TUM Department of Informatics
Technical University of Munich

Master’s Thesis in Information Systems

# Identification of API Management Patterns From an API Provider Perspective

**Andre Landgraf**

TUM Department of Informatics
Technical University of Munich

Master’s Thesis in Information Systems

# Identification of API Management Patterns From an API Provider Perspective

# Identifizierung von API Management Patterns aus Sicht des API Anbieters

| | |
|---|---|
| Author: | Andre Landgraf |
| Supervisor: | Prof. Dr. Florian Matthes |
| Advisors: | Gloria Bondel, M. Sc. |
| Submission Date: | February 15, 2021 |

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, February 15, 2021

ANDRE LANDGRAF

# Abstract

Application Programming Interfaces (APIs) provide the means for software engineers to co-create software systems. In today's distributed software architectures, web-based Application Programming Interfaces (web APIs) are used to enable loose coupling of software components and services. The co-creation of software systems demands new management responsibilities such as API management. The corresponding literature is sparse and lacks standards. The goal of this thesis is to identify API management concerns and document practical solutions from an API provider perspective. The communication between the different API provider and API consumer entities is the focus of this thesis. The final outcome is a pattern catalog. The pattern catalog links detected stakeholders, concerns, and influence factors to solution approaches. Each solution approach is documented as a pattern. Overall, four stakeholders, 32 concerns, 35 pattern candidates, and 23 validated patterns are documented in this study. To achieve described objectives, this study draws from both design and behavioral science research. An extensive knowledge base grounded in literature reviews is utilized to create the foundations for this thesis. The study is evaluated and justified through 16 semi-structured interviews with API provider stakeholders. The rule of three known uses within studied cases is utilized to validate pattern candidates as patterns.

# Contents

| | |
|---|---|
| **API** | Application Programming Interface |
| **B2B** | Business to Business |
| **B2C** | Business to Consumer |
| **B2G** | Business to Government |
| **BaaS** | Backend as a Service |
| **GDPR** | General Data Protection Regulation |
| **HTTP** | Hypertext Transfer Protocol |
| **IaaS** | Infrastructure as a Service |
| **IS** | information system |
| **ITIL** | Information Technology Infrastructure Library |
| **KPI** | Key Performance Indicator |
| **OAS** | OpenAPI Specification |
| **PaaS** | Platform as a Service |
| **REST** | Representational State Transfer |
| **RQ** | research question |
| **SaaS** | Software as a Service |
| **SDK** | Software Development Kit |
| **SLA** | service-level agreement |
| **SOA** | service-oriented architecture |
| **SOAP** | Simple Object Access Protocol |
| **URL** | Uniform Resource Locator |
| **web API** | web-based Application Programming Interface |

# 1 Introduction

## 1.1 Motivation

Digitization is changing every aspect of our lives (European Commission. JRC., 2019). It is disrupting industries and creating both opportunities and challenges (Yoo et al., 2010). New digital technologies and distributed system architectures allow for a better integration of products and services (Löhe & Legner, 2010b; Lübke et al., 2019). APIs enable software engineers to access utilities through defined interfaces (Basole, 2016). In cloud-native applications, microservice architectures, and the service-oriented architecture (SOA), each software service offers remotely accessible APIs (Fehling et al., 2014; Josuttis, 2007; Lübke et al., 2019; Papazoglou & van den Heuvel, 2007; Zimmermann, 2017). Since most are offered over the web, they are also called web APIs (Alonso et al., 2004; Lübke et al., 2019). Web APIs enable loose coupling and ease the integration and reuse of services but they also create new management tasks such as service brokering (Papazoglou & van den Heuvel, 2007).

Companies create digital platforms to manage the brokering of their services (de Reuver et al., 2018; Lübke et al., 2019). Platforms can be used as controlled environments for co-creation (Skog et al., 2018; Yoo et al., 2010). The incumbent technology firms take advantage of platform network effects and exercise control over ecosystems of products, so called software ecosystems (Bianco et al., 2014; Jansen & Cusumano, 2012; Manikas & Hansen, 2013; Skog et al., 2018). The central hubs within those ecosystems are called industry platforms and enable new business models as they scale among many factors (Gawer & Cusumano, 2014). Today, the incumbent technology companies are some of the most valuable companies in the world by market capitalization (Kelly, 2016). Since APIs are the interfaces to the platform's capabilities, the digital platform literature defines them as boundary resources (de Reuver et al., 2018).

New product architectures implement network and service layers to integrate and offer remote services (Yoo et al., 2010). Ultimately, this allows every digitized product to act as a platform (Yoo et al., 2010). Yoo et al. (2010) define this new product architecture as the layered modular architecture (Yoo et al., 2010). This interconnection between digitized products and services would not be possible without APIs, more specifically web APIs (Bonardi et al., 2016). They instigate new service economies and enable new business models (Basole, 2016; Bondel et al., 2020; Tan et al., 2016). The service-based digital integration between business partners reduces interaction costs and allows new value-adding composition of services (Löhe & Legner, 2010b; Tan et al., 2016). Today, the SOA emerges into an API-based service ecosystem which is referred to as the API Economy (Basole, 2016; Bondel et al., 2020; Tan et al., 2016). In the API Economy, service providers are also API providers (Lübke et al., 2019). Since each service in the ecosystem might be offered by a different organization, each API provider might have different goals and thus, different API strategies (Lübke et al., 2019).

In the following, a robot vacuum cleaner is used as an example to emphasize the described developments. Robot vacuum cleaners are digitized products that take advantage of sensors and digital signals for navigation. Furthermore, the vacuum cleaners act as a digital platform that integrates with value-adding services and other third-party platforms[1]. The vacuum cleaner can be controlled through smart speakers and voice assistants. The user authentication happens via the smartphone app that can be downloaded over the app store and offers additional features. Finally, since the vacuum cleaner is connected to Wi-Fi, it is capable of over-the-air software updates[2].

This example demonstrates the level of interconnection between today's products and services that is enabled by the API Economy. The end user is offered a new level of convenience and smarter products that share data and functions with each other (Bonardi et al., 2016). For businesses, however, the adoption of mentioned technologies and architectures requires a high level of technological responsiveness (Nicholls-Nixon & Woo, 2003). This is especially hard to achieve for established firms with existing and legacy structures (Bondel et al., 2020; Nicholls-Nixon & Woo, 2003). These organizations have to overcome legal, economic, social, technological, and organizational barriers to utilize the API Economy (Bondel et al., 2020). APIs are software artifacts and their evolution requires collaboration across organizational boundaries (Espinha et al., 2014; Fokaefs et al., 2011; Koci et al., 2019; Lübke et al., 2019).

From a research perspective, the socio-technological advancements change how organizations utilize information systems (Yoo et al., 2010). Yoo et al. (2010) argue that information system (IS) research needs to address these changing factors (Yoo et al., 2010). They propose new research questions and emphasize the importance of boundary resources for future research (Yoo et al., 2010). Henfridsson and Bygstad (2013) stress that boundary resources should be the unit of analysis since they facilitate the relationship between the platform's stakeholders (Henfridsson & Bygstad, 2013). De Reuver et al. (2017) argue that the effect of long-term decisions cannot be predicted easily and state that studies on the evolution of digital platforms and ecosystems are required (de Reuver et al., 2018; Germonprez & Hovorka, 2013). Koci et al. (2019) explain that the API management from a provider perspective lacks attention (Koci et al., 2019). Mathijssen et al. (2020) highlight that API management research is sparse and that best practices have to be identified (Mathijssen et al., 2020). To conclude, the API Economy draws from many different areas of research. The management of APIs is a highly requested field of research (Mathijssen et al., 2020).

[1]https://www.irobot.com/roomba/s-series

[2]https://homesupport.irobot.com/app/answers/detail/a_id/550/~/how-do-i-get%2Freceive-a-software-update-for-my-wi-fi-connected-robot

## 1.2 Objectives

The goal of this thesis is to identify API management concerns and document practical solutions from an API provider perspective. It is the API provider that has to manage the offered APIs and services. Therefore, we approach the concerns and solutions based on the view of the API provider. The API provider consists of several roles, teams, and stakeholders that have to collaborate to efficiently and effectively manage the API provision. Additionally, collaboration with the API consumer is required to adapt to customer needs. The communication between the different API provider and API consumer entities is set to be the focus of this thesis.

To collect data, we conducted semi-structured interviews with API provider stakeholders. In order to describe concerns and solutions in a standardized manner, a pattern catalog is developed as proposed by Buckl et al. (2013) (Buckl et al., 2013). Hereby, a pattern is a documented solution for common concerns based on a particular context (Buckl et al., 2008; Gamma et al., 1994). The pattern definition from Buckl et al. (2013) draws a clear path to creating a catalog of patterns (Buckl et al., 2013). As a result, we derive the following three research questions (RQs):

- **RQ1: What concerns do API providers face in their daily work?**
- **RQ2: What influence factors impact the API management?**
- **RQ3: How do API providers manage concerns and what is the rationale behind the solutions?**

First, the concerns of API providers have to be discovered and documented. RQ1 focuses on the common impediments of API management. Second, the specific context of each studied case has to be captured. To answer RQ2, this thesis aims to categorize and understand the context that influences decision making of the API provider stakeholders. De Reuver et al. (2018) argue for conceptual and methodological innovation and stress the fundamental differences of digital platforms to other fields of research (de Reuver et al., 2018). Since discovered patterns cannot claim to be generally valid without evaluation and further research, RQ2 addresses the specific context in which the concerns and solutions were discovered. Third, RQ3 follows up on the concerns from RQ1 and asks for solution approaches and their rationales. Yoo et al. (2010) call for research about the "methodological and technological principles of the design of technical boundary resources that help sustain continued developments of novel components in doubly distributed networks" (Yoo et al., 2010, p. 17). This study aims to address this research gap by raising RQ3. The objective is to gain insights about utilized best practices.

