated by Mathijssen et al. (2020), and to the best of the author's knowledge, in research, no framework or overview provides a holistic view on API management. Also, no comprehensive and widely accepted definition of API management exists. Hence, we presented a working definition of API management and an API management lifecycle in Section 2.4. The definition specifies API management as an organizational function that comprises all activities of an API provider (team) aiming to provide a successful Web API, including technical and social aspects.

RQ 1.2 *What patterns for API management exist in research and practice?*
An extensive review of pattern collections yielded the result that to the best of the author's knowledge, currently, no API management pattern collections explicitly focusing on API management exist. Instead, we analyzed API design including SOA and microservices pattern collections, service design, middleware design, object-oriented software design, and software architecture pattern collections to identify API management patterns. Consequently, we identified API management patterns within these pattern collections, but they focus primarily on technical solutions for API testing, security, performance, monitoring, and evolution.

RQ2: What are best practices for transferring knowledge to API consumers using code examples in official public, partner, and group Web API documentation?

RQ 2.1 *What are best practice candidates for code examples in official public, partner, and group Web API documentation?*
In settings where the consumers do not have direct access to the Web API developer team, an approach to knowledge transfer is the provision of documentation. An analysis of 17 research papers and 13 expert interviews yielded 48 best practice candidates for code examples in Web APIs documentation described in Chapter 4. Moreover, we categorized these best practice candidates according to the type of knowledge they aim to transfer or the form they prescribe. Also, we derived several implications from observations made while collecting the best practice candidates.

RQ 2.2 *What are validated best practices for code examples in official public, partner, and group Web API documentation?*
Drawing on the list of identified best practice candidates in Chapter 4, we chose eight candidates with little or no support in Web API-specific literature or contradicting statements about their impact on the API consumers' productivity and perception. As detailed in Chapter 5, we used a case study with 12 professional developers using two different documentation versions to solve tasks using a GraphQL API. As a result, we confirmed six best practice candidates to be actual best practices. Moreover, we observed that the applicability of best practices seems to depend on the API's context, e.g., the learning strategies and domain knowledge of the developers, the complexity of the task, or the complexity of the API itself.

RQ3: What are API management patterns for public, partner, and group Web APIs focusing on collaboration?

RQ 3.1 *Who are the stakeholders involved in public, partner, and group Web API management?*
We identified the stakeholders by collecting and analyzing 14 cases in the context of a design science research approach (Hevner et al., 2004; Hevner, 2007). We identified nine stakeholders, comprising stakeholders directly responsible for or interacting with Web API software artifacts, i.e., the *backend provider*, *API provider*, *API consumer*, and *end user*. Further stakeholders involved in API management are the *upper management*, *legal*, *sales & marketing*, *customer support*, and *integration partners*.

RQ 3.2 *What are API management patterns for public, partner, and group Web APIs with a Focus on collaboration?*
We created the AMPC using a design science research approach (Hevner et al., 2004; Hevner, 2007) comprising the creation and analysis of a case base holding 14 cases derived from interviews and publicly available data on API initiatives. We presented a catalog of interrelated Web API management patterns, i.e., the API Management Pattern Catalog (AMPC) published in Bondel and Matthes (2023) and summarized in Sections 6.4.3. The pattern catalog presents 22 related API management patterns and 37 pattern candidates. Each pattern explicitly describes involved stakeholders and their interactions. Also, we collected feedback from the scientific pattern community to improve the pattern form.

RQ 3.3 *How do the identified API management patterns relate to existing pattern languages and catalogs?*
We related the patterns in the AMPC (Bondel and Matthes, 2023) to previously identified API management patterns in other pattern collections as summarized in Section 6.4.4. Many of these patterns document practices for the technical realization of Web APIs. However, we also identified and related some patterns documenting social, organizational, or business aspects of API management. Hence, the AMPC links API management patterns from different sources.

RQ 3.4 *How do practitioners perceive the usefulness of API management patterns for public, partner, and group Web APIs with a focus on collaboration?*
As reported in Chapter 7, we used an online survey with 18 participants to collect practitioner feedback on the applicability, comprehensibility, usability, completeness, and correctness of the AMPC. The practitioners perceived the AMPC as applicable to real-world situations, comprehensive, and correct. However, the usability of the AMPC would benefit from a more searchable and navigable form. Also, the pattern catalog is incomplete but provides a good starting point for developing a holistic and flexible approach to API management.

8.2. Limitations

Even though we conducted all research presented in this dissertation with the highest possible accuracy and conscientiousness, we recognize some limitations in retrospect. More precisely, we identified limitations regarding the definition of best practices, the considered material, the identification and evaluation of best practices for code examples in Web API documentation, and the research methodology, design, and assessment of the AMPC. In this section, we describe these limitations.

8.2.1. Limitations to the Identification of API Management Best Practices

According to Bretschneider et al. (2005), the identification of best practices relies on three conditions. These conditions are that the case base must be complete, the cases must be comparable, and a clear cause-and-effect relationship between practice and outcome must exist.

However, in the context of code examples in Web API documentation, it is impossible to identify, collect, and analyze all official Web API documentation. Moreover, isolating a clear cause-and-effect relationship between practices and outcomes is impossible.

Similarly, it is impossible to analyze all current public, partner, and group API initiatives to identify API management best practices. Also, many influence factors impact the success of Web APIs, making it impossible to isolate a clear cause-and-effect relationship between practices and outcomes.

Hence, according to Bretschneider et al. (2005), the practices for code examples in Web API documentation and the patterns in the AMPC do not meet the conditions of "best" practices but are "good" practices instead. However, as described in Section 2.5.1, it is common to refer to practices that have been observed to support the achievement of a specific goal in several cases as *best practices* in industry. Hence, we adopted this convention.

8.2.2. Limitations of the Considered Material

Web API management across organizational boundaries is an inherently interdisciplinary topic situated at the overlap among Information Systems Research (ISR), computer science, and management science. Moreover, Web API management touches on technical aspects like Web API implementation, testing, and monitoring, as well as organizational and social aspects guiding the collaboration with stakeholders inside and outside an API provider organization. Also, knowledge about Web API management is spread across academic, practitioner-driven, and vendor-driven information sources.

We executed extensive but partial searches focusing on API management concepts and stakeholders, best practices, and patterns in computer science literature while writing this dissertation. In addition, we applied forward and backward searches to identify further relevant literature. Furthermore, we incorporated practitioner-written books, prominent practitioner-written articles, and, where appropriate, information presented by API management platform vendors. Hence, we do not claim completeness of the sources included in this dissertation.

8.2.3. Limitations to the Best Practices for Code Examples in Web API Documentation

In the following, we detail limitations related to the identification and evaluation of best practices for code examples in Web API documentation

Limitations to the Identification of Best Practice Candidates for Code Examples in Web API Documentation

As described in Chapter 4, we conducted a thorough literature review and interviewed 13 professional developers with several years of experience to identify best practice candidates for Web API documentation.

Several limitations pertain to the number and characteristics of interviewed experts. First, we conducted a limited number of 13 expert interviews. More interviews could have led to the detection of further good practice candidates. Also, since all interviewees have several years of experience, we were not able to capture the potential requirements of developers with little or no expertise using code examples in Web API documentation. In addition, the interview experts belonged to the same industry partner organization, which could introduce a bias. The industry partner is a digital leader and provides several publicly accessible APIs. As a result, API consumers from other organizations might have different expectations for Web API code examples.

In addition, only one researcher executed the research approach, i.e., the analysis of the literature, the analysis of the interviews, and the categorization of the best practice candidates into knowledge types and form.

Limitations to the Evaluation of Best Practices for Code Examples in Web API Documentation

We used a case study to evaluate eight best practices candidates for code examples in Web API documentation as described in Chapter 5.

> "Overall, we [...] recruited 12 participants for the case study. Due to the low number of participants, the statistical validity of the findings is limited. Also, a larger sample of case study participants might have led to additional findings. However, all [...] case study participants are professional developers with several years of experience. Therefore we believe the results of this study to present valid insights. Nevertheless, future studies should also investigate the validity of the best practices for API consumers with less experience to take into account the varying levels of experience that API consumers might have.
>
> Moreover, all participants are employees of a single industry partner organization that provides several publicly accessible Web APIs. Hence, these developers might be more interested in API documentation than the average professional developer."
>
> Another limitation arises from the Web API we chose for the case study. The Compass API is embedded in a complex domain and is a GraphQL API. APIs of different complexity and using other technology might have different documentation requirements."
>
> – (Bondel et al., 2022)

Also, we selected participants for the case study that have not previously used the Compass API. Therefore, results might not generalize to settings in which developers are already familiar with a certain API.

Finally, looking at validation of each best practice candidate in Section 5.2.3, we need to emphasize that the evaluation results are based on indications. Since we apply all good practice candidates simultaneously, we cannot determine any direct correlation, let alone causality, between the implementation of specific good practice candidates and the improvement of the API consumers' productivity or perceived usability (Bondel et al., 2022).

8.2.4. Limitations to the AMPC

In the following, we describe limitations related to the research methodology, design, and evaluation of the AMPC.

Limitations to the Research Methodology of the AMPC

We applied a design science approach according to Hevner et al. (2004) and Hevner (2007) to design the AMPC as detailed in Section 6.1. According to Frank (2006), design science suffers from four flaws and misconceptions.

The first flaw is a *lack of accounting for possible future worlds.* Instead of inspiring decision makers and fostering innovation, design science focuses on reproduction and topics deemed relevant by others.

Secondly, design science suffers from *insufficient conception of a scientific foundation* due to not reflecting on *"basic ontological and epistemological assumptions"* (Frank, 2006, p. 30). Hence, a solid foundation for integrating and configuring the developed artifact is missing.

Thirdly, design science relies on a *mechanistic world view* with precisely defined requirements and the ability of heuristic searches to produce satisfactory results. However, in reality, requirements are contingent and heuristic searches do not necessarily yield satisfactory results.

Finally, Hevner et al. (2004)'s research paper indicates a *lack of appropriate concepts for describing the IT-artifact*, i.e., is lacking concepts for describing design decisions and features of the artifact.

Especially the first flaw, i.e., the lack of accounting for innovation, is visible in the AMPC. However, patterns aim to capture proven solutions to recurring problems by design (Gamma et al., 1995; Buschmann et al., 1996).

Limitations to the Design of the AMPC

We designed the AMPC using a design science approach (Hevner et al., 2004; Hevner, 2007) employing grounded theory methodology (Wiesche et al., 2017) to identify pattern candidates and patterns from 12 cases as presented in Section 6.3.

There are several limitations pertaining to the case base. First, the case base holds a limited number of 12 case descriptions derived from expert interviews. A broader case base could lead to the detection of further pattern candidates and patterns. Also, analyzing additional cases could validate and enrich already documented patterns (Bondel et al., 2021b).

Next, the case base primarily documents relatively new API initiatives of German, traditionally non-digital organizations (Bondel et al., 2021b). While the AMPC aims to identify patterns for different types of organizations, including German, traditionally non-digital organizations, the identified patterns might be biased towards this type of organization. Consequently, we used public information on the API initiatives of Twilio[1] and Stripe[2] to enrich the descriptions of previously identified patterns. However, we did not analyze these public API initiatives to identify additional patterns.

Furthermore, we derived the case base from interviews conducted over half a year between August 2020 and January 2021. Hence, the patterns provide a screenshot of API management practices at the time (Bondel et al., 2021b).

The data analysis was conducted by two researchers sequentially. The results of the first data analysis influenced the data analysis of the second researcher.

Moreover, we enriched the pattern descriptions with knowledge from research and practitioner-driven literature. Even though we conducted an extensive search to identify relevant pattern languages and catalogs, we did not employ a structured approach. Similarly, we extensively searched practice-driven API management literature. However, we included only publications expressively concerned with API management.

[1]`https://www.twilio.com/de-de`
[2]`https://stripe.com/de`

Finally, we refined the pattern form and pattern catalog structure by participating in shepherding and a pattern writers workshop, a standard and long-standing approach for pattern improvement in the pattern community (Coplien, 1996). While the shepherd and the peers who participated in the writers workshop are IS researchers, they are not experts in the API management domain. Hence, their feedback mostly pertained to the pattern and pattern catalog structure.

Limitations to of the Evaluation of the AMPC

As described in Chapter 7, we used an online survey to evaluate the applicability, comprehensibility, usability, completeness, and correctness of the AMPC from a practitioner perspective.

Overall, 18 participants finalized the survey. Due to the limited number of participants, the results are not statistically significant and their generalizability is limited.

Next, the survey mostly targeted contacts employed at enterprises seated in Germany with prior contact to the sebis chair or the author of this dissertation in the context of API management research. While participation was anonymous, the group of evaluation participants is skewed towards the Enterprise Architect and Software Developer roles. Similarly, a majority of participants are employed in the financial services/insurance and the software services industry, i.e., rather digitized industry sectors. Finally, the sample represents experienced practitioners with at least five years of professional experience in IT. Thus, the evaluation is missing the perspective of novices.

Moreover, we chose an anonymous online survey to collect the evaluation data. We used a mix of open and closed questions. A downside of anonymous online surveys using open questions is that responses need interpretation and cannot be clarified in retrospect. Also, we applied a slightly wrong scale for the NPS[3]. Hence, we use the calculated NPS only as a hint towards the increased interest and perceived value of the AMPC for practitioners. Moreover, the responses have been analyzed by one researcher who is also the an author of the AMPC, potentially introducing a bias.