The outcomes from RQ1 and RQ2 create building stones that enable the creation of a final result for RQ3. RQ1 results in a list of concerns and gives insights into who raises those issues and what software artifacts are involved. The discovered relationships between roles, teams, stakeholders, and software artifacts result in a stakeholder-relationship map. RQ2 focuses on the context in which the concerns are raised. As argued by Buckl et al. (2013), the specific context should be part of each pattern (Buckl et al., 2013). The context aims to support the decision making process (Uludağ et al., 2019). The goal is to assign each pattern influencing factors by which the solution was found to be practical to solve the defined concerns. RQ3 builds on top of RQ1 and RQ2 to develop the pattern catalog. The second part of the interview questionnaire is used to gain insights into revised solutions and the reasoning behind the solution approaches. The pattern catalog connects the identified concerns, stakeholders, and context to solution approaches in form of patterns. Ultimately, the pattern catalog consists of a list of patterns that offer API providers solutions for recurring management concerns.

## 1.3 Outline

The remainder of this thesis is structured as follows. Chapter Foundations gives an overview of related concepts and adds to the knowledge base used within this thesis. Chapter Related Work follows with an overview of related literature from associated areas of research and the pattern literature. The research approach is laid out in chapter 4. The research methodologies are presented and insights about the data collection and pattern creation process are explained. Next, in chapter API Management Pattern Catalog, the novel artifacts are documented. Chapter Discussion addresses the research questions and critically analyzes the results. This leads to the final summary in chapter 7. The summary includes a short conclusion and lists realized and open goals. The identified limitations of this thesis are raised and an outlook for future work is given.

Chapter 5 is the main part of this thesis. It is structured in the sections Data Collection, Pattern Language, Roles and Stakeholders, Influence Factors, Concerns, Taxonomy, and API Management Patterns. First, the data collection is presented in section 5.1. Second, the pattern language is laid out. Section 5.3 presents the discovered roles, teams, and stakeholders and documents the stakeholder-relationship map. Section Influence Factors illustrates the creation of context attributes and describes the context distribution matrix. Next, a list of concerns is documented in section 5.5. Section 5.6 visualizes the pattern catalog based on its taxonomy and documents the pattern categories. The core of the pattern catalog materializes in a list of documented patterns and pattern candidates in section API Management Patterns.

# 2 Foundations

This study builds on foundations from several fields of research. In this chapter, the related literature is reviewed and the most important concepts and key terminology are laid out. First, the platform literature is reviewed. Second, the terms API, web API, and Software Development Kit (SDK) are defined and put into relation with each other. Third, scientific literature about knowledge transfer is summarized. Next, the related fields SOA, cloud computing, and web services are reviewed. An overview of the API management follows. At the end of this chapter, the term API Economy is derived.

## 2.1 Platforms and Boundary Resources

In this section, foundations of the platform and boundary resources literature are presented. Furthermore, a connection to the software ecosystem literature is drawn. The platform literature provides important foundations for this study. APIs are researched in the context of platforms (de Reuver et al., 2018; Yoo et al., 2010). Moreover, API management utilizes API management platforms (De, 2017, p. 15). Understanding the interconnection between platforms, their boundary resources, and the surrounding software ecosystems is required to build a knowledge base and common vocabulary.

### Platforms

In the following, a review of the platform literature is given. First, the platform concept is introduced from a management perspective. After that, the term digital platform is defined from a IS research perspective. The characterization of digital platforms leads to related term definitions in sections Boundary Resources and Software Ecosystems.

Platform literature is fragmented into several streams of research (Gawer, 2009; Skog et al., 2018; Thomas et al., 2014). One broad definition that is shared among the streams comes from Gawer (2009) who defines a platform as "a set of stable components that supports variety and evolvability in a system by constraining the linkages among other components" (Gawer, 2009, p. 19). Thomas et al. (2014) identify more than 900 papers in the management field that are related to the keyword platform and map the following four distinct yet overlapping fields of research:

- Organizational
- Product Family
- Market Intermediary
- Platform Ecosystem

The organizational stream treats organizations as platforms and analyses dynamic capabilities and organizational competencies (Thomas et al., 2014). The product family stream researches platforms that enable the reuse of core components within several products (Skog et al., 2018; Thomas et al., 2014). The market intermediary stream of literature researches multi-sided markets (de Reuver et al., 2018; Gawer, 2014). Multi-sided markets describe platforms where two groups of agents engage and the benefits awarded to each group depend on the size of the other group (Armstrong, 2006). This phenomenon is also known as an indirect or cross-side network effect (Bianco et al., 2014; Boudreau, 2011; de Reuver et al., 2018). In the platform ecosystem stream of research, a platform is described as a central hub that exercises control over a technology-based ecosystem (Thomas et al., 2014). In this context, Gawer and Cusumano (2014) define products, services, or technologies as industry platforms that provide a foundation for third-parties to co-create (Gawer & Cusumano, 2014).

De Reuver et al. (2018) offer an extensive analysis of the platform literature from an IS perspective (de Reuver et al., 2018). They differentiate digital and non-digital platforms (de Reuver et al., 2018). The non-digital platform definition draws from the management research as presented above. The digital platforms definition varies within the literature (de Reuver et al., 2018). Digital platforms can be defined as "purely technical artifacts where the platform is an extensible codebase, and the ecosystem comprises third-party modules complementing this codebase" (de Reuver et al., 2018, p. 126). In contrast, digital platforms are also defined as socio-technical concepts that embed technical artifacts (de Reuver et al., 2018). Digital platforms also build upon concepts from the management research, especially the product family and platform ecosystem streams (de Reuver et al., 2018; Skog et al., 2018). Furthermore, they can produce network effects as researched in the market intermediary stream (de Reuver et al., 2018). One key difference to non-technical platforms is the utilization of layered architectures that enable digital interconnection (de Reuver et al., 2018; Skog et al., 2018; Yoo et al., 2010).

A platform can be classified by its architectural openness (de Reuver et al., 2018; Thomas et al., 2014). Gawer and Cusumano (2014) specify internal platforms as a set of key components that enable efficient development of products and services (Gawer & Cusumano, 2014). Those platforms are not open to external parties (Gawer & Cusumano, 2014). Internal platforms become industry platforms when the network topology changes. A many-to-one topology appears when either the supply or demand side is opened to third-parties (Thomas et al., 2014). If both the supply and demand sides are open, the platform becomes a multi-sided market which is distinguished by a many-to-many network topology (Thomas et al., 2014).

Yoo et al. (2010) argue that digitization instigates a new product architecture, the layered modular architecture, effectively transforming every product into a platform (Yoo et al., 2010). In order for a platform to be accessible, it has to implement network and service layers (Yoo et al., 2010). The components that control the platform's boundaries and act as interfaces to the outside are called boundary resources (de Reuver et al., 2018; Yoo et al., 2010). Digitized products that implement layered modular architectures enable the integration with remote services on the service layer (Yoo et al., 2010). Karhu et al. (2018) argue that boundary resources can be used as a mean to control the openness of a digital platform (Karhu et al., 2018). The management of boundary resources is strategically important (Yoo et al., 2010).

### Boundary Resources

Ghazawneh and Henfridsson (2013) define boundary resources as "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 & Henfridsson, 2013, p. 174). They provide capabilities to both the third-party developers and applications (Bianco et al., 2014). Thus, boundary resources are the link that enables co-creation (Eaton et al., 2015). Two common instances of boundary resources are APIs and SDKs (de Reuver et al., 2018; Eaton et al., 2015; Yoo et al., 2010).

Boundary resources are commonly categorized as either technical or social resources (Bianco et al., 2014; Yoo et al., 2010). Social boundary resources include "incentives, intellectual property rights and control" (Yoo et al., 2010, p. 14). Technical boundary resources describe software artifacts like APIs and SDKs. Bianco et al. (2014) further break technical boundary resources up into application boundary resources and development boundary resources (Bianco et al., 2014). Since digital platforms provide capabilities for developers and applications, they argue for a differentiation between technical boundary resources used by developers and by applications (Bianco et al., 2014). Application boundary resources are consumed by the application (Bianco et al., 2014). They provide the interface between the application architecture and the platform architecture (Bianco et al., 2014). APIs are considered application boundary resources (Bianco et al., 2014). Development boundary resources, like SDKs, assist the third-party developer in the implementation and integration of the application over the software life-cycle of both the application and platform (Bianco et al., 2014).

The tuning of boundary resources is a management task with the goal to secure control over the platform ecosystem (Eaton et al., 2015). In this process, the boundary resources evolve (Eaton et al., 2015). Eaton et al. (2015) argue that the evolution of boundary resources results from "multilayered, overlapping, and contradicting actions by heterogeneous actors and artifacts" (Eaton et al., 2015, p. 241). The evolution of boundary resources is influenced by non-deterministic factors from the surrounding platform ecosystem which makes boundary resources themselves co-created (Eaton et al., 2015).