Also, we made minor changes to the AMPC after its evaluation but before its publication as detailed in Section D. One such change was the removal of screenshots in the 'Known Uses' sections of each pattern description due to copyright concerns. Since the survey participants praised the real-world examples and requested more examples, this might reduce the applicability of the published AMPC (Bondel and Matthes, 2023) to real-world settings. However, we included thorough descriptions and links to the respective websites.

Another threat to validity is that the evaluation relies on practitioner opinions and not on actual observations. The survey participants formed these opinions after reviewing the AMPC. Also, we communicated that it is sufficient to review the overview of related patterns (see Fig. 6.5) and one pattern description to participate in the survey. The reasoning behind this statement was to increase the number of responses, since the AMPC is a long document and IT professionals have little time. Participants were informed that they could contact the author of the AMPC to clarify possible questions if necessary.

[3]We used a scale from 1-10 instead of 0-10 as specified in Reichheld (2003).

8.3. Future Work

According to Alexander et al. (1977), *"[...] patterns are very much alive and evolving"* (Alexander et al., 1977, p. xv). Hence, we view the AMPC as a starting point for designing a holistic approach to API management that requires extension, refinement, and regular updates in the future. Therefore, we identify the following areas of future work:

- **Inclusion of Further Materials**:
  - *Extension of the case base:* Future work should comprise collecting and incorporating further knowledge into the AMPC by analyzing public information on public API initiatives and interviewing additional experts. The structured inclusion of more cases enables the identification of new pattern candidates and patterns (Bondel et al., 2021b). Also, new cases would allow for the validation of existing patterns (Bondel et al., 2021b) and, as mentioned during the writers workshop, the addition of more known uses to existing patterns. Such known uses should describe concrete applications of the patterns with as much detail as allowed to be published. The additional cases should capture API initiatives of different kinds of organizations, including Small and Medium Sized Enterprises (SMEs), organizations seated outside of Germany, and digital leaders (Bondel et al., 2021b).
  - *Inclusion of additional literature sources:* Future work should enrich the pattern descriptions with further knowledge from scientific and practice-driven literature. Also, a survey participant requested the enrichment of pattern descriptions with best practices provided by API management software providers (P11). Additional information needs to be incorporated as concise as possible to prevent pattern descriptions from becoming bloated.
- **Improvements of the AMPC's Form:**
  - *New form of the AMPC:* The evaluation of the AMPC in Chapter 7 yielded the result, that the AMPC's navigatability and searchability is limited due to its form as a PDF. Hence, future work should identify and evaluate a more suitable form.
- **Evaluation of the AMPC:**
  - *Further evaluations from a practitioners viewpoint:* Future work should further validate patterns and the pattern catalog structure from a practitioner's perspective. As proposed by the pattern writers workshop participants, evaluations should analyze the completeness and usefulness of the AMPC based on expert feedback.
  - *Implementation of patterns in an organization:* The evaluation of the AMPC presented in Chapter 7 relies on expert opinions. Hence, as previously discussed in Bondel et al. (2021b) and proposed by the peers of the pattern writers workshop, future research endeavors should observe and document the actual implementation of one or several patterns in an organization.
  - *Application of the AMPC to different API management activities:* We identified several activities that the AMPC can support, e.g., the design of new APIs, the improve-

ment of existing APIs, the definition of roles and responsibilities in API initiatives, the training of novices, or vendor and technology analysis (see Section 7.3.2). Future evaluations should comprise action research using the AMPC to guide each activity in real-world settings.

- **Temporal Insights into API Management Practices:**
  - *Evolution approach:* The authors should design and communicate an approach for regular future updates of the AMPC.
  - *Accompanying API initiatives over time:* Repeated interviews with experts on the API initiatives already in the case base would enable observing the evolution of API management practices (Bondel et al., 2021b; Buckl et al., 2013).
  - *Analysis of changes to the pattern catalog over time:* Future work should comprise the continuous evolution of the AMPC as a whole, i.e., identifying new patterns, evolving existing patterns, and removing obsolete patterns. The authors should document these changes to allow for theorizing on the evolution of the API management discipline (Buckl et al., 2013).

Bibliography

Akhan Akbulut and Harry G. Perros. Software Versioning with Microservices through the API Gateway Design Pattern. In *2019 9th International Conference on Advanced Computer Information Technologies (ACIT)*, pages 289–292, 2019. ISBN 978-1-7281-0450-8. doi: 10.1109/ACITT.2019.8779952.

Christopher Alexander. *Notes on the Synthesis of Form.* Harvard University Press, Cambridge, MA, USA, 1 edition, 1973. ISBN 9780674627512.

Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, NY, USA, 1 edition, 1977. ISBN 0195019199.

Jan Algermissen. Classification of HTTP-based APIs. Website, February 2010. URL `http://algermissen.io/classification_of_http_apis.html`. Last accessed on 28th of December 2023.

Axway Inc. Amplify API Management - API lifecycle. Website. URL `https://docs.axway.com/bundle/axway-open-docs/page/docs/api_mgmt_overview/api_mgmt_lifecycle/index.html`. Last accessed on 28th of December 2023.

Rahul C. Basole. Accelerating Digital Transformation: Visual Insights from the API Ecosystem. *IT Professional*, 18(6):20–25, 2016. ISSN 1941-045X. doi: 10.1109/MITP.2016.105.

Rahul C. Basole. On the evolution of service ecosystems: A study of the emerging api economy. In *Handbook of Service Science, Volume II*, pages 479–495, Cham, 2019. Springer International Publishing. ISBN 978-3-319-98512-1. doi: 10.1007/978-3-319-98512-1_21.

Rahul C. Basole, Arjun Srinivasan, Hyunwoo Park, and Shiv Patel. Ecoxight: Discovery, Exploration, and Analysis of Business Ecosystems Using Interactive Visualization. *ACM Transactions on Management Information Systems*, 9(2):1–26, 2018. ISSN 2158-656X. doi: 10.1145/3185047.

Abdelkarim Belkhir, Manel Abdellatif, Rafik Tighilt, Naouel Moha, Yann-Gaël Guéhéneuc, and Éric Beaudry. An Observational Study on the State of REST API Uses in Android Mobile Applications. In *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems*, pages 66–75, 2019. ISBN 978-1-7281-3395-9. doi: 10.1109/MOBILESoft.2019.00020.

Jesus Bellido, Rosa Alarcón, and Cesare Pautasso. Control-Flow Patterns for Decentralized RESTful Service Composition. *ACM Trans. Web*, 8(1), December 2013. ISSN 1559-1131. doi: 10.1145/2535911.

David Bermbach and Erik Wittern. Benchmarking Web API Quality. In *Web Engineering*, pages 188–206, 2016. ISBN 978-3-319-38791-8. doi: 10.1007/978-3-319-38791-8_11.

Gloria Bondel and Florian Matthes. API Management Pattern Catalog for Public, Partner, and Group Web APIs with a Focus on Collaboration. Technical Report TUM-I23101, Software Engineering for Business Information Systems (sebis), Chair for Informatics 19, Technische Universität München, Garching b. München, Germany, December 2023.

Gloria Bondel, Josef Kamysek, Markus Kraft, and Florian Matthes. Design and Implementation of a Test Tool for PSD2 Compliant Interfaces. In *Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021) - Volume 2*, pages 249–256, 2021a. ISBN 978-989-758-509-8. doi: 10.5220/0010439502490256.

Gloria Bondel, Andre Landgraf, and Florian Matthes. API Management Patterns for Public, Partner, and Group Web API Initiatives with a Focus on Collaboration. In *26th European Conference on Pattern Languages of Programs*, EuroPLoP'21, 2021b. ISBN 9781450389976. doi: 10.1145/3489449.3490012.

Gloria Bondel, Arif Cerit, and Florian Matthes. Challenges of API Documentation from a Provider Perspective and Best Practices for Examples in Public Web API Documentation. In *Proceedings of the 24th International Conference on Enterprise Information Systems (ICEIS 2022) - Volume 2*, 2022. ISBN 978-989-758-569-2. doi: 10.5220/0011089700003179.

David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture: W3C Working Group Note 11 February 2004. W3C Recommendation, World Wide Web Consortium (W3C), February 2004. URL `https://www.w3.org/TR/ws-arch/`.

Hayet Brabra, Achraf Mtibaa, Fabio Petrillo, Philippe Merle, Layth Sliman, Naouel Moha, Walid Gaaloul, Yann-Gaël Guéhéneuc, Boualem Benatallah, and Faïez Gargouri. On Semantic Detection of Cloud API (Anti)Patterns. *Information and Software Technology*, 107:65–82, 2019. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.10.012.

Stuart Bretschneider, Frederick J. Marc-Aurele, and Jiannan Wu. "Best Practices" Research: A Methodological Guide for the Perplexed. *Journal of Public Administration Research and Theory: J-PART*, 15(2):307–323, 2005. ISSN 10531858, 14779803. doi: 10.1093/jopart/mui017. URL `http://www.jstor.org/stable/3525702`.

John Brooke. SUS: A 'Quick and Dirty' Usability Scale. In *Usability Evaluation In Industry*, pages 189–194, 1996. ISBN 9780429157011. doi: 10.1201/9781498710411-35.

Sabine Buckl, Alexander M. Ernst, Josef Lankes, and Florian Matthes. Enterprise Architecture Management Pattern Catalog Version 1.0. Technical Report TB 0801, Software Engineering for Business Information Systems (sebis), Chair for Informatics 19, Technische Universität München, Garching b. München, Germany, February 2008.

Sabine Buckl, Florian Matthes, Alexander W. Schneider, and Christian M. Schweda. Pattern-Based Design Research – An Iterative Research Method Balancing Rigor and Relevance. In *Design Science at the Intersection of Physical and Virtual Design*, pages 73–87, 2013. ISBN 978-3-642-38827-9. doi: 10.1007/978-3-642-38827-9_6.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture - A system of patterns*, volume 1. John Wiley & Sons, Chichester, West Sussex, England, 1 edition, 1996. ISBN 0-471-95869-7.

Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern Oriented Software Architecture: On Patterns and Pattern Languages*, volume 5. John Wiley & Sons, Chichester, West Sussex, England, 1 edition, April 2007a. ISBN 0471486485.

Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, volume 4. John Wiley & Sons, Chichester, West Sussex, England, 1 edition, March 2007b. ISBN 0470059028.

Arif Cerit. Improving the developer experience of API consumers using usage scenarios and examples. Master's thesis, Technical University of Munich - Department of Informatics, Garching bei München, Germany, September 2019.

Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Recommendation, World Wide Web Consortium (W3C), June 2007. URL `http://www.w3.org/TR/2007/REC-wsdl20-20070626`.

Ellen Christiaanse, Tonja Van Diepen, and Jan Damsgaard. Proprietary versus internet technologies and the adoption and impact of electronic marketplaces. *The Journal of Strategic Information Systems*, 13(2):151–165, 2004. ISSN 0963-8687. doi: 10.1016/j.jsis.2004.02.004. Strategic Information Systems in the Post-Net Era.

Steven Clarke. Measuring API usability. *Dr. Dobbs Journal*, (29):6–9, 2004.

James O. Coplien. *Software patterns*. SIGS Books & Multimedia, New York, NY, 1996. ISBN 1-884842-50-X.

Ira W. Cotton and Frank S. Greatorex. Data Structures and Techniques for Remote Computer Graphics. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), page 533–544, New York, NY, USA, 1968. ISBN 9781450378994. doi: 10.1145/1476589.1476661.

Robert Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison Wesley, Boston, MA, USA, November 2011. ISBN 032154420X.

Vittorio Dal Bianco, Varvana Myllärniemi, Marko Komssi, and Mikko Raatikainen. The Role of Platform Boundary Resources in Software Ecosystems: A Case Study. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 11–20, 2014. ISBN 978-1-4799-3412-6. doi: 10.1109/WICSA.2014.41.

Brajesh De. *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. Apress, Berkeley, CA, 1 edition, 2017. ISBN 9781484213063.

Mark de Reuver, Carsten Sørensen, and Rahul C Basole. The Digital Platform: A Research Agenda. *Journal of Information Technology*, 33(2):124–135, 2018. doi: 10.1057/s41265-016-0033-3.

Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 221–230, New York, NY, USA, 2004. ISBN 1581138555. doi: 10.1145/1029894.1029925.

Bill Doerrfeld, Bruno Pedro, Kristopher Sandoval, and Andreas Krohn. The API Lifecycle - An Agile Process for Managing the Life of an API. Technical report, Nordic APIs AB, 2015. URL `https://nordicapis.com/wp-content/uploads/theapilifecycle.pdf`. Last accessed on 28th of December 2023.

Ekwa Duala-Ekoko and Martin P Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 266–276, 2012. ISBN 978-1-4673-1067-3. doi: 10.1109/ICSE.2012.6227187.

Lisa Dusseault and James Snell. PATCH Method for HTTP. RFC 5789, Internet Engineering Task Force (IETF), March 2010. URL `https://datatracker.ietf.org/doc/html/rfc5789`.

Paul Dyson and Andy Longshaw. *Architecting Enterprise Solutions - Patterns for High-Capability Internet-Based Systems*. John Wiley & Sons, Chichester, West Sussex, England, 1 edition, 2004. ISBN 9780470856123.

Ben Eaton, Silvia Elaluf-Calderwood, Carsten Sørensen, and Youngjin Yoo. Distributed tuning of boundary resources: The case of apple's ios service system. *MIS Quarterly*, 39(1):217–244, March 2015. ISSN 0276-7783. doi: 10.25300/MISQ/2015/39.1.10.