## Software Ecosystems

Today's software is no longer developed by individual organizations but co-created by a network of different entities (Jansen et al., 2009). This instigates the creation of software ecosystems (de Reuver et al., 2018; Jansen & Cusumano, 2012; Jansen et al., 2009). As defined in section Platforms, in the platform ecosystem stream of literature, platforms exercise control over technology-based ecosystems (Thomas et al., 2014). The platform leader is also referred to as keystone firm (Bianco et al., 2014; Gawer, 2014). The control over a network of firms and individual agents grants the keystone firm an ecosystem for innovating new products and services (Gawer, 2014; Ghazawneh & Henfridsson, 2010). Digital innovation enables extensive development possibilities (Boudreau, 2011). It is characterized as representational and informational (Boudreau, 2011). Thereby possessing 'generativity', a term used to illustrate the high pace of novelty (Boudreau, 2011; de Reuver et al., 2018; Yoo et al., 2010; Zittrain, 2006). As a result, at the heart of digital platforms lies the rapid innovation enabled by reuse and recombination (Boudreau, 2011; Lemley & Cohen, 2000; Yoo et al., 2010).

The term software ecosystem is researched from a business and management, a software architecture, and a software engineering perspective (Manikas & Hansen, 2013). One common definition of the term is given by Jansen and Cusumano (2012) (Jansen & Cusumano, 2012; Manikas & Hansen, 2013). They define: "A software ecosystem is a set of actors functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently underpinned by a common technological platform or market and operate through the exchange of information, resources and artifacts." (Jansen & Cusumano, 2012, p. 46). Thus, the software ecosystem can be described as a software related subset of a business ecosystem (Jansen & Cusumano, 2012).

Jansen and Cusumano (2012) use the term software ecosystem to highlight the co-creation of software applications within digital ecosystems (Jansen & Cusumano, 2012). In a software ecosystem, each organization develops additional applications and services for the end users or other stakeholders in the ecosystem (Bianco et al., 2014; Manikas & Hansen, 2013). This network of participating firms and third-party agents effectively changes the way software systems are created (Jansen et al., 2009). Software vendors depend on internal and external software and service suppliers, value-adding composition, and even the customer itself to co-create (Jansen et al., 2009). Thereafter, the network around a digital platform is a software ecosystem (Bianco et al., 2014; Jansen & Cusumano, 2012; Jansen et al., 2009; Manikas & Hansen, 2013).

Bianco et al. (2014) define the properties of a software ecosystem (Bianco et al., 2014). They describe the self-regulatory characteristics of the ecosystem (Bianco et al., 2014). The keystone firm is controlling the platform and adapts to the network while the network is adapting to platform evolution (Bianco et al., 2014; Hora et al., 2018). In this context, one important aspect of platform evolution is the evolution of the boundary resources, especially the APIs (Hora et al., 2018).

Another important aspect is the shared value (Bianco et al., 2014). Each agent in the network might have a different business model and even compete (Bianco et al., 2014). The shared value which manifests both in software artifacts and the business ecosystem is the motivation for each agent to co-create (Bianco et al., 2014).

Described co-creation, adoption, and evolution demand high levels of extensibility, portability, and variability (Jansen et al., 2009). The integration between software systems requires interfaces (Jansen et al., 2009). In this context, APIs and web APIs play an important role (de Reuver et al., 2018; Gamma et al., 1994; Yoo et al., 2010). They are defined in the following sections.

## 2.2 APIs

The term API was first used in 1968 by Cotton and Greatorex (European Commission. JRC., 2019). They describe requirements for data structures and techniques for computer graphics and define 'Application Program Interfaces' as one building block of software development and refer to hardware abstractions as one possible use case for APIs (Cotton & Greatorex, 1968). Since the concept of APIs is so generic, it can be applied to many different implementations. It is also evolving over time based on technological change. Shnier (1996) refines the API definition and summarizes APIs as "the calls, subroutines, or software interrupts that comprise a documented interface so that an application program can use the services and functions of another application, operating system, network operating system, driver, or other lower-level software program" (Shnier, 1996).

In conclusion, APIs enable software engineers to access utilities through defined interfaces (Basole, 2016). This abstraction of capabilities allows module-based separation of concerns and enables layered architectures, both of which improve loose coupling within the system (de Reuver et al., 2018; Gamma et al., 1994; Yoo et al., 2010). Therefore, APIs can be used as a way to improve reusability of capabilities which leads to improved software productivity and quality (Bonardi et al., 2016; Hou & Yao, 2011; Koci et al., 2019).

APIs have been part of software architectures for decades but new technologies allow new applications for them (Basole, 2016; de Reuver et al., 2018). The development of distributed networks instigate remote APIs (Lübke et al., 2019). Those APIs allow for new integrations and motivated SOA-based, microservice, and cloud-native applications (Lübke et al., 2019). Described developments increase the importance of APIs in the industry but also for the IS research and platform literature (Basole, 2016; de Reuver et al., 2018; Yoo et al., 2010).

## 2.3 Web APIs

Today's cloud-native applications and microservices offer web-based remote interfaces (Haupt et al., 2018; Lübke et al., 2019; Papazoglou & van den Heuvel, 2007). Those APIs are commonly referred to as web APIs (European Commission. JRC., 2019; Lübke et al., 2019; Papazoglou & van den Heuvel, 2007). The European Commission published an extensive technical report in 2019 that aims to suggest general purpose standards and definitions for web APIs (European Commission. JRC., 2019). They refer to Definition.net[1] to define web APIs as follows: "A Web API is a source code interface that a computer system uses to support requests for services to be made by a computer program. Web APIs deliver your request to the service provider, and then deliver the response back to you" (European Commission. JRC., 2019). It follows that web APIs operate on the network layer of the layered modular architecture (Yoo et al., 2010). The literature differentiates between public, partner, and private APIs to define the architectural openness of the API endpoints. Public or external APIs are offered to third-parties and made available over the web (Bondel et al., 2020; European Commission. JRC., 2019). Partner APIs are offered solely to partnering organizations in a Business to Business (B2B) relationship (De, 2017, p. 6). Private or internal APIs are consumed within the own organizational boundaries (Bondel et al., 2020; De, 2017; European Commission. JRC., 2019).

Web APIs play a key role in the SOA and state-of-the-art product and service architectures (Basole, 2016; de Reuver et al., 2018; Krintz & Wolski, 2013; Lübke et al., 2019; Tan et al., 2016; Yoo et al., 2010). Since web APIs are accessed remotely, they enable new network-based business models (Bondel et al., 2020). Those network-based environments are distributed systems that use web APIs as interfaces for communication (Espinha et al., 2014; Lübke et al., 2019). As mentioned in section Boundary Resources, boundary resources like web APIs evolve and adapt based on changes in the system (Espinha et al., 2014; Lübke et al., 2019).

On a technical level, those changes are the implementation of new features, the fixing of bugs in the system, and the discontinuation of features (Lübke et al., 2019). On a higher level, API providers have to manage web APIs based on various, competing issues (Lübke et al., 2019). Web APIs act as an abstraction layer for the assets life-cycle and protect them from technological changes (Krintz & Wolski, 2013). Additionally, web APIs offer scalability through the utilization of standardization (Krintz & Wolski, 2013).

The standardization of web APIs is enabled through the usage of common web protocols. Web APIs use public internet and web technologies like the Hypertext Transfer Protocol (HTTP) to transport information between provider and consumer (Bondel et al., 2020; European Commission. JRC., 2019; Fielding, 2000). On top of the layered protocols of the World-Wide Web, architectural styles and protocols have been developed to standardize the communication of web APIs (European Commission. JRC., 2019; Fielding, 2000). Early SOA approaches utilize

[1]https://www.definition.net/define/api

the Simple Object Access Protocol (SOAP) and XML (De, 2017, p. 12). Recently, GraphQL has gained popularity as a distributed data query language for client-server communication[2]. The most popular architectural style for resource sharing is the Representational State Transfer (REST) (European Commission. JRC., 2019).

## REST

REST was developed by Fielding in 2000 and has gained popularity among software engineers (European Commission. JRC., 2019). It utilizes HTTP and a client-server architecture to define constraints and architectural elements (Fielding, 2000). APIs that follow the REST style guide are also called RESTful APIs (European Commission. JRC., 2019; Fielding, 2000).

The client-server architecture describes an architectural approach to manage centralized capability control and accessibility (Papazoglou, 2008, p. 59). Thus, REST offers an architecture to offer the API consumer defined endpoints to access and manipulate resources (Fielding, 2000; Papazoglou, 2008). It operates on the application-level of the web and utilizes native HTTP utilities like request methods, error codes, and headers (Fielding, 2000; Leach et al., 1999). Since REST utilizes HTTP, it can integrate with HTTP-based specifications such as the OpenAPI Specification (OAS)[3].