Thomas Erl. *SOA Design Patterns*. Prentice Hall, Boston, MA, USA, 1 edition, December 2008. ISBN 0136135161.

EU Directive 2015/2366. Directive (EU) 2015/2366 of the European Parliament and of the Council of 25 November 2015 on payment services in the internal market, amending Directives 2002/65/EC, 2009/110/EC and 2013/36/EU and Regulation (EU) No 1093/2010, and repealing Directive 2007/64/EC (Text with EEA relevance). EU Directive 2015/2366, OJ L 337, Official Journal of the European Union, December 2015. URL `https://eur-lex.europa.eu/eli/dir/2015/2366/oj`.

Peter C. Evans and Rahul C. Basole. Revealing the API Ecosystem and Enterprise Strategy via Visual Analytics. *Communications of the ACM*, 59(2):26–28, January 2016. ISSN 0001-0782. doi: 10.1145/2856447.

Roy Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. URL `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

Roy Fielding. REST APIs must be hypertext-driven. Website, October 2008. URL `https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`. Last accessed on 28th of December 2023.

Roy Fielding and Julian Reschke. RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, Internet Engineering Task Force (IETF), June 2014a. URL `https://datatracker.ietf.org/doc/html/rfc7230`.

Roy Fielding and Julian Reschke. RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, Internet Engineering Task Force (IETF), June 2014b. URL `https://datatracker.ietf.org/doc/html/rfc7231`.

Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, Boston, MA, USA, 1 edition, 2003. ISBN 978-0321127426.

Martin Fowler. Richardson Maturity Model - steps toward the glory of REST. Website, March 2010. URL `https://martinfowler.com/articles/richardsonMaturityModel.html`. Last accessed on 28th of December 2023.

Ulrich Frank. Towards a Pluralistic Conception of Research Methods in Information Systems. ICB-Research Report No.7, Institut für Informatik und Wirschaftsinformatik (ICB), Universität Duisburg-Essen, Essen, Germany, December 2006. URL `https://doi.org/10.17185/duepublico/47166`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1995. ISBN 0201633612.

Gartner. Gartner Glossary: Best Practice. Website. URL `https://www.gartner.com/en/information-technology/glossary/best-practice`. Last accessed on 28th of December 2023.

JJ Geewax. *API Design Patterns*. Manning Publications, Shelter Island, NY, USA, 1 edition, July 2021. ISBN 161729585X.

Ahmad Ghazawneh and Ola Henfridsson. Governing third-party development through platform boundary resources. In *Proceedings of the International Conference on Information Systems (ICIS) 2010*, pages 1–18. AIS Electronic Library (AISeL), 2010. URL `https://aisel.aisnet.org/icis2010_submissions/48`.

Ahmad Ghazawneh and Ola Henfridsson. Balancing platform control and external contribution in third-party development: the boundary resources model. *Information Systems Journal*, 23 (2):173–192, 2013. doi: 10.1111/j.1365-2575.2012.00406.x.

Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. Visualizing API Usage Examples at Scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 1–12, 2018. ISBN 9781450356206. doi: 10.1145/3173574.3174154.

Herman Heine Goldstine and John Von Neumann. *Planning and coding of problems for an electronic computing instrument - Reports on the mathematical and logical aspects of an electronic computing instrument, Part 2, Volume 1-3.* Institute for Advanced Study, Princeton, NJ, USA, 1948.

Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C Recommendation, World Wide Web Consortium (W3C), April 2007. URL `https://www.w3.org/TR/soap12-part1/`.

Naga Mallika Gunturu. Enterprise API transformation: Driving towards API economy. *International Journal of Computer Trends and Technology*, 70(6):44–50, July 2022. ISSN 2231-2803. doi: 10.14445/22312803/ijctt-v70i6p105.

John Hagel III and John Seely Brown. Your next it strategy. *Harvard Business Review*, October 2001. URL `https://hbr.org/2001/10/your-next-it-strategy`. Last accessed on 28th of December 2023.

Florian Haupt, Frank Leymann, and Cesare Pautasso. A Conversation Based Approach for Modeling REST APIs. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 165–174, 2015. ISBN 978-1-4799-1922-2. doi: 10.1109/WICSA.2015.20.

Florian Haupt, Frank Leymann, Anton Scherer, and Karolina Vukojevic-Haupt. A Framework for the Structural Analysis of REST APIs. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 55–58, 2017. ISBN 978-1-5090-5729-0. doi: 10.1109/ICSA.2017.40.

Alan R. Hevner. A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems*, 19(2), 2007. URL `https://aisel.aisnet.org/sjis/vol19/iss2/4`.

Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in Information Systems Research. *MIS Quarterly*, pages 75–105, 2004. doi: 10.2307/25148625. URL `https://www.jstor.org/stable/25148625`.

Hillside Europe e. V. EuroPLoP Program. Website, a. URL `https://www.europlop.net/conference/`. Last accessed on 28th of December 2023.

Hillside Europe e. V. EuroPLoP Submission. Website, b. URL `https://www.europlop.net/submission/`. Last accessed on 28th of December 2023.

Hillside Group. The Hillside Group. Website. URL `https://hillside.net/`. Last accessed on 28th of December 2023.

Daniel Hoffman and Paul Strooper. Prose + Test Cases= Specifications. In *Proceedings. 34th International Conference on Technology of Object-Oriented Languages and Systems-TOOLS 34*, pages 239–250. IEEE, 2000. doi: 10.1109/TOOLS.2000.868975.

Daniel Hoffman and Paul Strooper. API documentation with executable examples. *Journal of Systems and Software*, 66(2):143–156, 2003. ISSN 0164-1212. doi: 10.1016/S0164-1212(02)00055-9.

Gregor Hohpe. Enterprise integration patterns. Website. URL `https://www.enterpriseintegrationpatterns.com/index.html`. Last accessed on 28th of December 2023.

Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, Boston, MA, USA, 1 edition, October 2003. ISBN 978-0321200686.

Masaki Hosono, Hironori Washizaki, Kiyoshi Honda, Hiromasa Nagumo, Hisanobu Sonoda, Yoshiaki Fukazawa, Kazuki Munakata, Takao Nakagawa, Yusuke Nemoto, Susumu Tokumoto, et al. Inappropriate usage examples in web api documentations. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 343–347, 2019. doi: 10.1109/ICSME.2019.00052.

Jukka Huhtamäki, Rahul C. Basole, Kaisa Still, Martha G. Russell, and Marko Seppänen. Visualizing the Geography of Platform Boundary Resources: The Case of the Global API Ecosystem. In *Proceedings of the 50th Hawaii International Conference on System Sciences (HICSS)*, January 2017. ISBN 978-0-9981331-0-2. doi: 10.24251/HICSS.2017.642.

Sergio Inzunza, Reyes Juárez-Ramírez, and Samantha Jiménez. Api documentation - a conceptual evaluation model. In *Trends and Advances in Information Systems and Technologies*, pages 229–239, 2018. ISBN 978-3-319-77712-2. doi: 10.1007/978-3-319-77712-2_22.

Anna Sigridur Islind, Tomas Lindroth, Ulrika Lundh Snis, and Carsten Sørensen. Co-creation and Fine-Tuning of Boundary Resources in Small-Scale Platformization. In *Scandinavian conference on information systems (SCIS 2016): Nordic Contributions in IS Research*, pages 149–162, 2016. ISBN 978-3-319-43597-8. doi: 10.1007/978-3-319-43597-8_11.

ISO 20077-1. ISO 20077-1:2017 - Road Vehicles - Extended vehicle (ExVe) methodology - Part 1: General information. ISO 20077-1:2017, International Organization for Standardization, Geneva, CH, December 2017.

ISO 20078-1. ISO 20078-1:2021 - Road vehicles - Extended vehicle (ExVe) web services - Part 1: Content and definitions. ISO 20078-1:2021, International Organization for Standardization, Geneva, CH, November 2021.

ISO 20080. ISO 20080:2019 - Road vehicles - Information for remote diagnostic support - General requirements, definitions and use cases. ISO 20080:2019, International Organization for Standardization, Geneva, CH, March 2019.

ISO/IEC/IEEE 29119-1:2013. ISO/IEC/IEEE 29119-1:2013 –Software and systems engineering – Software testing – Part 1: Concepts and definitions – First edition 2013-09-01. ISO/IEC/IEEE 29119-1:2013(E), International Organization for Standardization, Geneva, CH, September 2013.

ISO/IEC/IEEE 29119-1:2022. ISO/IEC/IEEE 29119-1:2022 –Software and systems engineering – Software testing – Part 1: General concepts. ISO/IEC/IEEE 29119-1:2022(E), International Organization for Standardization, Geneva, CH, January 2013.

Bala Iyer and Mohan Subramaniam. Corporate alliances matter less thanks to apis. *Harvard Business Review*, June 2015. URL `https://hbr.org/2015/06/corporate-alliances-matter-less-thanks-to-apis`. Last accessed on 28th of December 2023.

Daniel Jacobson, Greg Brail, and Dan Woods. *APIs: A Strategy Guide*. O'Reilly, Sebastopol, CA, USA, 2012. ISBN 978-1-449-30892-6.

Sae Young Jeong, Yingyu Xie, Jack Beaton, Brad A. Myers, Jeff Stylos, Ralf Ehret, Jan Karstens, Arkin Efeoglu, and Daniela K. Busse. Improving Documentation for eSOA APIs through User Studies. In *End-User Development*, volume 5435, pages 86–105, 2009. ISBN 978-3-642-00427-8. doi: 10.1007/978-3-642-00427-8_6.

Juanjuan Jiang, Johannes Koskinen, Anna Ruokonen, and Tarja Systa. Constructing usage scenarios for API redocumentation. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 259–264, 2007. doi: 10.1109/ICPC.2007.16.

Tom Johnson. Documenting APIs: A guide for technical writers and engineers. Website, April 2023. URL `https://idratherbewriting.com/learnapidoc/`. Last accessed on 28th of December 2023.

Kimmo Karhu, Robin Gustafsson, and Kalle Lyytinen. Exploiting and defending open digital platforms with boundary resources: Android's five platform forks. *Information Systems Research*, 29(2):479–497, 2018. doi: 10.1287/isre.2018.0786.

Tom Kendrick. *Identifying and Managing Project Risk: Essential Tools for Failure-Proofing Your Project*. AMACOM, New York, NY, USA, 3 edition, March 2015. ISBN 978-0-8144-3608-0.

Pouya Aleatrati Khosroshahi, Matheus Hauder, Alexander W. Schneider, and Florian Matthes. Enterprise architecture management pattern catalog version 2.0. Technical report, Software Engineering for Business Information Systems (sebis), Chair for Informatics 19, Technische Universität München, Garching b. München, Germany, November 2015.

Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management*, volume 3. John Wiley & Sons, Chichester, West Sussex, England, 1 edition, April 2004. ISBN 0470845252.

Andrew J. Ko and Yann Riche. The role of conceptual knowledge in API usability. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 173–176, 2011. doi: 10.1109/VLHCC.2011.6070395.

Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, pages 199–206, 2004. doi: 10.1109/VLHCC.2004.47.

Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 344–353, 2007. ISBN 0769528287. doi: 10.1109/ICSE.2007.45.

Jacek Kopecký, Paul Fremantle, and Rich Boakes. A history and future of Web APIs. *it - Information Technology*, 56(3):90–97, 2014. doi: 10.1515/itit-2013-1035.

Sebastian Kotstein and Justus Bogner. Which RESTful API design rules are important and how do they improve software quality? A delphi study with industry experts. In *Service-Oriented Computing*, pages 154–173, 2021. ISBN 978-3-030-87568-8. doi: 10.1007/978-3-030-87568-8_10.

Andre Landgraf. Identification of API management patterns from an api provider perspective. Master's thesis, Technical University of Munich - Department of Informatics, Garching b. München, Germany, February 2021.

Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE software*, 20(6):35–39, 2003. ISSN 1937-4194. doi: 10.1109/MS.2003.1241364.

Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. Interface evolution patterns: Balancing compatibility and extensibility across service life cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, EuroPLop '19, 2019. ISBN 9781450362061. doi: 10.1145/3361149.3361164.

Kalle Lyytinen and Jan Damsgaard. Inter-organizational information systems adoption – A configuration analysis approach. *European Journal of Information Systems*, 20(5):496–509, 2011. doi: 10.1057/ejis.2010.71.

Walid Maalej and Martin P. Robillard. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013. ISSN 1939-3520. doi: 10.1109/TSE.2013.12.

Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating Web APIs on the world wide web. In *2010 Eighth IEEE European Conference on Web Services*, pages 107–114, 2010. ISBN 978-1-4244-9397-5. doi: 10.1109/ECOWS.2010.9.

Nicolas Masse. Full API lifecycle management: A primer. Website, February 2019. URL `https://developers.redhat.com/blog/2019/02/25/full-api-lifecycle-management-a-primer/`. Last accessed on 28th of December 2023.

Max Mathijssen, Michiel Overeem, and Slinger Jansen. Identification of practices and capabilities in API management: A systematic literature review. 2020. doi: 10.48550/arXiv.2006.10481. URL `https://arxiv.org/abs/2006.10481`.

Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay I. Spinuzzi. Building more usable APIs. *IEEE software*, 15(3):78–86, 1998. ISSN 1937-4194. doi: 10.1109/52.676963.

Mehdi Medjaoui, Erik Wilde, Ronnie Mitra, and Mike Amundsen. *Continuous API Management - Making the right decisions in an evolving landscape*. O'Reilly, 1 edition, 2018. ISBN 978-1-492-04355-3.

Michael Meng, Stephanie Steinhardt, and Andreas Schubert. Application programming interface documentation: What do software developers want? *Journal of Technical Writing and Communication*, 48(3):295–330, 2018. doi: 10.1177/0047281617721853.

Michael Meng, Stephanie Steinhardt, and Andreas Schubert. How developers use API documentation: An observation study. *Communication Design Quarterly Review*, 7(2):40–49, 2019. doi: 10.1145/3358931.3358937.

Michael Meng, Stephanie M. Steinhardt, and Andreas Schubert. Optimizing api documentation: Some guidelines and effects. In *Proceedings of the 38th ACM International Conference on Design of Communication*, SIGDOC '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375252. doi: 10.1145/3380851.3416759.

Gerard Meszaros and Jim Doble. A pattern language for pattern writing. In *Pattern Languages of Program Design 3*, volume 3, pages 529–574, 1997. ISBN 0201310112.

Fabrizio Montesi and Janine Weber. Circuit Breakers, Discovery, and API Gateways in Microservices. 2016. doi: 10.48550/arXiv.1609.05830. URL https://arxiv.org/abs/1609.05830.

MuleSoft. 2023 Connectivity Benchmark Report. Technical report, MuleSoft Research, CA, USA, 2023. URL https://www.mulesoft.com/lp/reports/connectivity-benchmark. Last accessed on 28th of December 2023.

Daniel Müssig, Robert Stricker, Jörg Lässig, and Jens Heider. Highly scalable microservice-based enterprise architecture for smart ecosystems in hybrid cloud environments. In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017)*, volume 3, pages 454–459, 2017. ISBN 978-989-758-249-3. doi: 10.5220/0006373304540459.

Brad A. Myers and Jeffrey Stylos. Improving API usability. *Communications of the ACM*, 59 (6):62–69, 2016. ISSN 0001-0782. doi: 10.1145/2896587.

Seyed Mehdi Nasehi and Frank Maurer. Unit tests as API usage examples. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010. doi: 10.1109/ICSM.2010.5609553.

Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34, 2012. doi: 10.1109/ICSM.2012.6405249.

Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An analysis of public REST Web service APIs. *IEEE Transactions on Services Computing*, 14(4):957–970, 2021. ISSN 1939-1374. doi: 10.1109/TSC.2018.2847344.

Sam Newman. Sam newman & associates - patterns. Website. URL https://samnewman.io/patterns/. Last accessed on 28th of December 2023.

Sam Newman. *Monolith To Microservices - Evolutionary Patterns to Transform your Monolith.* O'Reilly Media, Sebastopol, CA, USA, 1 edition, December 2019. ISBN 1492047848.

NIST. Glossary: Best practice. Website. URL https://csrc.nist.gov/glossary/term/best_practice. Last accessed on 28th of December 2023.

Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. What Programmers Really Want: Results of a Needs Assessment for SDK Documentation. In *Proceedings of the 20th Annual International Conference on Computer Documentation*, SIGDOC '02, page 133–141, 2002. ISBN 1581135432. doi: 10.1145/584955.584976.

Joshua Ofoeda, Richard Boateng, and John Effah. Application programming interface (API) research: A review of the past to inform the future. *International Journal of Enterprise Information Systems (IJEIS)*, 15(3):76–95, 2019. ISSN 1548-1115. doi: 10.4018/IJEIS.2019070105.

Francis Palma, Johann Dubois, Naouel Moha, and Yann-Gaël Guéhéneuc. Detection of REST patterns and antipatterns: A heuristics-based approach. In *Service-Oriented Computing*, pages 230–244, 2014. ISBN 978-3-662-45391-9. doi: 10.1007/978-3-662-45391-9_16.

Francis Palma, Javier Gonzalez-Huerta, Naouel Moha, Yann-Gaël Guéhéneuc, and Guy Tremblay. Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti)Patterns. In *Service-Oriented Computing*, pages 171–187, 2015. ISBN 978-3-662-48616-0. doi: 10.1007/978-3-662-48616-0_11.

David L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972. ISSN 0001-0782. doi: 10.1145/361598.361623.

Cesare Pautasso and Erik Wilde. RESTful web services: Principles, patterns, emerging technologies. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1359–1360, 2010. ISBN 978-1-4614-7518-7. doi: 10.1145/1772690.1772929.

Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. A Pattern Language for RESTful Conversations. In *Proceedings of the 21st European Conference on Pattern Languages of Programs*, EuroPlop '16, 2016. ISBN 9781450340748. doi: 10.1145/3011784.3011788.

Fabio Petrillo, Philippe Merle, Naouel Moha, and Yann-Gaël Guéhéneuc. Are REST APIs for cloud computing well-designed? An exploratory study. In *Service-Oriented Computing*, pages 157–170, 2016. ISBN 978-3-319-46295-0. doi: 10.1007/978-3-319-46295-0_10.

Shameen Pillai, Kimihiko Iijima, Mark O'Neill, John Santoro, Akash Jain, and Fintan Ryan. Magic quadrant for full life cycle API management. Technical Report ID G00735998, Gartner, CT, USA, September 2021.

Frederick F. Reichheld. The one number you need to grow. *Harvard Business Review*, December 2003. URL https://hbr.org/2003/12/the-one-number-you-need-to-grow. Last accessed on 28th of December 2023.

Dominik Renzel, Patrick Schlebusch, and Ralf Klamma. Today's top "RESTful" services and why they are not RESTful. In *Web Information Systems Engineering - WISE 2012*, pages 354–367, 2012. ISBN 978-3-642-35063-4. doi: 10.1007/978-3-642-35063-4_26.

Eric Rescorla. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Network Working Group, August 2018. URL https://datatracker.ietf.org/doc/html/rfc8446.

Chris Richardson. Microservices architecture - A pattern language for microservices. Website. URL https://microservices.io/patterns/index.html. Last accessed on 28th of December 2023.

Chris Richardson. *Microservices patterns*. Mannig Publications Co., Shelter Island, NY, USA, 1 edition, 2019. ISBN 9781617294549.

Leonard Richardson. Justice will take us millions of intricate moves, act three: The maturity heuristic. Website, January 2009. URL https://www.crummy.com/writing/speaking/2008-QCon/act3.html. Last accessed on 28th of December 2023.

Martin P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34, 2009. ISSN 1937-4194. doi: 10.1109/MS.2009.193.

Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011. doi: 10.1007/s10664-010-9150-8.

Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. REST APIs: A large-scale analysis of compliance with principles and best practices. In *Web Engineering*, pages 21–39, 2016. ISBN 978-3-319-38791-8. doi: 10.1007/978-3-319-38791-8_2.

Arnon Rotem-Gal-Oz. *SOA Patterns*. Manning Publications, Shelter Island, NY, USA, 1 edition, April 2012. ISBN 978-1933988269.

Ivan Salvadori and Frank Siqueira. A maturity model for semantic RESTful Web APIs. In *2015 IEEE International Conference on Web Services*, pages 703–710, 2015. ISBN 978-1-4673-7272-5. doi: 10.1109/ICWS.2015.98.

Mattia Santoro, Lorenzino Vaccari, Dimitrios Mavridis, Robin Smith, Monica Posada, and Dietmar Gattwinkel. Web Application Programming Interfaces (APIs): General purpose standards, terms and European Commission initiatives. EUR 29984, Publications Office of the European Union, Luxembourg, 2019. JRC118082.

SAP America. Compass Readme. Website, 2023. URL https://github.com/kyma-incubator/compass. Last accessed on 28th of December 2023.

Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, Chichester, West Sussex, England, 1 edition, August 2000. ISBN 0471606952.

Paul Schmiedmayer. *Designing Evolvable Web Services*. PhD thesis, Department of Computer Science, Technical University Munich, Munich, Germany, February 2022.

Mitchell Shnier. *Dictionary of PC Hardware and Data Communications Terms*. O'Reilly and Associates, Sebastopol, CA, USA, 1 edition, April 1996. ISBN 1-56592-158-5.

S. M. Sohan, Frank Maurer, Craig Anslow, and Martin P. Robillard. A study of the effectiveness of usage examples in REST API documentation. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 53–61. IEEE, IEEE, 2017. doi: 10.1109/VLHCC.2017.8103450.

Kai Spichale. *API-Design - Praxishandbuch für Java- und Webservice-Entwickler*. dpunkt.verlag, Heidelberg, Germany, 1 edition, 2017. ISBN 978-3-86490-387-8.

Mirko Stocker, Olaf Zimmermann, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. Interface Quality Patterns: Communicating and Improving the Quality of Microservices APIs. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, EuroPLoP '18, 2018. ISBN 9781450363877. doi: 10.1145/3282308.3282319.

Jeffrey Stylos and Steven Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 529–539, 2007. ISBN 0769528287. doi: 10.1109/ICSE.2007.92.

Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A. Myers. Improving API documentation using API usage information. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 119–126, 2009. doi: 10.1109/VLHCC.2009.5295283.

Wei Tan, Yushun Fan, Ahmed Ghoneim, M. Anwar Hossain, and Schahram Dustdar. From the service-oriented architecture to the Web API economy. *IEEE Internet Computing*, 20(4): 64–68, 2016. ISSN 1941-0131. doi: 10.1109/MIC.2016.74.

Kyle Thayer, Sarah E. Chasins, and Amy J. Ko. A theory of robust API knowledge. *ACM Transactions on Computing Education (TOCE)*, 21(1):1–32, 2021. doi: 10.1145/3444945.

The GraphQL Foundation. GraphQL best practices. Website, a. URL `https://graphql.org/learn/best-practices/`. Last accessed on 28th of December 2023.

The GraphQL Foundation. GraphQL - october 2021 edition. Website, b. URL `https://spec.graphql.org/October2021/`. Last accessed on 28th of December 2023.

The GraphQL Foundation. Introduction to GraphQL. Website, c. URL `https://graphql.org/learn/`. Last accessed on 28th of December 2023.

The GraphQL Foundation. GraphQL - schemas and types. Website, d. URL `https://graphql.org/learn/schema/`. Last accessed on 28th of December 2023.

The GraphQL Foundation. GraphQL - serving over HTTP. Website, e. URL `https://graphql.org/learn/serving-over-http/`. Last accessed on 28th of December 2023.

Martin Treiber, Hong-Linh Truong, and Schahram Dustdar. On analyzing evolutionary changes of Web services. In *Service-Oriented Computing – ICSOC 2008 Workshops*, pages 284–297, 2009. ISBN 978-3-642-01247-1. doi: 10.1007/978-3-642-01247-1_29.

Gias Uddin and Martin P. Robillard. How API documentation fails. *IEEE software*, 32(4): 68–75, 2015. ISSN 1937-4194. doi: 10.1109/MS.2014.80.

Ömer Uludağ, Nina-Mareike Harders, and Florian Matthes. Documenting recurring concerns and patterns in large-scale agile development. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, EuroPLop '19, 2019. ISBN 9781450362061. doi: 10.1145/3361149.3361176.

Steve Vinoski. Serendipitous reuse. *IEEE Internet Computing*, 12(1):84–87, 2008. ISSN 1941-0131. doi: 10.1109/MIC.2008.20.

Markus Völter, Michael Kircher, and Uwe Zdun. *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware.* John Wiley & Sons, Hoboken, NJ, USA, 1 edition, December 2004. ISBN 978-0470856628.

Robert B. Watson. Development and Application of a Heuristic to Assess Trends in API Documentation. In *Proceedings of the 30th ACM International Conference on Design of Communication*, SIGDOC '12, pages 295–302, 2012. ISBN 9781450314978. doi: 10.1145/2379057.2379112.

Robert B. Watson, Mark Stamnes, Jacob Jeannot-Schroeder, and Jan H. Spyridakis. API Documentation and Software Community Values: A Survey of Open-Source API Documentation. In *Proceedings of the 31st ACM International Conference on Design of Communication*, SIGDOC '13, pages 165–174, 2013. ISBN 9781450321310. doi: 10.1145/2507065.2507076.

Jane Webster and Richard T. Watson. Analyzing the past to prepare for the future: Writing a literature review. *MIS Quarterly*, 26(2):xiii–xxiii, 2002. ISSN 02767783. URL http://www.jstor.org/stable/4132319.

Michael Weiss and G. R. Gangadharan. Modeling the mashup ecosystem: Structure and growth. *R&D Management*, 40(1):40–49, 2010. doi: 10.1111/j.1467-9310.2009.00582.x.

D. J. Wheeler. The use of sub-routines in programmes. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*, ACM '52, pages 235–236, 1952. doi: 10.1145/609784.609816.

Manuel Wiesche, Marlen C. Jurisch, Philip W. Yetton, and Helmut Krcmar. Grounded Theory Methodology in Information Systems Research. *MIS Quarterly*, 41(3):685–701, 2017. ISSN 0276-7783. doi: 10.25300/MISQ/2017/41.3.02.

Erik Wittern, Annie T.T. Ying, Yunhui Zheng, Jim A. Laredo, Julian Dolby, Christopher C. Young, and Aleksander A. Slominski. Opportunities in software engineering research for Web API consumption. In *2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI)*, pages 7–10, 2017. ISBN 978-1-5386-2805-8. doi: 10.1109/WAPI.2017.1.

Robert K. Yin. *Case study research: Design and methods*, volume 5. Sage Publications, Los Angeles, CA, 2013. ISBN 1452242569.