## Open API Specification

The OAS is a community-driven specification governed by the OpenAPI Initiative[4]. It is a programming language-agnostic description language for APIs that is based on HTTP (OpenAPI Initiative, 2020). The goal of OAS is to provide both machines and humans a standardized language to explore and understand the endpoints of an API (OpenAPI Initiative, 2020). The OpenAPI Initiative (2020) describes use cases of OAS which include: "interactive documentation; code generation for documentation, clients, and servers; and automation of test cases" (OpenAPI Initiative, 2020). Thus, OAS enables the automation and discoverability of capabilities. One commonly used tool set that implements the OAS and offers utilities to both the API provider and consumer is Swagger[5]. Swagger provides tools[6] such as an user interface for visualizing API endpoints and tools to create client SDKs.

[2]https://www.redhat.com/en/topics/api/what-is-graphql
[3]http://spec.openapis.org/oas/v3.0.3
[4]https://www.openapis.org/
[5]https://swagger.io
[6]https://swagger.io/tools/

## 2.4 SDKs

The platform literature defines APIs and SDKs as boundary resources of a platform (de Reuver et al., 2018). Bianco et al. (2014) further define an SDK as a development boundary resource (Bianco et al., 2014). Thus, SDKs offer utilities to developers to create software for a specified target platform (Bianco et al., 2014). These can include documentation, libraries, developer environments, guidelines, APIs, and more[7]. The goal of an SDK is to reduce the complexity of developing applications for the target platform (RapidAPI, 2019). The core of an SDK is usually a set of software libraries or application frameworks (RapidAPI, 2019). Those software artifacts are maintained by the platform or capability provider and offered as software utilities to third-party developers (RapidAPI, 2019). Thereby, SDKs increase productivity and software quality (Hou & Yao, 2011).

A framework is defined as utilities that create a reusable design for a software architecture (Gamma et al., 1994, p. 26). The framework dictates how the application architecture needs to look (Gamma et al., 1994, p. 26). It provides the structure of the application and the developer needs to integrate the business logic (Gamma et al., 1994, p. 26). On the other hand, software libraries are sets of general-purpose utilities (Gamma et al., 1994, p. 26). They are provided as a toolkit that can be integrated into the main body of the application (Gamma et al., 1994, p. 26). Platform providers can offer libraries to support the integration with their target platform (Gamma et al., 1994, p. 26). To summarize, application frameworks and software libraries offer different approaches but share the common goal of providing support to the developer.

The integration of web APIs requires the API consumer to implement the API requests into the consuming application. For instance, in the case of a RESTful API, the API consumer has to implement HTTP requests (Fielding, 2000). These integration can get complicated and require domain knowledge of the underlying platform (Mulloy, 2012, p. 31). The API provider can support third-party developers by creating software libraries that offer client-side, platform-specific code utilities and implement the API request into a set of function calls (Mulloy, 2012, p. 31). Those functions can be invoked within the client application to make API requests (Mulloy, 2012, p. 31). To conclude, software libraries can be used to supplement web APIs and API providers can offer SDKs to support the service consumer in the implementation of web APIs (De, 2017; Islind et al., 2016; Mulloy, 2012).

In the context of web APIs, client-side code utilities can increase the code quality and productivity of the third-party developer (Hou & Yao, 2011). This can further improve the speed of adoption and simplify the integration (Mulloy, 2012, p. 31). As mentioned in section Open API Specification, standardized APIs that follow the OAS can generate client-side code for common programming languages automatically (OpenAPI Initiative, 2020). The API provider can make use of client-side code generation to develop software libraries. Additionally, OAS can help

[7]https://rapidapi.com/blog/api-vs-sdk/

the API consumer to understand the web APIs through tooling and a standardized description language (OpenAPI Initiative, 2020).

SDKs are used to transfer knowledge to distant third-party developers (Bianco et al., 2014; Islind et al., 2016). The knowledge is thereby embedded within software artifacts (Islind et al., 2016). In the following section, the topic of knowledge transfer is reviewed.

## 2.5 Knowledge Transfer

The scientific literature about knowledge transfer formalizes the collaborative aspect of software co-creation. The platform literature emphasizes the importance of boundary resources as the connectors between the platform owner and the developers (Bianco et al., 2014; Islind et al., 2016; Yoo et al., 2010). Social boundary resources in particular are utilized to transfer knowledge to the third-party developers (Bianco et al., 2014; Islind et al., 2016).

Communication is studied by several fields of research that develop different organizational and curricular models (Calhoun, 2011). The area of knowledge transfer is researched from an IS perspective (Boland et al., 1994; Hislop, 2002). Knowledge transfer depends on social and cultural processes of communication (Boisot, 1986; Scarbrough, 1995). In this context, three different types of knowledge communication are distinguished: professionalism, objectification, and organizational sedimentation (Scarbrough, 1995). Professionalism describes the processes of learning and gaining experience in a practicing community (Islind et al., 2016; Scarbrough, 1995). Objectification defines standardization which enables portability of knowledge (Scarbrough, 1995). Organizational sedimentation captures knowledge communication via rules, standards, routines, and structures (Scarbrough, 1995).

Knowledge can also be differentiated from information. One definition from Newman and Newman (1985) states: "Information is the answer to a question. Knowledge is a framework that enables the question to be asked" (Newman & Newman, 1985, p. 499). Therefore, information is easy to duplicate while the ability to transfer knowledge depends upon the prior knowledge a person has (Scarbrough, 1995). Knowledge transfer also does not remove knowledge from the source but is rather shared and has to be re-created at the target (Scarbrough, 1995).

The literature differentiates between tacit and implicit knowledge (Hislop, 2002; Nonaka & Takeuchi, 1995). Explicit knowledge can be represented in a tangible form (Hislop, 2002; Nonaka & Takeuchi, 1995). For instance, it can be represented and transferred through documentation and other social boundary resources. Tacit knowledge is knowledge for physical skills (e.g. riding a bike) and cognitive frameworks (e.g. value systems) (Hislop, 2002; Nonaka & Takeuchi, 1995). Tacit knowledge is possessed by people and cannot be encoded and transferred easily (Hislop, 2002; Nonaka & Takeuchi, 1995). Another form of knowledge that plays an important role for software development is technical knowledge (Islind et al., 2016). It is used to describe expert knowledge required to complete tasks (Islind et al., 2016).

Explicit technical knowledge assets can be shared through IT systems (Boland et al., 1994; Hislop, 2002). Thereby, information technology enables individuals to create rich representations of their knowledge (Boland et al., 1994). The dependency of people on IT system is increasing as the relationship between people and IT systems becomes interwoven (Sørensen & Snis, 2001). However, knowledge transfer via IT systems has limitations (Scarbrough, 1995). The intrinsic properties of knowledge make the transfer difficult (Cook & Brown, 1999; Hislop, 2002).

Tacit and distributed knowledge within software ecosystems can be shared using professionalism (Islind et al., 2016). As mentioned above, professionalism includes the co-creation of knowledge within a practicing community (Islind et al., 2016; Scarbrough, 1995). The knowledge is shared through dialog and interaction among groups of individuals (Islind et al., 2016). Professionalism enables knowledge transfer when standardization and rules required for objectification and organizational sedimentation have not been created (Islind et al., 2016).

Islind et al. (2016) emphasize the importance of non-technical knowledge in smaller platforms but also confirms that the importance of commoditized knowledge increases with the size of the platform (Islind et al., 2016). To transfer technical knowledge, the platform owner has to manage knowledge 'at arms length' (Islind et al., 2016). Social boundary resources in particular enable the communication of explicit knowledge (Islind et al., 2016). Thereby, the knowledge communication can be defined as an economic exchange where the knowledge is objectified in the boundary resources (Islind et al., 2016).

The collaborative aspects of software ecosystems can be connected to the process of distributed cognition. Distributed cognition describes a process in which autonomous individuals interact based on their own situation and react to other individuals' situations (Boland et al., 1994). It describes a model of organizational learning and emphasizes the importance of knowledge for decision making and collaboration (Boland et al., 1994). Distributed cognition can be supported and facilitated by information systems (Boland et al., 1994). Rich representations of knowledge codified in IT systems can be used to communicate understanding (Boland et al., 1994).

The development of software is a social activity that requires knowledge transfer (Bianco et al., 2014). Knowledge transfer is a fundamental part of co-creation (Bianco et al., 2014; Islind et al., 2016). The platform has to implement a knowledge transfer strategy (Islind et al., 2016). The distribution of knowledge requires the support of information systems (Islind et al., 2016). This especially holds true if knowledge has to be distributed between distant third-party developers (Islind et al., 2016). The distribution of knowledge within a software ecosystem can be linked to the process of distributed cognition.

## 2.6 SOA

Service-oriented concepts are intensively investigated by IS research (Demirkan et al., 2008). Studied concepts include SOAs, service science, and service-oriented computing (Demirkan et al., 2008; Papazoglou & Georgakopoulos, 2003). Since SOA is a mature concept, its implementation approaches have changed over time (Amaravadi, 2014; Tan et al., 2016; Zimmermann, 2017). Today's definition of SOA describe it as software architecture that isolates services into independent and reusable remote components that are discoverable and accessible without requiring knowledge of the implementation details (Amaravadi, 2014; Demirkan et al., 2008; Löhe & Legner, 2010a; Papazoglou & van den Heuvel, 2007). It utilizes web technologies and open standards to enable ubiquitous access within a distributed system of services (Löhe & Legner, 2010a; Papazoglou & Georgakopoulos, 2003; Papazoglou & van den Heuvel, 2007).

Papazoglou and van den Heuvel (2007) define roles within SOA (Papazoglou & van den Heuvel, 2007). The service provider and requester communicate through digital service requests (Papazoglou & van den Heuvel, 2007). Additionally, they introduce a service aggregator or broker that manages service discovery and distribution between service providers and requesters (Papazoglou & van den Heuvel, 2007). This new role of service brokerage adds a layer of convenience for the service requester (Papazoglou & van den Heuvel, 2007). The Information Technology Infrastructure Library (ITIL) (2019) emphasizes the service relationship between the provider and consumer which is described as joint activities that have to be managed by both sides (Limited & Office, 2019, p. 15). The service provision includes tasks such as the management of provided resources, access management to the resources, service fulfillment, and continual development (Limited & Office, 2019, p. 15). The quality and reliability of the service relationship can be standardized through Service Level Agreements (SLAs) (Ranabahu et al., 2009, p. 2). SLAs document the expected level of service that is provided by the service provider to the service consumer (Ranabahu et al., 2009, p. 2).

The management of remote interfaces between the service provider and consumer is of growing importance (de Reuver et al., 2018; Lübke et al., 2019). In practice, many companies have successfully integrated service-oriented thinking (Demirkan et al., 2008). Luthria and Rabhi (2009) conclude that SOA supports the interoperability of backend services within an organization (Luthria & Rabhi, 2009). The reuse of capabilities within SOA can reduce costs and enable new value composition (Demirkan et al., 2008). SOA enables digital business networks and inter-organization integrations where each organization provides and consumes internal and external services (Löhe & Legner, 2010a; Papazoglou & van den Heuvel, 2007). As a result, SOA is also researched from an organizational and management point of view (Löhe & Legner, 2010a; Luthria & Rabhi, 2009).

SOA implementation approaches have changed over time (Tan et al., 2016; Zimmermann, 2017). This is caused by technological advancements that enable new architecture approaches (Zimmermann, 2017). Since SOA utilizes (web) application servers, most used remote interfaces are web APIs (Lübke et al., 2019; Papazoglou & van den Heuvel, 2007). Early SOA approaches are based on SOAP and XML (De, 2017, p. 12). Today's SOA employs state-of-the-art technologies such as REST to enable new service economies (Tan et al., 2016). One paradigm that is changing SOA is cloud computing (Limited & Office, 2019; Pahl et al., 2017; Ranabahu et al., 2009; Zimmermann, 2017, p. 29). The scientific literature about cloud computing is reviewed in the following section.

## 2.7 Cloud Computing

Pallis (2010) describes cloud computing as a model that offers easy and on-demand access to computing resources as a service (Pallis, 2010). Its utilization reduces operation costs and allows for an economy of scale (Roberts & Chapin, 2017, p. 2-3). Common computing resources that are offered include infrastructure, platforms, and applications (Amaravadi, 2014; Pahl et al., 2017; Ranabahu et al., 2009). Infrastructure as a Service (IaaS) cloud offerings consist of storage and network services (Amaravadi, 2014; Roberts & Chapin, 2017; Tan et al., 2016). Platform as a Service (PaaS) offers development environments for hosting applications (Amaravadi, 2014; Roberts & Chapin, 2017). It is a cheap and convenient way for firms to deploy applications (Amaravadi, 2014; Roberts & Chapin, 2017). Software as a Service (SaaS) provides software services on demand (Amaravadi, 2014).

Applications deployed to PaaS are called cloud-native applications (Lübke et al., 2019). They can be used in software intensive systems and offer web APIs for data exchange within the distributed network (European Commission. JRC., 2019; Lübke et al., 2019). They enable decomposition and loose coupling (European Commission. JRC., 2019; Karmel et al., 2016). In this context, the term microservice describes the decoupled service component and cloud-native application servers are used to implement microservices (Karmel et al., 2016; Papazoglou & van den Heuvel, 2007). The National Institute of Standards and Technology (2016) defines microservices as "[...] a basic element that results from the architectural decomposition of an application's components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol and a set of well-defined APIs, independent of any vendor, product or technology" (Karmel et al., 2016, p. 2). Microservices are both argued to be a new architectural style and an implementation approach to SOA (Zimmermann, 2017). From a SOA perspective, cloud computing enables a modern SOA approach (Pahl et al., 2017; Zimmermann, 2017).

Microservices that are exposed to external parties are also called Backend as a Service (BaaS) (European Commission. JRC., 2019; Roberts & Chapin, 2017). BaaS is related to SaaS but focuses on the idea of breaking up applications into a composition of services (Roberts & Chapin, 2017, p. 6). It enables the outsourcing of services (Roberts & Chapin, 2017, p. 6). In the following, the related term web service is defined.

## 2.8 Web Service

In the literature, the term web service is used in conjunction with related concepts such as SOA, microservices, and web APIs (European Commission. JRC., 2019; Fokaefs et al., 2011; Karmel et al., 2016; Löhe & Legner, 2010a). A digital services is defined as a reusable activity that has an idempotent outcome[8] (IBM Developer Staff, 2018). However, this definition also holds true for RESTful APIs (Fielding, 2000; IBM Developer Staff, 2018). The technical report of the European Commission (2019) derives the following difference: "Web Services and APIs differ at the design level but not at the technological level" (European Commission. JRC., 2019, p. 6). Thus, web services and web APIs are complementary concepts (IBM Developer Staff, 2018). They differ at the targeted abstractions level (IBM Developer Staff, 2018). APIs provide low-level functionalities while web services define interfaces for higher-level capabilities (IBM Developer Staff, 2018).

Papazoglou (2008) defines a web service as a: "[...] self-describing, self-contained software module available via a network, such as the Internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or application" (Papazoglou, 2008, p. 5). Thereafter, a web service is a concept that enables decomposition, loose coupling, and shares similarities with microservices (Hillpot, 2020). The W3C offers a similar definition and emphasizes the inter-operable, machine-to-machine communication via web-related standards like HTTP[9]. In this context, a web service includes a web API and can be described as a wrapper around a digital service or application to offer capabilities over the web (Hillpot, 2020). In contrast, as described in section Cloud Computing, a microservice is meant to break down a software system in independent services accessible in a distributed system (Hillpot, 2020). Within a software ecosystem, a web service can provide its capabilities to other web services, microservices, or to the end user directly (Hillpot, 2020).

A web service can provide one business object, a business task, a business process, or an entire application via web APIs (Papazoglou, 2008, p. 5). Similarly, Löhe and Legner (2010) use the term service granularity to categorize the offered services within a SOA. They differentiate between business processes, activities and tasks, and utilities and entities as three granularity levels (Löhe & Legner, 2010a, 2010b). In this categorization, business processes are defined as entire workflows, activities and tasks describe single process steps, and utilities and entities equal generic infrastructure functionalities (Löhe & Legner, 2010a, 2010b). Both categorizations emphasize the multitude of applications for web-based APIs.

[8]https://developer.ibm.com/devpractices/api/articles/api-vs-services-whats-the-difference/
[9]https://www.w3.org/TR/ws-arch/#whatis

## 2.9 API Management

Mathijssen et al. (2020) conduct a systematic literature review over the topic API management (Mathijssen et al., 2020). They conclude that the scientific literature around API management is rather sparse (Mathijssen et al., 2020). The most commonly cited definition of the term API management comes from De (2017) (De, 2017; Mathijssen et al., 2020). De (2017) defines API management as a platform and states: "An API management platform helps an organization publish APIs to internal, partner, and external developers to unlock the unique potential of their assets. It provides the core capabilities to ensure a successful API program through developer engagement, business insights, analytics, security, and protection." (De, 2017, p. 15). Additionally, Mathijssen et al. (2020) themselves argue that API management is required in order to perform tasks including maintaining documentation, controlling and monitoring access, and carrying out analytics of the API usage (Mathijssen et al., 2020). Furthermore, API management platforms support activities for internal, partner, and external API consumers (Mathijssen et al., 2020).

On its website, RedHat documents goals of API management[10]. They describe API management as a support function that enables the organization to create and use APIs (Red Hat, Inc., 2021). It centralizes core capabilities to manage and fulfill requirements of developers and applications which enables API consumption in a compliant and secure manner (Red Hat, Inc., 2021). Mathijssen et al. (2020) list the following capabilities offered by API management as mentioned in the literature:

- API Publication & Deployment
- Analytics
- Authentication
- Catalog & Documentation
- Monetization
- Monitoring
- Security
- Version Management

New APIs have to be deployed and published before they can be provided to the API consumer (De, 2017, p. 27). Deployment describes the process of pushing code changes to a production environment (De, 2017, p. 27). Publication is the process of creating and publishing marketing and documentation material about the API offerings (De, 2017, p. 27). The API management is responsible for managing both the publication and deployment of APIs (De, 2017, p. 16).

Analytics is required to answer key questions of the API management (Red Hat, Inc., 2021). For instance, analytics provides insights about the volume of requests

[10]https://www.redhat.com/en/topics/api/what-is-api-management

for each API (Red Hat, Inc., 2021). It is closely connected to monitoring which collects data about Key Performance Indicators (KPIs) such as request counts, fail rates, and reason of failure within the system (Red Hat, Inc., 2021).

APIs introduce new security threats to the providing organizations (De, 2017, p. 112). Public APIs enable access to internal capabilities to third-parties. Authentication and other security related topics have to be managed (De, 2017, p. 112). Authentication determines the identity of the API consumer and validates access to protected source (De, 2017, p. 113).