Youngjin Yoo, Ola Henfridsson, and Kalle Lyytinen. Research commentary: The new organizing logic of digital innovation: An agenda for information systems research. *Information Systems Research*, 21(4):724–735, 2010. ISSN 10477047, 15265536. URL http://www.jstor.org/stable/23015640.

U. Zdun, M. Kircher, and M. Volter. Remoting patterns: Design reuse of distributed object middleware solutions. *IEEE Internet Computing*, 8(6):60–68, 2004. doi: 10.1109/MIC.2004.70.

Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding Architectural Decision Making on Quality Aspects in Microservice APIs. In *Service-Oriented Computing*, pages 73–89, 2018. ISBN 978-3-030-03596-9. doi: 10.1007/978-3-030-03596-9_5.

Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. Patterns for API design. Website. URL https://microservice-api-patterns.org/. Last accessed on 28th of December 2023.

Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, 2017. ISBN 9781450348485. doi: 10.1145/3147704.3147734.

Olaf Zimmermann, Daniel Lübke, Uwe Zdun, Cesare Pautasso, and Mirko Stocker. Interface Responsibility Patterns: Processing Resources and Operation Responsibilities. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, EuroPLoP '20, 2020a. ISBN 9781450377690. doi: 10.1145/3424771.3424822.

Olaf Zimmermann, Cesare Pautasso, Daniel Lübke, Uwe Zdun, and Mirko Stocker. Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, EuroPLoP '20, 2020b. ISBN 9781450377690. doi: 10.1145/3424771.3424821.

Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. Introduction to microservice API patterns (MAP). In *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*, volume 78 of *OpenAccess Series in Informatics (OASIcs)*, pages 4:1–4:17, Dagstuhl, Germany, 2020c. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-137-5. doi: 10.4230/OASIcs. Microservices.2017-2019.4. URL `https://drops.dagstuhl.de/opus/volltexte/2020/11826`.

Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley Professional, 1 edition, December 2022. ISBN 0137670109.

Jonathan L. Zittrain. The generative internet. *Harvard Law Review*, 119(7):1974–2040, 2006. ISSN 0017811X. URL `http://www.jstor.org/stable/4093608`.

Abbreviations

AMPC	API Management Pattern Catalog
API	Application Programming Interface
CD	Continuous Deployment
CI	Continuous Integration
CORBA	Common Object Request Broker Architecture
CRM	Customer Relationship Management
CRUD	Create, Read, Update, and Delete
DCOM	Distributed Component Object Mode
DDoS	Distributed Denial of Service
EDI	Electronic Data Exchange
GTM	Grounded Theory Methodology

HATEOAS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
ID	Identifier
IoT	Internet of Things
IP	Internet Protocol
IS	Information Systems
ISR	Information Systems Research
IT	Information Technology
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
NIST	National Institute of Standards and Technology
NPS	Net Promoter Score
MAP	Microservice API Patterns
OAS	OpenAPI Specification
OEM	Original Equipment Manufacturer

PDF	Portable Document Format
POSA	Pattern-oriented Software Architecture
QoS	Quality of Service
REST	REpresentational State Transfer
RPC	Remote Procedure Call
RPI	Remote Procedure Invocation
RQ	Research Question
PSD2	Revised Payment Services Directive
SDK	Software Development Kit
sebis	Software Engineering for Business Information Systems
SME	Small and Medium Sized Enterprise
SMS	Short Message Service
SOA	Service-oriented Architecture
SUS	System Usability Scale
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Unified Resource Identifier
URL	Unified Resource Locator

US United States of America

W3C World Wide Web Consortium

WSDL Web Service Description Language

WWW World Wide Web

XaaS Everything-as-a-Service

XML Extensible Markup Language

XSD XML Schema Definition

APPENDIX A

Prior Publications and Student Thesis in the Context of this Dissertation

The dissertation at hand builds on prior publications of the author. In addition to referencing them in the bibliography and relevant sections, we list them in the following.

- **Bondel et al. (2021a):** Gloria Bondel, Josef Kamysek, Markus Kraft, and Florian Matthes. Design and Implementation of a Test Tool for PSD2 Compliant Interfaces. In *Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021)* - Volume 2, pages 249–256, 2021a. ISBN 978-989-758-509-8. doi: 10.5220/0010439502490256.
- **Bondel et al. (2021b):** Gloria Bondel, Andre Landgraf, and Florian Matthes. API Management Patterns for Public, Partner, and Group Web API Initiatives with a Focus on Collaboration. In *26th European Conference on Pattern Languages of Programs*, EuroPLoP'21, 2021b. ISBN 9781450389976. doi: 10.1145/3489449.3490012.
- **Bondel et al. (2022):** Gloria Bondel, Arif Cerit, and Florian Matthes. Challenges of API Documentation from a Provider Perspective and Best Practices for Examples in Public Web API Documentation. In *Proceedings of the 24th International Conference on Enterprise Information Systems (ICEIS 2022) - Volume 2*, 2022. ISBN 978-989-758-569-2. doi: 10.5220/0011089700003179.
- **Bondel and Matthes (2023):** Gloria Bondel and Florian Matthes. API Management Pattern Catalog for Public, Partner, and Group Web APIs with a Focus on Collaboration. Technical Report TUM-I23101, Software Engineering for Business Information Systems (sebis), Chair for Informatics 19, Technische Universität München, Garching b. München, Germany, December 2023.

Moreover, the author advised several student theses in the context of this dissertation. In addition to referencing them in the bibliography and relevant sections, we list relevant student theses below.

- **Cerit (2019):** Arif Cerit. Improving the developer experience of API consumers using usage scenarios and examples. Master's thesis, Technical University of Munich - Department of Informatics, Garching bei München, Germany, September 2019.
- **Landgraf (2021):** Andre Landgraf. Identification of API management patterns from an api provider perspective. Master's thesis, Technical University of Munich - Department of Informatics, Garching b. München, Germany, February 2021.

APPENDIX B

Evolution of Patterns in the AMPC

We present an overview of changes between the patterns shown in Landgraf (2021) and the AMPC (Bondel and Matthes, 2023) in Tab. B.1. If not stated otherwise in the description of the significant changes, the pattern essence is the same in both catalogs. However, the pattern structure of the patterns in Landgraf (2021) and the AMPC differ, and we enriched the AMPC's pattern descriptions with additional information. Thus, the pattern descriptions document different information.

Pattern name in Landgraf (2021)	**Pattern Name in the AMPC (Bondel and Matthes, 2023)**	**Result of the Change**	**Description of Major Changes**
P1: Internal API registry		Removed	As the name "Internal API Registry" already states, this is a pattern exclusively observed in private API initiatives. Therefore, we removed it from the pattern catalog.
P2: Company-wide ticketing system	API provider-wide ticketing management	Adopted	
P3: API testing strategy	Test Strategy	Adopted	
P4: Pilot Project	Collaborative Pilot Project	Adopted	In the AMPC, the pattern emphasizes the collaboration between API providers and consumers more prominently. Also, we exclude C13 from the 'Known Uses'.
P5: Frontend Venture	Frontend Venture	Adopted	
P6: SLAs with backend providers	SLA	Adopted	In Landgraf (2021), it is difficult to understand the difference between P6 and P7. Therefore, in the AMPC, we merged the two patterns into one pattern.
P7: SLAs with API consumers		Removed	See explanation above (P6: SLAs with backend providers).
P8: Data Clearance	Data Clearing Process	Adopted	In Landgraf (2021), this pattern focuses a lot on reusing internal backends externally. In the AMPC, we broadened the focus to comprise data clearing for all new APIs and major changes to existing APIs.
P9: API orchestration layer	API Facade	Adopted	
P10: Tailoring APIs to products	API-as-a-Product	Adopted	We add C6 to the 'Known Uses'.
P11: API product validation		Removed	Landgraf (2021) describes the solution as collecting any kind of consumer feedback along the API lifecycle. Since this description is very generic, we removed the pattern in the AMPC.
P12: Idea Backlog	Idea Backlog	Adopted	We added Twilio (C17) to the 'Known Uses'.
P13: API product documentation	Consumer-centric API Description	Adopted	We remove the C13 and C14 from the 'Known Uses'.
P14: Cookbooks	Integration Guide	Adopted	
P15: Software libraries	Client Libraries	Adopted	
P16: Integration partner management	Integration Partner Program	Adopted	We added Stripe (C16) to the 'Known Uses'.
P17: Role-based marketing	Role-based marketing	Adopted	Landgraf (2021) distinguished between business and technical stakeholders. In comparison, in the AMPC, we presented two variants, i.e., (1) business and technical stakeholders and (2) platform users and third-party developers. Also, we added C11 to the 'Known Uses'.
P18: Newsletter	Newsletter	Adopted	We added C13, Stripe (C16), and Twilio (C17) to the 'Known Uses'.
P19: Customer success stories	Customer Success Stories	Adopted	We add C11, Stripe (C16), and Twilio (C17) to the 'Known Uses'.
P20: First-level support	Dedicated Support Team	Adopted	
P21: Service desk software		Removed	The pattern was very similar to the pattern `P2: Company-wide ticketing system` making them difficult to distinguish. Also, the pattern describes a type of software. Hence, we removed the pattern from the AMPC.
P22: Self-service	Onboarding Self-service	Adopted	We add C2, Stripe (C16), and Twilio (C17) to the 'Known Uses'.
P23: Multi-tenant management		Removed	While the interviewees of cases C6, C9, and C13 use terms like *multi-tenancy* [C6] and *tenant isolation* [C9], the interview partners address different concepts. Hence, we cannot confirm the pattern `P23: Multi-tenant management` but instead, describe the pattern candidates `Several Developer Portal Instances`, `White-label Marketplace`, and `Tenant Isolation`.
	API Product Owner	New	We identified this pattern during the repeated data analysis.
	API Quality Monitoring	New	We identified this pattern during the repeated data analysis.
	Play-it-fast approach	New	We identified this pattern during the repeated data analysis.
	Web API	New	We identified this pattern during the repeated data analysis.

Table B.1.: Overview of the major changes of patterns between Landgraf (2021) and the AMPC (Bondel and Matthes, 2023).

APPENDIX C

Questions for the EuroPLoP Writer's Workshop

This appendix presents the questions transmitted to the participants of the Writer's Workshop of the EuroPLoP[1] conference prior to the workshop.

Questionnaire

Dear Reviewer of the EuroPLoP paper,

thank you very much for reviewing the article „API Management Patterns for Public, Partner, and Group API Initiatives with a Focus on Collaboration “. The paper summarizes the research approach and presents two exemplary patterns of a more extensive pattern language. We are currently still finalizing the pattern language, which will be a significant artifact of my doctoral Thesis.

The goal of participating in the EuroPLoP is to validate the research approach and the pattern structure scientifically. We will also transfer your feedback on the pattern structure to the other patterns of the pattern language. Therefore, we have compiled a set of questions that came up during the design of the pattern language and the discussions with our shepherd, hoping that these questions can guide you during the review process and the workshop discussions:

Research approach:

- We have chosen a design science approach in combination with the "rule of three" introduced by Coplien (1996). Do you have any suggestions for improving the research approach or the research approach description?
- The next step for creating the overall pattern language will be to finalize the current version

[1] `https://www.europlop.net/conference/`

based on you feedback and afterward evaluate it. The evaluation will consist of showing the pattern language to industry experts and conducting semi-structured interviews. What do you think of this approach? Do you have any suggestions on how to maximize the outcome of this approach?

Pattern language:

- We summarized all patterns of the pattern language in the problem/solution summary and visually related the patterns in Fig. 3 and Fig. 5. Are the visual representations easy to read and do they add value for the reader?
- Do you have any suggestions for improving the problem/solution summary or the visual representations?

Pattern

- Does the current structure of the pattern descriptions contain all necessary information?
- Does the current structure of the pattern descriptions allow for an intuitive reading flow?
- What do you think about the abstraction level in the section *Implementation Details*? Does the section contain too much or too little information?
- Does the section *Related Patterns* contain too much or too little information?
- We added some "meta" findings related to the cases in which we observed the patterns in the section *Known Uses*. Do you think this is the right section to include this kind of information, or should it be included in a different section (e.g., in the section *Forces*)?

General findings:

- Do you think the general findings are valuable?

Thank you very much, and I am looking forward to your feedback during the EuroPLoP workshop.

APPENDIX D

Changes to the AMPC before its Publication

We published the AMPC online, free of charge in Bondel and Matthes (2023). After its evaluation but before its publication, we made minor changes to the AMPC:

- We identified further related patterns in other pattern collections. Hence, we added relations between the AMPC's patterns and patterns presented in these pattern collections in the respective 'Other Related Patterns' sections and the visualization of these relations. More precisely, we added relations to patterns presented in Geewax (2021), Bellido et al. (2013), Erl (2008), Rotem-Gal-Oz (2012), Völter et al. (2004), and Dyson and Longshaw (2004).
- Initially, the goal of the AMPC comprised providing API management patterns also aimed at SMEs. However, the case base did not hold any API initiative descriptions of SMEs. Also, the practitioners who participated in the evaluation did not belong to SMEs. Hence, we cannot make any statement about the suitability of the patterns for SMEs. Therefore, we removed the claim of supporting SMEs from the AMPC.
- We removed screenshots of examples in the pattern descriptions due to copyright concerns. However, we included thorough descriptions and links to the respective websites.
- We aligned the 'Summary' section with the initial publication of the observations in Bondel et al. (2021b).
- We corrected spelling, punctuation, and grammatical errors.
- We improved the quality and design of visual illustrations.