API offerings have to be discoverable and understandable (De, 2017, p. 25). The API management is responsible for the knowledge transfer to the developers (De, 2017, p. 59). Documentation is used to communicate the functionality and usage of the APIs (De, 2017, p. 59). API catalogs are used to make API offerings discoverable (De, 2017, p. 25).

API providers directly or indirectly monetize the consumption of the resources exposed through web APIs (Bondel et al., 2020). The monetization is based on a business model and includes the management of the monetization strategy, usage contracts, pricing, billing, and related tasks (De, 2017; Red Hat, Inc., 2021). Common monetization models include: one-time fees, pay-per-API transaction, and tiered pricing (De, 2017, p. 146-148). Alternatively, indirect pricing strategies can be utilized that include mutual benefits for the API provider and consumer (De, 2017, p. 146).

Version management is an important task within the API evolution and lifecycle management (Lübke et al., 2019; Red Hat, Inc., 2021). It implements a change strategy and focuses on the communication and technical implementation of changing interfaces (Lübke et al., 2019).

API management utilizes state-of-the-art API management platforms to automate and centralize listed capabilities (De, 2017; Red Hat, Inc., 2021). As visible in figure 2.1, the API management platform organizes services in a hierarchical order (De, 2017, p. 17). The API gateway builds the core platform of the API management (De, 2017, p. 16). It provides core utilities utilized by other platform layers (De, 2017, p. 16). In the following, the services of the API gateway will be described in detail.

### API Gateway

The API gateway is a reverse proxy for API clients (Red Hat, Inc., 2021). Thus, it intercepts incoming requests, retrieves the requested resources from the backend services, and returns them to the client (Red Hat, Inc., 2021). As such, it enables the centralization of management tasks as it is the first point-of-contact for all clients (De, 2017; Red Hat, Inc., 2021). The management tasks achieved with an API gateway include authentication, rate limiting, statistics and analytics, monitoring, policies, alerts, and security[11]. De (2017) summarizes mentioned tasks

[11]https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do

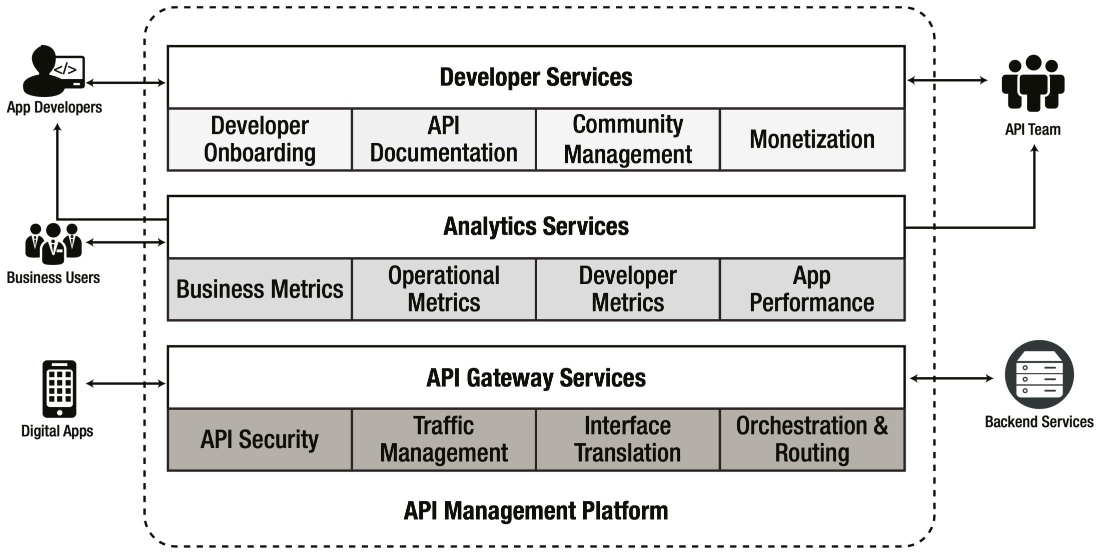


Figure 2.1: API Management Platform Hierarchy from De (2017) (De, 2017, p. 17)

as API security, traffic management, interface translation, and orchestration and routing (De, 2017, p. 17).

Analytics and developer services build on top of the API gateway (De, 2017, p. 16-17). Analytics services utilize the data collected by the API gateway (De, 2017, p. 16). In order for API providers to communicate with API consumers, publish new backend services, and for the application developer to register as a consumer, a developer portal is used (De, 2017, p. 25).

## Developer Portal

The developer portal is the second fundamental part for API management (Red Hat, Inc., 2021). In contrast to the API gateway, which is a infrastructure platform and part of the API runtime, the developer portal serves as a web client for the API gateway and is used by both API provider and API consumer (De, 2017; Red Hat, Inc., 2021). The API provider uses the developer portal to publish and communicate information with the API consumer (De, 2017, p. 25). Commonly, the API documentation, terms and conditions, contact information, marketing information, changelogs, status updates, and social content like forums or blogs are hosted on the developer portal (De, 2017, p. 25).

The developer portal is provided to the API consumer as a tool to manage and support the API integration (De, 2017, p. 26). The API consumer can access the API documentation and follow the API client on-boarding process (De, 2017, p. 26). State-of-the-art developer portals enable self-service and transparent pricing for the API consumer (De, 2017, p. 26). Thus, the developers are able to register themselves as API consumers, create so-called client applications within the developer portal, and automatically generate API keys (De, 2017, p. 26). API keys are utilized to authorize applications to use the APIs (De, 2017, p. 26). They identify and track the applications within the API gateway and allow API call-based monetization strategies (De, 2017, p. 26). When an API key is generated through the developer portal, the associated access rights are communicated to the API gateway which enables the API consumption (De, 2017, p. 26).

### Documentation

Bianco et al. (2014) categorize documentation as a social boundary resource since it contains intellectual property that supports the third-party developer in the integration of the technical boundary resources (Bianco et al., 2014). As mentioned in section SDKs, documentation can accompany or be part of an SDK and document software artifacts provided to the developers (Mulloy, 2012; RapidAPI, 2019). In addition, a web API itself also requires specifications (De, 2017; Koci et al., 2019). As mentioned in section Open API Specification, parts of the API documentation can be standardized using open specifications like the OAS (De, 2017; OpenAPI Initiative, 2020).

De (2017) states that the API documentation of the developer portal should be the single source of truth (De, 2017, p. 172). The technical specification of a web API can be accompanied by FAQs, tutorials, code, and usage examples (De, 2017, p. 176). De (2017) stresses the importance of getting-started and how-to guides that support the on-boarding of the API consumer (De, 2017, p. 176). Furthermore, real-world examples help illustrating the use cases from an API consumer perspective (De, 2017, p. 176).

### API Governance

API governance provides further capabilities to the API program (De, 2017; Krintz & Wolski, 2013). API governance defines guidelines, policies, standards, processes, and best practices for all API providers within the organization (De, 2017; Krintz & Wolski, 2013). Thus, the API governance acts as a central authority and provides quality assurance over the API life-cycle and encompasses the API runtime (De, 2017; Krintz & Wolski, 2013). Krintz and Wolski (2013) argue that there currently do not exist commercial software systems that cover most aspects of API governance (Krintz & Wolski, 2013).

## 2.10 API Economy

The emergence of the API Economy is based on technological changes that is researched in several fields of study. In the following, the term API Economy is used to connect the foundations reviewed in this chapter. Tan et al. (2016) describe how SOA for business information systems moved APIs in the focus of IS research and the industry (Tan et al., 2016). Section Web Service summarizes the development of web services based on web APIs that further instantiate SOA and service-orientation (Zimmermann, 2017). Web APIs also play an important role in new product and service architectures (Yoo et al., 2010). The advancements in cloud computing and the creation of microservice architectures further amplify the importance of APIs (Lübke et al., 2019). The possibility to easily deploy applications to the cloud and outsource services instigates new business models (Bondel et al., 2020; Roberts & Chapin, 2017). Combined with the rise of digital platforms, these changes induce software ecosystems where value is co-created in a network of firms and individual agents (Jansen & Cusumano, 2012; Jansen et al., 2009). All those advancements combine to create the API Economy (Basole, 2016; Bondel et al., 2020; Eaton et al., 2015).

The API Economy is described as a service ecosystem and captures the growth in the number of publicly available web APIs (Basole, 2016; Bondel et al., 2020). ProgrammableWeb is the most extensive public API directory (Basole, 2016; Bondel et al., 2020). It currently registers close to 24,000 publicly available APIs[12]. In the API Economy, value is co-created by a composition of different services (Eaton et al., 2015). A provided service enables the realization of new applications and services and thereafter new business models (Bondel et al., 2020). An application or service that consumes a set of services to create value for end users is also called a mashup (European Commission. JRC., 2019; Fichter, 2006; Fichter & Wisniewski, 2009; Maximilien et al., 2008).

The API Economy illustrates the rising importance of APIs for industry and research. The reviewed foundations add to the knowledge base of this thesis. In the following, the scientific literature related to this study is presented.

[12]https://www.programmableweb.com/apis/directory

# 3 Related Work

In the following, related literature will be summarized. This chapter emphasizes the current research in the areas reviewed in chapter Foundations. Research gaps and open challenges are highlighted. Additionally, the pattern literature is reviewed to present studies that follow similar research approaches.

**Platform literature** stresses the importance of boundary resources management and calls for research around the evolution and tuning of boundary resources (de Reuver et al., 2018; Eaton et al., 2015; Henfridsson & Bygstad, 2013; Yoo et al., 2010).

This study follows the research agenda from Yoo et al. (2010) and de Reuver et al. (2017). Yoo et al. (2010) develop a new product architecture, the layered modular architecture (Yoo et al., 2010). Furthermore, they document a theoretical framework and changing factors for organizations and argue that IS research needs to address these changing factors (Yoo et al., 2010). They propose new research questions that have to be investigated and specifically emphasize boundary resources for future research (Yoo et al., 2010).

De Reuver et al. (2017) review the platform literature and analyze the rising complexity within distributed digital platform architectures across different industries (de Reuver et al., 2018). They develop a research agenda for IS resarch to address new research challenges (de Reuver et al., 2018). For instance, they argue that the effect of long term decisions cannot be predicted easily and state that studies on the evolution of digital platforms and ecosystems are therefore required (de Reuver et al., 2018).

In the following, further calls for research within the platform literature are listed. Henfridsson and Bygstad (2013) highlight that boundary resources should be the unit of the analysis since they facilitate the relationship between the platform's stakeholders (Henfridsson & Bygstad, 2013). Similarly, Eaton et al. (2015) argue that the central role of boundary resources requires research about the creation, maintenance, and evolution to research the innovation that takes place in digital service systems (Eaton et al., 2015).

Additional work that influences this study comes from Islind et al. (2016) and Bianco et al. (2014). Islind et al. (2016) research the creation and fine tuning of boundary resources and knowledge communication in smaller platforms based on action research (Islind et al., 2016). They stress the importance of knowledge transfer and differentiate different ways of knowledge communication between the platform owner and third-party application developers (Islind et al., 2016). Bianco et al. (2014) categorize boundary resources and analyze how boundary resources should be built. They identify three different types of boundary resources by dividing technical boundary resources into developer and application boundary resources (Bianco et al., 2014).

**Software ecosystems** change the way software is developed and instigate new challenges that have to be addressed by the IS research (Jansen et al., 2009).

Jansen et al. (2009) state that co-creation changes the way software is developed. They argue that today's networked software ecosystems introduce new research challenges for both technical and business areas of research (Jansen et al., 2009). Software ecosystems introduce new challenges for business models, the management of platform control, and others (Jansen et al., 2009). They stress that the research community should investigate the changing factors (Jansen et al., 2009).

**Service Orientation** is a well-researched field that offers many similarities to API management (De, 2017, p. 12).

This study utilizes influence factors within SOA implementations developed by Löhe and Legner. Löhe and Legner (2010) review the current state of the SOA literature and present a framework for the empirical analysis of influence factors in real-world SOA (Löhe & Legner, 2010a, 2010b). For this, they analyze 33 cases of SOA implementations and develop matrices of attributes and attribute values which create insights about the context of the analyzed cases (Löhe & Legner, 2010a, 2010b).

**API Economy** is a fairly new term that aims to capture the emergence of new service ecosystems (Tan et al., 2016).

Tan et al. (2016) describe that the SOA emerges into an API Economy (Tan et al., 2016). They explain the emergence of the API Economy based on technological change and case studies (Tan et al., 2016). Bondel et al. (2020) argue that there has been no research answering the question why the API Economy is advancing with different pace in different sectors (Bondel et al., 2020). Therefore, they research barriers preventing the advancement of the API Economy within the automotive industry (Bondel et al., 2020).

**API & API management literature** focuses on an API consumer perspective and lacks standards for API management (Koci et al., 2019; Mathijssen et al., 2020).

In the following, the related literature around APIs and API management is listed. Sohan et al. (2015) research versioning of documentation and its communication based on a case study of releases of popular web APIs (Sohan et al., 2015). They identify six API change patterns. Overall, they discuss different approaches and argue for a lack of standards (Sohan et al., 2015). Haupt et al. (2015) present tools and processes to conduct technical API governance based on REST API designs (Haupt et al., 2018). Hou et al. (2011) research the intent behind API change within a software library (Hou & Yao, 2011). Similarly, Jezek and Dietrich (2017) compare tools that aim to support software library API evolution (Jezek & Dietrich, 2017).

The management of APIs is a highly requested field of research (Mathijssen et al., 2020). The literature identifies several research gaps. Koci et al. (2019) explain that the focus of research is currently on the API consumer and that the API management from a provider perspective lacks attention (Koci et al., 2019). Mathijssen et al. (2020) state that API management research is sparse and that more best practices have to be identified (Mathijssen et al., 2020). Similarly, Sohan et al. (2015) stress a lack of conventions and present that most API changes are not documented and communicated to the API consumers (Sohan et al., 2015).

To conclude, several research gaps and calls for research have been identified within the related literature. Changing factors of digitization require new ways for companies to collaborate and transfer knowledge. New product and service-oriented architectures instigate new service economies. To the best knowledge, the scientific literature of API management is lacking standardization and formulation of best practices. This study aims to address the API provider perspective of the API provision management and collect best practices and solution approaches to common concerns by developing a pattern catalog. In following, relations to the pattern literature are presented.

**Relations to other pattern languages** are reviewed to distinguish this study from past work but also to review the scientific foundations on which this study develops its own pattern language.

The following pattern languages are foremost utilized as foundations for the development of the pattern language. Gamma et al. (1994) first introduce design patterns to the field of software engineering (Buckl et al., 2013; Gamma et al., 1994). They present a list of patterns to create reusable objects in object-oriented programming (Gamma et al., 1994). Coplien (1994) offers a pattern family of organizational development processes (Coplien, 1994). Brown et al.(1998) provide a definition for patterns and anti-patterns and promote a set of anti-patterns for software engineering (Brown et al., 1998).

Lübke and Zimmermann offer a series of papers about API and microservice design (Lübke et al., 2019; Pautasso et al., 2017; Zimmermann, 2017; Zimmermann et al., 2020; Zimmermann et al., 2017). They also maintain a website[1] that promotes their findings and current research. Lübke et al. (2019) discuss a wide set of concerns that the API provider has to manage and offer a pattern catalog of technical API management patterns to address those concerns (Lübke et al., 2019). Zimmermann et al. (2020) present further technical microservice and API patterns targeting API endpoint design (Zimmermann et al., 2020). Zimmermann et al. (2017) provide a pattern catalog of interface representation patterns that document solution approaches for API design (Zimmermann et al., 2017). They also offer an extensive overview of pattern languages that relate to the topic of API design and pattern languages that document patterns of service-orientation and web services (Zimmermann et al., 2017). In the following, those foundations are reviewed.

Fowler (2002) offers a pattern catalog for enterprise applications and also denote remote API design aspects (Fowler, 2002; Zimmermann et al., 2017). Voelter et al. (2004) document a pattern language for middlewares in distributed systems (Voelter et al., 2004; Zimmermann et al., 2017). Buschmann et al. (2007) com-

[1]https://microservice-api-patterns.org/

bine several pattern languages together to provide one source for distributed system patterns (Buschmann et al., 2007b; Zimmermann et al., 2017). Rotem-Gal-Oz (2012) researches SOA infrastructure and platforms and provides patterns and anti-patterns for architectural guidance of SOA (Rotem-Gal-Oz, 2012; Zimmermann et al., 2017).

With regards to service-orientation, the following pattern languages are reviewed by Zimmermann et al. (2017) (Zimmermann et al., 2017). Daigenau (2011) documents service design patterns (Daigneau, 2011; Zimmermann et al., 2017). The patterns thereby focus on a SOAP protocol or REST paradigm level level(Daigneau, 2011; Zimmermann et al., 2017). Similarly, Pautasso et al. (2016) offer a pattern catalog about RESTful communication patterns (Pautasso et al., 2016; Zimmermann et al., 2017). On a process level of SOA, Hentrich and Zdun (2011) provide a pattern catalog about process and workflow orchestration (Hentrich & Zdun, 2011; Zimmermann et al., 2017).

Following pattern languages focus on related fields of management and collaboration. Buckl et al. (2008) present an enterprise architecture management catalog (Buckl et al., 2008). Khosroshahi et al. (2015) re-iterate on the catalog from Buckl et al. and publish version 2.0 in 2015 (Khosroshahi et al., 2015). Uludağ et al. (2019) follow the pattern-based design research method by Buckl et al. (2013) and document patterns about large-scale agile development (Buckl et al., 2013; Uludağ et al., 2019). Furthermore, they provide an extensive overview of related pattern languages in their field of research (Uludağ et al., 2019).

Listed pattern languages are found to be complimentary with the API provider perspective on API management that is conducted in this study. Identified pattern languages from Lübke and Zimmermann focus on technical aspects of API management while management-oriented pattern languages such as the enterprise architecture management catalog from Buckl and Khosroshahi and the large-scale agile development pattern catalog from Uludağ et al. (2019) research related fields of management and collaboration.