APPENDIX E

AMPC Pattern Summaries

In the following, we summarize each pattern presented in the AMPC. Tab. E.1 summarizes the *Interface Type Patterns*, Tab. E.2 summarizes *API Provider Internal Patterns*, and Tab. E.3 summarizes the *API Consumer faceing Patterns*. The tables have been previously presented in Bondel et al. (2021b).

Table E.1.: Summaries of the Interface Type Patterns adopted from Bondel and Matthes (2023).

Pattern Name	**Pattern Solution Description**
`Web API`	A provider uses a Web Application Programming Interface (Web API) that exposes functionality or data via the public internet, i.e., uses HTTP(S) as communication protocol (Bermbach and Wittern, 2016; De, 2017; Santoro et al., 2019). The Web API decouples the functionalities implementation from the interface that makes it accessible (Spichale, 2017; De, 2017; Medjaoui et al., 2018). Also, the Web API defines the contract for interactions between the backend and the client application (De, 2017; Jacobson et al., 2012).
`Client Library`	A client library wraps a Web API and enables consumers to access it using code in a specific programming language and compliant with a certain framework (De (2017); C3). Hence, the application consumer does not have to interact with the Web API directly but indirectly through the library functions in the programming language of choice.
`Frontend Venture`	The API provider enables consumers who cannot, for any reason, integrate an API or client library, to still use the functionality or data via a simple frontend, i.e., a website with fields and buttons that trigger API functionality. If enough consumers are interested in the frontend, a product team can take over the development and maintenance of the frontend.

Table E.2.: Summaries of the API Provider Internal Patterns adopted from Bondel and Matthes (2023).

Pattern Name	**Pattern Solution Description**
`API-as-a-Product`	The API provider treats APIs like any other consumer-facing (software) product, including technical, business, legal, marketing, and other aspects.
`API Product Owner`	An API product owner is responsible for an API's economic success, designs and evolves the API according to consumers' needs, and represents the API internally.
`Collaborative Pilot Project`	The API provider designs a new API iteratively in close collaboration with one or a limited set of API consumers to increase the likelihood of the API meeting API consumers' needs.
`Play-it-fast Approach`	The API provider designs and publishes an API based on initially provided consumer requirements but without consumer collaboration during API design and implementation (C4) to achieve fast time-to-market.
`Idea Backlog`	An idea backlog is a dynamic list that stores and aggregates consumer wishes for API endpoints derived from consumer support requests, discussions, or surveys.
`Testing Strategy`	A centrally defined testing strategy enforces the testing of new APIs or changes to existing APIs to reduce the likelihood of unexpected behavior of new or changed APIs or backends (C3).
`Data Clearing Process`	A data clearing process ensures that all API endpoints comply with legal and strategic requirements before they are published externally by involving different stakeholders who provide feedback and need to sign off on a new API or the change to an existing API.
`API Facade`	An API facade abstracts the invocation of several backend services into a single API (Gamma et al., 1995). The API facade thereby supports the tailoring of APIs that fit the user stories of the API consumers.
`API Quality Monitoring`	API quality monitoring describes continuously testing an API's non-functional properties to detect anomalies and take countermeasures quickly.

Table E.3.: Summaries of the API Consumer-facing Patterns adopted from Bondel and Matthes (2023).

Pattern Name	**Pattern Solution Description**
`Role-based Marketing`	Role-based marketing denotes the clear separation of marketing material and other consumer-facing resources targeted at different user roles in the developer portal.
`Customer Success Stories`	A customer success story exemplifies an API consumer's successfully finalized use case or product implementation utilizing the provider's APIs (C3) with the aim to demonstrate an API's potential to future consumers.
`Newsletter`	The API provider publishes summaries of changes to existing APIs (De, 2017) and other announcements related to APIs in a newsletter to keep current and potential future API consumers up-to-date.
`Consumer-centric API Description`	The API provider describes the API products functionality from a consumer perspective as use cases or user stories addressing a consumer's business need.
`Integration Guide`	An integration guide documents the implementation of common functionality using step-by-step instructions (Spichale, 2017) to reduce consumers' effort implementing the specific functionality (Medjaoui et al., 2018).
`Onboarding Self-service`	An onboarding self-service automates (parts of) the API onboarding process by allowing API consumers to choose a monetization plan, register a user account, generate authentication credentials, and register a finalized client application without interacting with API provider team members.
`Integration Partner Program`	API providers support API consumers with finding suitable integration partners by creating and maintaining a curated list of potential integration partners that meet specific quality criteria.
`API provider-wide Ticketing Management`	The API provider uses a uniform ticketing system that manages all API-related tickets and is available to all teams involved in API provision. Hence, the ticketing system enables transparency, e.g., on ticket resolution times or recurring issues.
`Dedicated Support Team`	The dedicated support team accepts all API consumers' questions, service requests, and incident reports and immediately answers or resolves low- or medium-complexity tickets. Only high-complexity tickets are forwarded to the respective experts, relieving the API and backend provider teams of a portion of the support activities.
`Service Level Agreement (SLA)`	An SLA is an agreement between two parties that specifies the quality of services, i.e., the APIs' non-functional properties and support service levels, as well as contractual punishments in case of SLA breaches. Hence, an SLA increases API consumers' trust in an API's quality.

APPENDIX F

AMPC Pattern Examples

In the following, we present the two patterns `Frontend Venture` and `Collaborative Pilot Project` as previously published in Bondel and Matthes (2023).

Frontend Venture

A previous version of this pattern has been published in Bondel et al. (2021b). In this pattern catalog, we evolved the pattern.

Pattern Overview	
Name	`Frontend venture`
Pattern Type	Interface Type Pattern
Summary	The API provider enables consumers who cannot, for any reason, integrate an API or client library, to still use the functionality or data via a simple frontend, i.e., a website with fields and buttons that trigger API functionality. If enough consumers are interested in the frontend, a product team can take over the development and maintenance of the frontend.

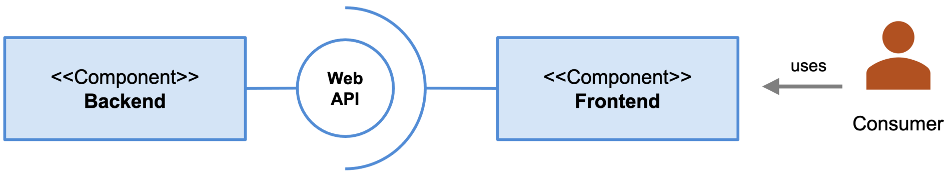


Figure F.1.: A `Frontend Venture` makes data and functionality available to consumers lacking the capabilities to use a `Web API` or `Client Library`.

Context:

Integrating an API into an existing IT landscape creates effort for the API consumers. However, some API consumers lack technical capabilities or the budget for API integrations and can thus not realize beneficial use cases. Such consumers are often municipalities or small, non-digital businesses (C3).

Concern:

How can an API provider enable API consumers that cannot, for any possible reason, integrate an API to use the API's functionality or data nonetheless?

Forces:

- *Consumer capabilities.* Some potential consumers, especially municipalities or small, non-digital organizations, are not able to use or integrate APIs since they have no or small IT departments that are already working at capacity. For example, in one observed case, the API consumers are small, non-digital organizations that usually only have a website based on a content management system maintained by freelancers (C3). Furthermore, these potential consumers often have no budget to hire external IT service providers to execute the integration project (C3).

- *Profits.* API consumers are willing to pay for solutions that meet their needs.
- *Public perception.* The API provider can make data and functionality available to municipalities or small, non-digital businesses in a way that supports some societal interest. Such initiatives are viewed positively by the public (C3).
- *Legal obligation.* An API provider can be legally obliged to make specific data available to certain consumer groups through APIs, even if some consumers cannot use the API (C3). Such legal obligations exist, for example, in the banking or automotive industry.
- *Effort.* API providers often work at capacity and consumers use a software solution only if the API provider can provide it with a certain level of quality.
- *Reusability.* Solutions always need to balance the specific needs of the first API consumer or end user with the reusability of the solution for other consumers.
- *Flexibility.* An API and an interface provide different flexibility with regards to integration (C9), automatizing (C9), customization (C7), and branding (C7).

Solution:

The API provider identifies and evaluates a use case with an API consumer or a consumer group and implements a frontend, i.e., a website with fields and buttons that trigger API functionality. As soon as the frontend reaches a certain level of maturity, the API provider markets it to other potential consumers. If enough consumers are interested, a product team takes over the development and maintenance of the product. Thus, implementing a frontend can be a venture opportunity.

Stakeholders:

The *API provider* has to design and implement the frontend. During the design and implementation, the API provider has to collaborate with the *API consumers* to ensure that the frontend meets the consumers' needs concerning functional and non-functional requirements. Furthermore, the API provider should aim to hand over the responsibility for the frontend to a dedicated *product team* after its publication.

Implementation Hints:

Basic approach. A frontend implements a specific use case for a consumer or consumer group. Therefore, the first step is to analyze the need of the future consumer or consumer group. Once the use case is defined, the API provider assesses the benefits and drawbacks. The benefits can include additional direct profits from billing the consumers, the chance of further profits from reselling the frontend to other external consumers, internal reuse of the frontend, or positive marketing impact (C3). The API provider has to weigh these advantages against the additional effort required to develop and maintain the frontend.

Furthermore, as part of the initial analysis, the API provider should check if similar frontends exist within the organization. If an internal team has already built a similar frontend, the API provider can reuse (parts of) it (C3). Based on this benefit-cost evaluation, the API provider decides if or how to implement the frontend.

Assuming the API provider decides to move forward with the frontend implementation, in the next step, the API provider designs and implements the frontend. The API provider can choose to employ either a `Collaborative Pilot Project` or a `Play-it-fast` to design and implement the frontend. An alternative approach would be to implement a simple first version of the frontend in the course of a `hackathon`[1].

After the frontend reaches a certain level of maturity, the API provider can present it to other interested parties. If enough consumers are interested in using the frontend, the API provider can hand it over to a dedicated product team to evolve and maintain it as a product following the `API-as-a-Product` pattern (C3). Thus, designing and implementing a frontend is a venture opportunity for the providing organization. The product team then acts as an IT provider to the consumer while the API platform provides the underlying API services (C3).

Types of frontends. A frontend realizes a specific use case and typically concentrates on a subset of the APIs functionality. In general, the goal is to provide a user interface using state-of-the-art design elements that consumers without technical capabilities can easily and intuitively use. The most basic type of a frontend is a user interface that simply makes the as-is API functionality usable for non-technical consumers. For example, the API provider can implement a website that allows users to upload data, set transformation parameters and response filtering options, and subsequently download a file containing the APIs response (C3; C7). A more advanced frontend can also implement some logic that augments the response data with additional data or visualizations. As an example, a frontend can show a map and locate certain events on it (C3).

Consequences:

Benefits:

- *Consumer capabilities.* The frontend allows API providers to make API data and functionality available to organizations with insufficient IT capabilities or budgets. Still, other interested API consumers with enough IT capabilities can consume the API to create custom integrations or proprietary frontends.
- *Profits.* The API provider can monetize the implemented frontend, and it can thus become a source of profit. If the API provider does not provide the frontend, a third party could skim these profits.

[1]A hackathon is an event with a pre-defined timeframe during which small development teams compete to implement the best solution to a pre-defined problem.

- *Public perception.* Making data and functionality available to municipalities or small, non-digital businesses in a way that supports some societal interest results in positive publicity for the API provider (C3).
- *Legal obligation.* In case of a legal obligation, the API provider has to implement the API anyway. Therefore, the API provider might view the legal obligation as an opportunity to create new business relationships or profits through the frontend (C3).

Drawbacks:

- *Effort.* The design, implementation, and especially the maintenance of the frontend create additional effort for the API provider. If the frontend is of insufficient quality, the API consumers will not use it, and it would thus be a loss of investment (C3). The API provider has to ensure that enough resources are available to realize a frontend venture.
- *Reusability.* The API provider has to put effort into balancing the needs of different frontend consumers (C3). If the API provider tailors the frontend too much to the need of one API consumer, other API consumers will not use the frontend. However, if the frontend is too generic, it might be of less value for all consumers.
- *Flexibility.* A frontend limits the consumers flexibility compared to an API with regards to integration (C9), automation (C9), configuration (C7), and branding (C7). Thus, it is important that the API provider also keeps providing the API.

Related Patterns within this Pattern Catalog:

An API provider can design and publish a `Frontend Venture` in addition to a `Web API` if, for any possible reason, the consumer cannot integrate a `Web API` (C3). However, the provider should also publish the `Web API` to preserve the flexibility of consumers that want to integrate it directly (C7; C9).

Also, the API provider can realize a `Frontend Venture` through a `Collaborative Pilot Project` or a `Play-it-fast`. Furthermore, `API Quality Monitoring` can monitor the breach of non-functional properties of a `Frontend Venture`. The `Dedicated Support Team` enables consumers to report issues related to `Frontend Venture` and a `Newsletter` can communicate changes of a `Frontend Venture`. Finally, an `Onboarding Self-service` enables access to a `Frontend Venture`.

Finally, the pattern `Frontend Venture` is suitable if the consumer has neither the technical nor the financial capabilities to integrate a `Web API`. However, suppose the consumer lacks only technical capabilities but has a sufficient budget for API integration. In that case, the API provider can alternatively use a `Integration Partner Program` to introduce integration partners to the consumer.

Known Uses:

We observed the pattern in three cases:

The API portal provider of an automotive organization (C3) wanted to provide data to two municipalities via APIs. However, the municipalities did not have the capabilities to integrate the APIs and requested a user interface that a non-technical stakeholder can use. The API provider implemented such an interface during a pilot project since the organization expected positive marketing effects. After the pilot phase, the API provider team handed the prototype over to a product team that now maintains the frontend. As a result, the API provider reports that only 50% of consumers directly access the API behind the frontend, while 50% use the frontend.

The organization C7 is an insurance subsidiary that provides insurance services within a group setting. The API management team offers a mix of APIs and frontends. If the API provider exposes an API directly or uses a frontend depends on the nature of the product. The products that need much integration into backend systems, customization, or branding are exposed as APIs. Additionally, the API provider provides frontends for products that do not need much integration. Those are primarily products offering only data access or simple functionality.

Finally, in C9, the organization offers a marketplace for IoT applications. The API provider again offers APIs and frontends, however, frontends are dominant. According to the interview partner, most consumers prefer frontends. Nevertheless, the API provider also provides APIs to enable integration and automation.

Cross-case observations:

All the cases are in early phases (pilot phase or early production phase). This makes sense, since it can be beneficial for API initiatives in early stages to implement frontends to attract first consumers. However, it is also essential to maintain the Web APIs since Web APIs provide more flexibility regarding backend integration or frontend customization to organizations with more IT capabilities or budget.

Also, not surprisingly, the consumers are rather small business or municipalities.

Collaborative Pilot Project

Pattern Overview	
Name	`Collaborative Pilot Project`
Pattern Type	API Provider Internal Pattern
Summary	The API provider designs a new API iteratively in close collaboration with one or a limited set of API consumers to increase the likelihood of the API meeting API consumers' needs.

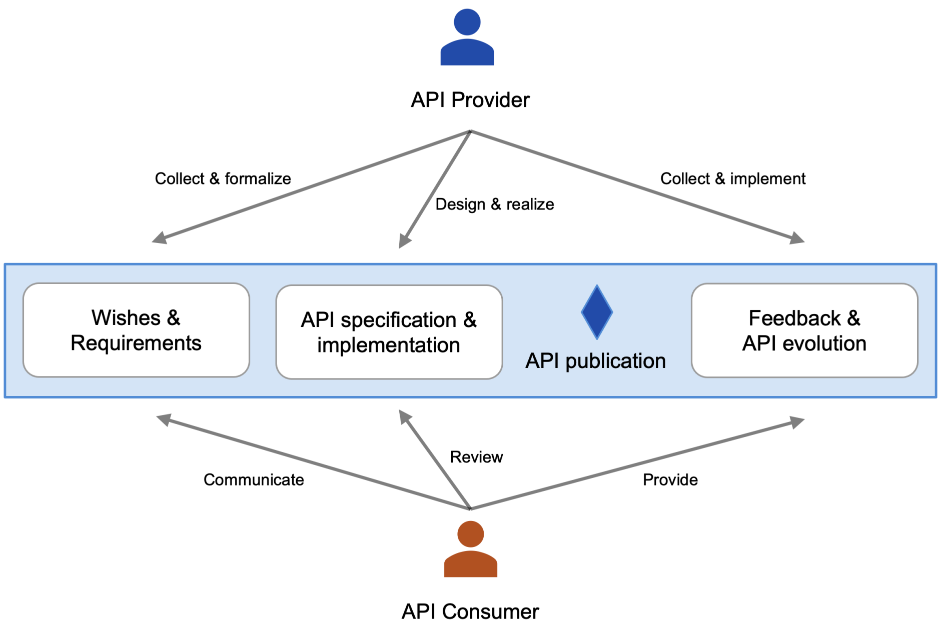


Figure F.2.: The pattern `Collaborative Pilot Project` involves one or a selected number of API consumers in all steps of the API design. This includes the API consumer reviewing and providing feedback on the API's specification and the prototypical implementation before publication to all consumers.

Context:

An API provider wants to create a new API. For an API to succeed, the APIs must meet the API consumers' needs. Especially for complex use cases, it can be challenging to understand the API consumers' needs correctly (C2).

Concern:

How can the API provider ensure that a new API meets the consumer's needs?

Forces:

- *Consumer demand.* The API provider needs to ensure that APIs meet the consumers' needs so that consumers adopt the APIs. The creation of an API that does not meet consumer needs leads to a loss of investment (C3).
- *Time-to-market.* In specific markets, i.e., high-speed markets, API providers must publish APIs as fast as possible to gain or maintain a competitive advantage (C3). Moreover, even outside of high-speed markets, API consumers that rely on the API to run their business can be negatively affected by long waiting times for new APIs (Medjaoui et al., 2018).
- *Monetization.* The API provider might want to create a new API only if its future monetization is secured. The API provider can ensure future API monetization by negotiating contracts with future API consumers before the API implementation.
- *API stability.* API consumers rely on the stability of the APIs that they integrate. Frequent changes, especially breaking changes, creates effort for API consumers (C2) and can lead to them abandoning an API (Medjaoui et al., 2018).
- *Generic API design.* In public API and marketplace settings, the API provider wants to design APIs to meet the needs of several API consumers.

Solution:

The API provider designs the API iteratively in close collaboration with one or a limited amount of API consumers in a pilot project. The partnering API consumer communicates needs and requirements and provides feedback to the API provider before and during the API implementation project (C3). Hence, the API provider can easily realize changes to the API design before its publication, i.e., before the API is accessible to and integrated by consumers (Medjaoui et al., 2018).

Stakeholders:

The *API provider* has to collaborate with the partnering *API consumer(s)* to discuss and test API implementations during the pilot phase. Usually, these collaborations are based on a contractual agreement. Thus, the provider has to involve the *legal* team. Also, the API provider has to collaborate with the *backend provider(s)* who provide the functionality accessible via the API.

Implementation Hints:

Partner identification. A collaborative pilot project starts with identifying a use case and a consumer who wants to partner. The API provider can identify use cases and partnering consumers during informal direct discussion or via account management or other consumer-facing teams. Alternatively, the API provider can use a more structured process, e.g., the use of an `Idea Backlog`. However, it is also possible that an API consumer actively approaches an API provider and proposes a pilot project. In all cases, the API provider has to evaluate the business case before agreeing to the partnership (C3).

For public APIs, the API provider can also approach top developers with respected positions in specific developer communities. These developers' involvement in the design of new APIs enables improvements of the API and free marketing through evangelization (Jacobson et al., 2012).

<u>Collaboration mode agreement.</u> The partners must negotiate the legal basis for the project. Also, the pilot project partners have to formalize the collaboration and agree upon general conditions, including the mode and intervals of collaboration. These topics can be discussed during a `Pilot Workshop`, which launches the pilot project.

<u>Collaboration modes.</u> The API provider can collect feedback using lab-based usability tests, focus groups, surveys, and interviews (Medjaoui et al., 2018).

Generally, the API consumer can provide feedback at some or all of the following stages of the collaborative pilot project. First, the API provider and consumer can create, discuss, and agree upon API specifications before the API implementation (C3). For example, mocks form a basis to discuss and review APIs (Spichale, 2017). Also, API consumers can review and test the API implementation at specific points during the API implementation, e.g., after each iteration in an agile context (C2; C3). Finally, the consumer can perform the final acceptance test before an API is released into production and potentially made accessible to other external consumers (C2). Furthermore, the API provider can also request feedback from the API consumer on additional artifacts, like the developer portal or API documentation (C2).

Consequences:

Benefits:

- *Consumer demand.* Collecting consumer feedback before an API's publication reduces the need to make assumptions about the API design (Medjaoui et al., 2018) by creating a better understanding of API consumers and their needs. Thus, the chances of API adoption increase. Also, close collaboration creates API consumer buy-in.
- *Monetization.* An API consumer is more likely to sign a contract to use and pay for an API before its implementation if they are closely involved during the APIs design and implementation phases. Hence, a `Collaborative Pilot Project` increases the likelihood of an API consumer agreeing to an upfront contractual monetization agreement.
- *API stability.* It is harder to change APIs after their publication when API consumers have already built integrations (Medjaoui et al., 2018). Iterative improvement of an API based on consumer feedback before publication should reduce the need to make changes after publication. A stable API with few changes affecting consumer integrations prevents a negative impact on API consumer satisfaction.

Drawbacks:

- *Consumer demand.* API providers sometimes have to implement APIs for legal reasons. In such settings, the consumers' demand for such APIs is a secondary concern for the API provider (C3).
- *Time-to-market.* The API provider depends on the partner providing feedback while testing a new API design. API consumers can have other priorities in their daily business and may not provide feedback to the API provider in a timely manner. Thus, close collaboration with an API consumer can lead to waiting times for the API provider team (C4). Such waiting times disrupt the development processes of the API provider, and postpone planned API go-live dates (C4).
- *Generic API design.* Close collaboration with just one API consumer during the design of an API can lead to a specialized API. Potentially, only one or a small group of API consumers can use an overfitted API (C4). A specialized API is suitable if the API provider aims for a point-to-point integration but not if the API provider wants to provide a public API or a platform in a developer ecosystem that many API consumers can integrate (C12). Finally, close collaboration with third parties comes with the risk of scope creeping (Kendrick, 2015).

Related Patterns within the Pattern Catalog:

A `Collaborative Pilot Project` is an approach to realize a new `Web API`, `Frontend Venture`, or `Client Library` that passed the `API Clearing Process`. The provider can apply a `Testing Strategy` for testing as part of the `Collaborative Pilot Project`.

A `Collaborative Pilot Project` presents an alternative approach to designing and implementing a new API compared to the `Play-it-fast`. While `Collaborative Pilot Project` increases the likelihood of a new API meeting consumer needs due to close and ongoing collaboration, it also increases time-to-market. In comparison, a `Play-it-fast` enables fast time-to-market for new APIs but also increases the risk of not meeting API consumer needs.

Known Uses:

We observed the pattern in three cases:

In case C2, the API provider offers simulation and modeling algorithms for the analysis of energy data. The API provider initiates a new project only if a concrete demand exists, i.e., if an API consumer agrees to partner with them during a pilot project. The API provider gives the collaborating partner access to the developer portal and asks them for feedback. Based on the feedback, the API provider evolves the API endpoint.

The API portal provider of an automotive organization (C3) shares anonymized road condition data with public authorities. The API provider partners with API consumers in pilot projects to better understand the market demand for a new API. Different stakeholders can trigger new projects, but primarily external organizations with prior relations to the provider organization contact the API provider team with new use cases. Furthermore, the API provider aims to design a first prototype of the API endpoint to show to the consumer as fast as possible. Based on the prototype, which can be nothing more than a handwritten specification in the first iteration, the API provider discusses the API endpoint design with the consumer. However, the API provider admits that this kind of close collaboration is not always possible for each API endpoint. Instead, in some cases, the API provider also applies the `Play-it-fast`, i.e., develops and releases an endpoint based on only one prior discussion with a potential API consumer to realize a faster time-to-market.

Case C12 captures a financial services provider that provides SaaS software to end-users. APIs enable other software providers to integrate their software with the SaaS system of C12, thus offering an integrated solution to the end users. The API provider closely collaborates with consumers when creating new APIs for the software product since API consumers often have very concrete expectations regarding the API. However, this approach leads to the creation of several APIs for the same software product, i.e., one API for each consumer.

Cross-case observations:

These cases represent API initiatives still in a pilot phase or in production. The API initiatives are partner and public API initiatives but have a relatively small number of API consumers. This makes sense since collaborative pilot projects usually focus on the specific requirements of single API consumers.

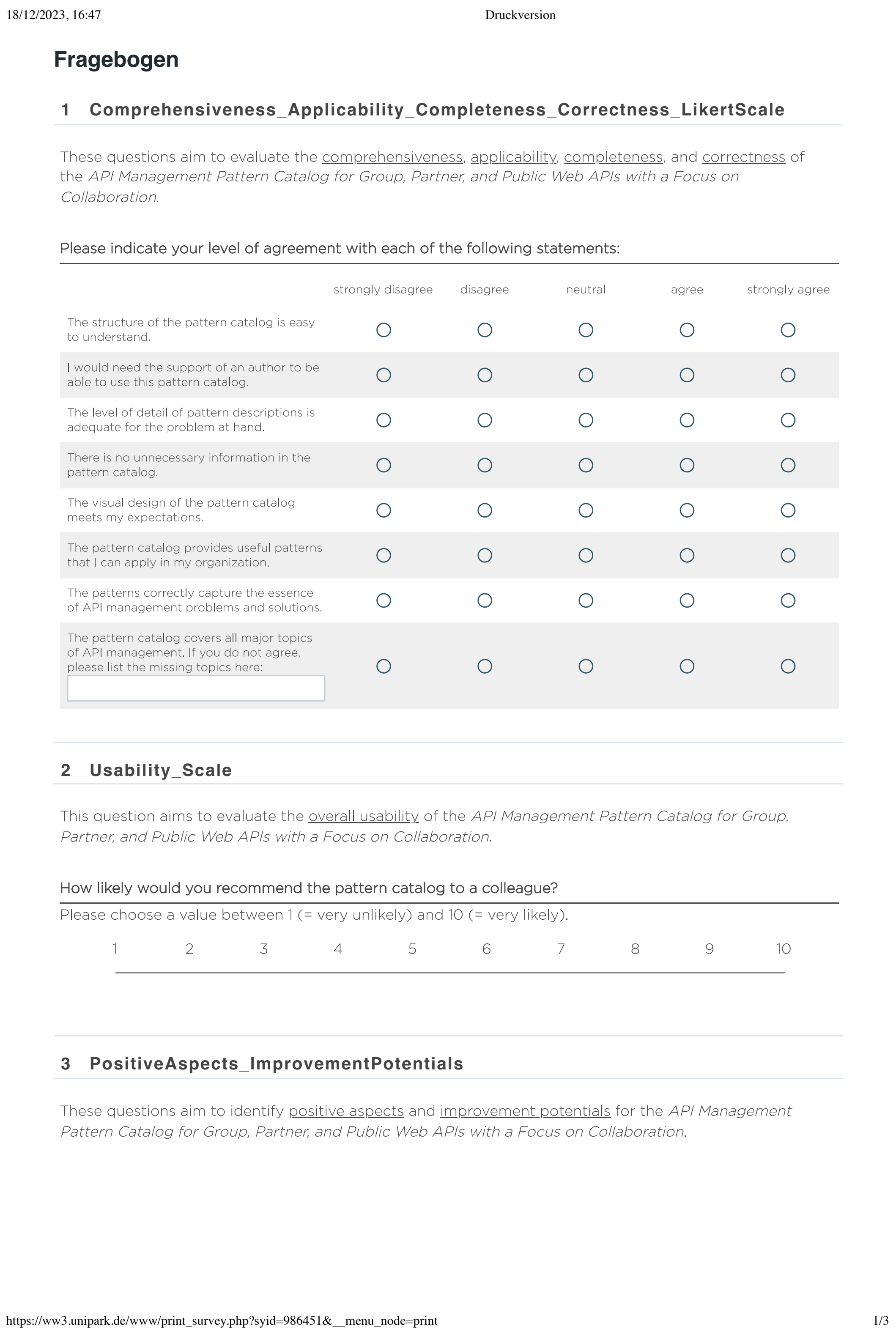
Further, the API providers apply `Collaborative Pilot Project` for API initiatives with other business organizations as well as with government institutions.

APPENDIX G

Survey Questions

This section presents the survey questions and structure used to evaluate the AMPC (Bondel and Matthes, 2023) from a practitioners point of view as detailed in Section 7.1.2. We used the Unipark[1] survey software to create and conduct the survey.

[1]`https://www.unipark.com/umfragesoftware-bestellen/`.



18/12/2023, 16:47 Druckversion

Fragebogen

1 Comprehensiveness_Applicability_Completeness_Correctness_LikertScale

These questions aim to evaluate the comprehensiveness, applicability, completeness, and correctness of the *API Management Pattern Catalog for Group, Partner, and Public Web APIs with a Focus on Collaboration*.

Please indicate your level of agreement with each of the following statements:

	strongly disagree	disagree	neutral	agree	strongly agree
The structure of the pattern catalog is easy to understand.	○	○	○	○	○
I would need the support of an author to be able to use this pattern catalog.	○	○	○	○	○
The level of detail of pattern descriptions is adequate for the problem at hand.	○	○	○	○	○
There is no unnecessary information in the pattern catalog.	○	○	○	○	○
The visual design of the pattern catalog meets my expectations.	○	○	○	○	○
The pattern catalog provides useful patterns that I can apply in my organization.	○	○	○	○	○
The patterns correctly capture the essence of API management problems and solutions.	○	○	○	○	○
The pattern catalog covers all major topics of API management. If you do not agree, please list the missing topics here:	○	○	○	○	○

2 Usability_Scale

This question aims to evaluate the overall usability of the *API Management Pattern Catalog for Group, Partner, and Public Web APIs with a Focus on Collaboration*.

How likely would you recommend the pattern catalog to a colleague?

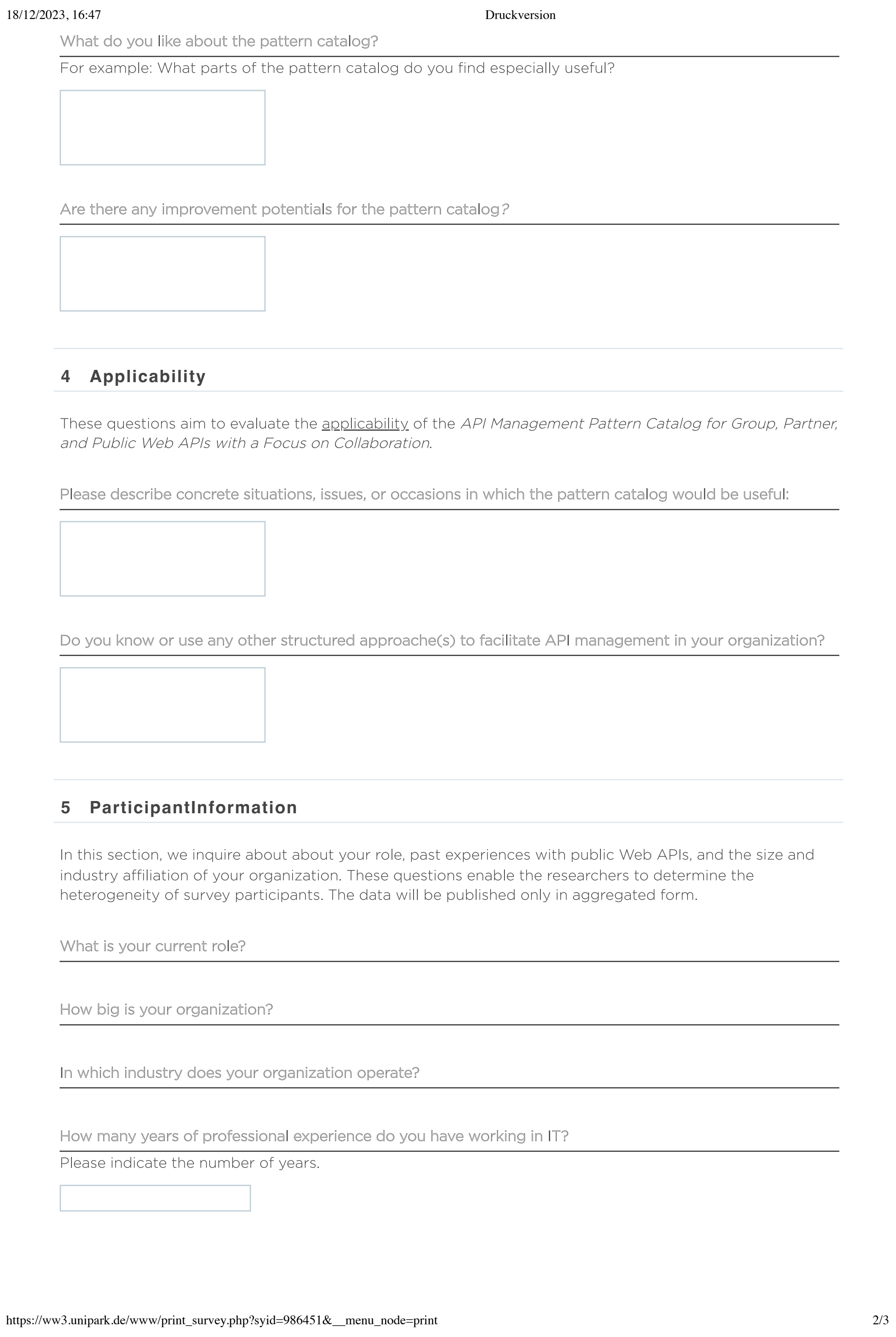
Please choose a value between 1 (= very unlikely) and 10 (= very likely).

1	2	3	4	5	6	7	8	9	10

3 PositiveAspects_ImprovementPotentials

These questions aim to identify positive aspects and improvement potentials for the *API Management Pattern Catalog for Group, Partner, and Public Web APIs with a Focus on Collaboration*.

https://ww3.unipark.de/www/print_survey.php?syid=986451&__menu_node=print 1/3



18/12/2023, 16:47 Druckversion

What do you like about the pattern catalog?

For example: What parts of the pattern catalog do you find especially useful?

Are there any improvement potentials for the pattern catalog?

4 Applicability

These questions aim to evaluate the applicability of the *API Management Pattern Catalog for Group, Partner, and Public Web APIs with a Focus on Collaboration*.

Please describe concrete situations, issues, or occasions in which the pattern catalog would be useful:

Do you know or use any other structured approache(s) to facilitate API management in your organization?

5 ParticipantInformation

In this section, we inquire about about your role, past experiences with public Web APIs, and the size and industry affiliation of your organization. These questions enable the researchers to determine the heterogeneity of survey participants. The data will be published only in aggregated form.

What is your current role?

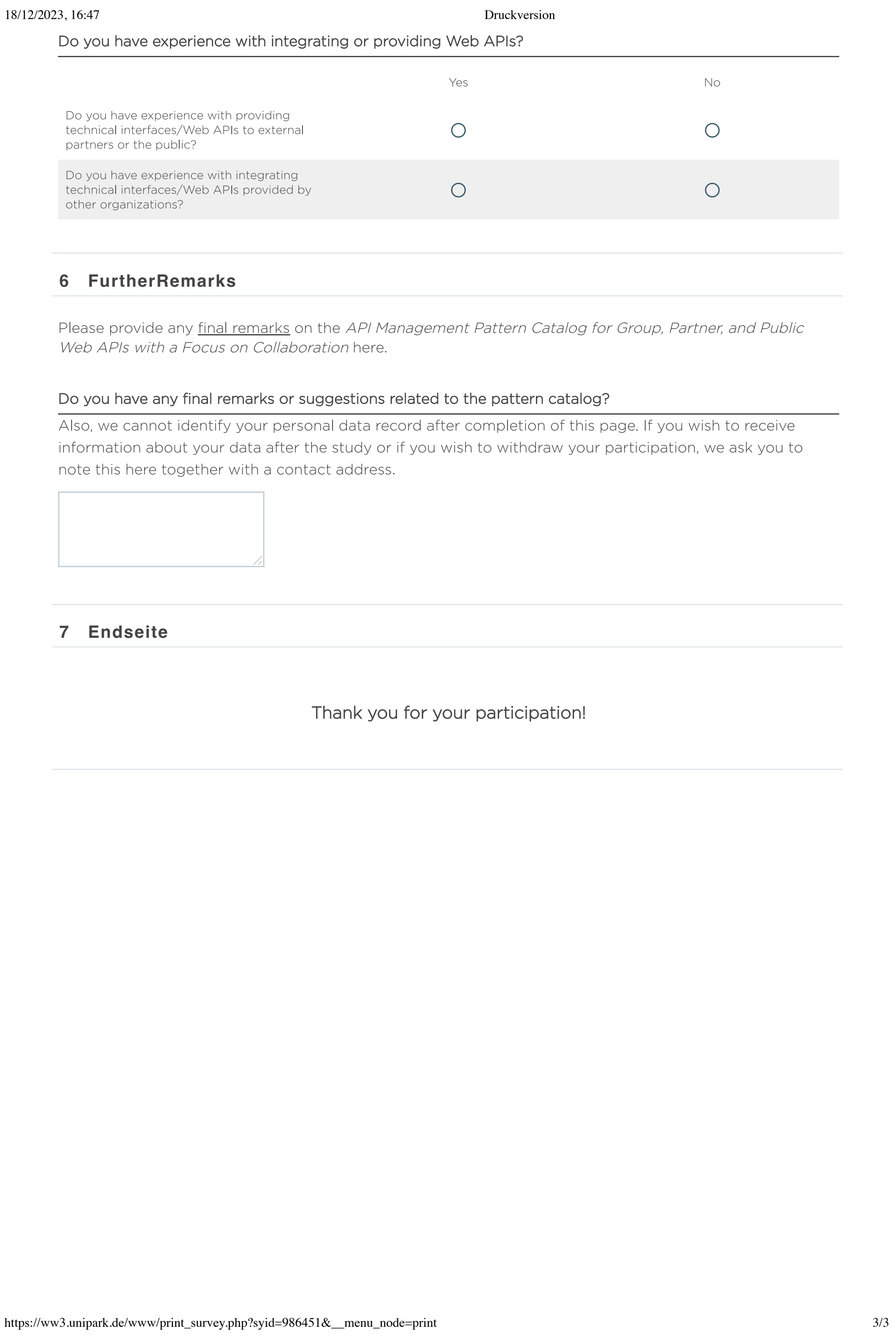
How big is your organization?

In which industry does your organization operate?

How many years of professional experience do you have working in IT?

Please indicate the number of years.

https://ww3.unipark.de/www/print_survey.php?syid=986451&__menu_node=print 2/3



18/12/2023, 16:47 Druckversion

Do you have experience with integrating or providing Web APIs?

	Yes	No
Do you have experience with providing technical interfaces/Web APIs to external partners or the public?	○	○
Do you have experience with integrating technical interfaces/Web APIs provided by other organizations?	○	○

6 FurtherRemarks

Please provide any final remarks on the *API Management Pattern Catalog for Group, Partner, and Public Web APIs with a Focus on Collaboration* here.

Do you have any final remarks or suggestions related to the pattern catalog?

Also, we cannot identify your personal data record after completion of this page. If you wish to receive information about your data after the study or if you wish to withdraw your participation, we ask you to note this here together with a contact address.

7 Endseite

Thank you for your participation!

https://ww3.unipark.de/www/print_survey.php?syid=986451&__menu_node=print 3/3