The API management literature is rather spare (Mathijssen et al., 2020). De (2017) offers a catalog of patterns for API management, security, deployment, and adoption (De, 2017, p. 86). The patterns illustrate the most common practices within the technical aspects of API management and management of API platforms (De, 2017, p. 86). This thesis focuses on the relationships between different roles, teams, and stakeholders of API management and their communication, collaboration, and knowledge transfer. One pattern documented by De (2017) overlaps with a solution approach identified in this study. De (2017) documents API Facade Patterns such as API Composition (De, 2017, p. 86-88). The creation of an orchestration layer is also documented in this thesis. In general, the best practices described by De (2017) can be regarded as more general and complimentary to this thesis.

# 4 Research Approach

As outlined in section Objectives, this thesis aims to create a pattern catalog to capture best practices and solution approaches to common problems within the API provision management. In the following, the detailed research approach is laid out.

The IS field utilizes a multitude of paradigms, methods, and research approaches (Buckl et al., 2013; Hevner et al., 2004; Urquhart et al., 2009). Popular approaches come from behavioral science and design science fields (Buckl et al., 2013; Hevner et al., 2004). This thesis follows a design science framework derived from Hevner et al. (2004) (Hevner et al., 2004). Their method describes an approach to create novel artifacts to solve specific problems (Buckl et al., 2013; Hevner & Chatterjee, 2010; Hevner et al., 2004). The framework is based on rigor and relevance (Buckl et al., 2013; Hevner & Chatterjee, 2010; Hevner et al., 2004). Figure 4.1 illustrates how the research addresses this criteria.

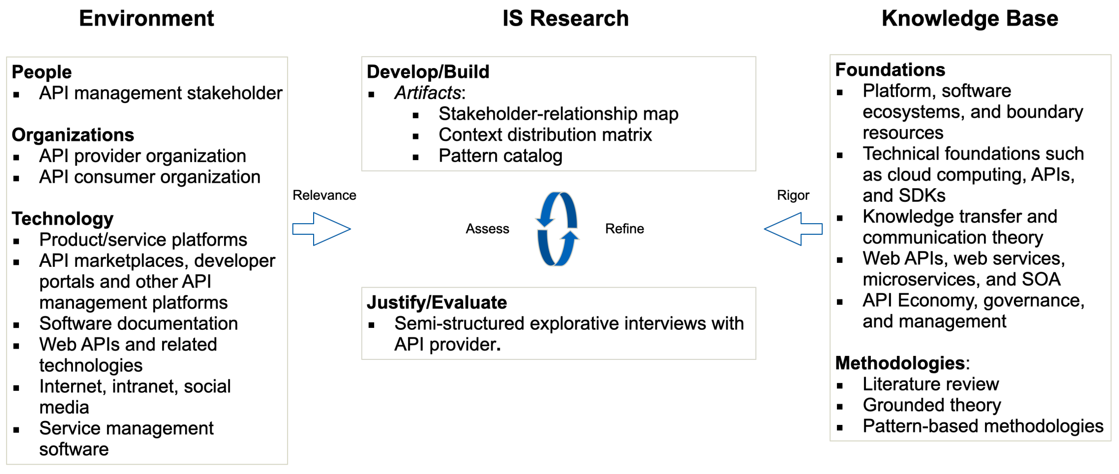

Environment
People
API management stakeholder
Organizations
API provider organization
API consumer organization
Technology
Product/service platforms
API marketplaces, developer portals and other API management platforms
Software documentation
Web APIs and related technologies
Internet, intranet, social media
Service management software
Relevance
IS Research
Develop/Build
Artifacts:
Stakeholder-relationship map
Context distribution matrix
Pattern catalog
Assess
Refine
Justify/Evaluate
Semi-structured explorative interviews with API provider.
Rigor
Knowledge Base
Foundations
Platform, software ecosystems, and boundary resources
Technical foundations such as cloud computing, APIs, and SDKs
Knowledge transfer and communication theory
Web APIs, web services, microservices, and SOA
API Economy, governance, and management
Methodologies:
Literature review
Grounded theory
Pattern-based methodologies

Figure 4.1: Research Framework following Hevner et al. (2004)

Rigor is achieved by using sound research methods (Buckl et al., 2013; Hevner et al., 2004). As shown in figure 4.1, we utilize three methodologies within the larger framework. First, a literature review is used to build upon a knowledge base. The goal of the literature review is to describe terms and theory that will be drawn upon (Strauss & Corbin, 1998). Furthermore, it is used to motivate the relevancy of the study and stress the knowledge gap within the API management field that we aim to address (Strauss & Corbin, 1998). This thesis follows a concept-centric review approach as described by Webster et al. (2002) (Webster & Watson, 2002). An initial keyword search in common databases creates a starting point for going backwards and forwards through citations (Webster & Watson, 2002). Since IS is an interdisciplinary field, the literature review includes work from related fields such as software engineering and management (Webster & Watson, 2002). The content of the literature review is laid out in the chapters Introduction, Foundations, and Related Work. The most important concepts for the literature review are listed in figure 4.1 and include research streams that influence the API management field.

Next, a grounded theory approach[1], as first described by Glaser and Strauss (1967), is embedded within the overall framework of this study (Glaser & Strauss, 1967; Urquhart et al., 2009). As discussed by Wiesche et al. (2017), the grounded theory methodology can be used for more than novel theory development (Wiesche et al., 2017). In the IS field, it is a common behavioral science approach used to support data collection and to create rich descriptions or models (Urquhart et al., 2009; Wiesche et al., 2017). We conducted semi-structured interviews with API providers to gather data about impediments, context, and best practices for API management from an API provider perspective. The interview data is used to justify and evaluate the novel artifacts in an iterative process (Hevner & Chatterjee, 2010; Hevner et al., 2004). Furthermore, Buckl et al. (2013) note that using practitioners can achieve relevance (Buckl et al., 2013). The encoding of the interviews follows guidance from Chametzky (2016) (Chametzky, 2016). The codes were iteratively refined whenever a concept gained importance during interview analysis (Bianco et al., 2014). Through constant comparison, the encoded interview data and analysis results lead to the initial pattern candidates (Charmaz, 2014; Urquhart et al., 2009).

To craft novel artifacts, this thesis draws aspects from pattern-based research and from the pattern-based design research method recommended by Buckl et al. (2013) (Buckl et al., 2013; Gamma et al., 1994). They build upon design science to balance rigor and relevance by utilizing patterns (Buckl et al., 2013). In this context, a pattern is as a medium to create design science artifacts (Buckl et al., 2013). The overall goal of these artifacts is to address organizational challenges (Hevner et al., 2004). To accomplish this, the artifacts must be documented efficiently to enable implementation and usage (Hevner et al., 2004). As argued by Buckl et al. (2013), a pattern catalog can fulfill those requirements (Buckl et al., 2013).

We differentiate pattern candidates and patterns. Each pattern candidate follows the pattern definition as defined in section Objectives. Hereby, a pattern is a doc-

[1]http://www.groundedtheory.com/

umented solution for common concerns based on a particular context (Buckl et al., 2008; Gamma et al., 1994). It holds a problem description, documents the captured environment, and illustrates a solution approach as discovered in known uses (Buckl et al., 2013; Lübke et al., 2019; Uludağ et al., 2019). Pattern candidates are added to the list of final patterns if they fulfill the rule of three known uses as established by Coplien (1994) (Buckl et al., 2008; Coplien, 1994). Those patterns can be seen as design principles and thus, building blocks for a design theory (Buckl et al., 2013). The objective of the pattern catalog is to create a knowledge base of best practices for stakeholders of API management.

To conclude, our grounded theories approach is used to create the interview guidelines, conduct, transcribe, and encode the interviews and conceptualize the pattern candidates while the pattern-based research method provides a justification to create a pattern-catalog to conduct design-science research (Buckl et al., 2013; Uludağ et al., 2019). This thesis applies sound methodologies to achieve rigor. Relevance is achieved by following real-world concerns as discovered in collaboration with API provider stakeholders within the explorative interviews and by following the research gaps as motivated in chapter Related Work.

# 5 API Management Pattern Catalog

This chapter presents a pattern catalog that is used to answer the three RQs of this thesis. First, the data collection process is documented. Second, the pattern language that is used to create the pattern catalog is specified. Next, each element of the pattern language is instantiated. The identified stakeholders are described in section Roles and Stakeholders. Section Influence Factors documents context attributes that act as influence factors for the pattern language. All identified concerns are categorized and listed in section Concerns. In section Taxonomy, an overview of the pattern catalog is given that connects stakeholders, concerns, and solution approaches. Section API Management Patterns lists all documented patterns and pattern candidates.

## 5.1 Data Collection

In this section, studied cases are derived from the interview data. Additionally, public visible cases of API platforms are introduced as further examples of known uses of the solution approaches.

We conducted semi-structured interviews with API provider stakeholders to answer the three RQs of this study. As visible in the interview guideline attached to this thesis, the interviews consist of two parts. The first part aims to answer general questions about the context of the API, industry, team, and broader environment. The second part targets the current work of the API provider, current and past issues the interviewees face, and applied solution approaches and their outcome. In total, 15 interviewees working in the field of API management have been questioned in 16 interviews. Table 5.1 gives a classification of the participants and their firms.

As visible in table 5.1, this study draws interview data from several different industries including finance, automotive, industrial manufacturing, IT services, and insurance. Furthermore, the participating companies vary in size from small start-ups to big international corporations. The goal is to capture experience from as many backgrounds as possible. One firm originates from the US, the others from Europe.

In the interviews, the interviewers tried to narrow down the questions to the most prominent project that the interviewee is currently working on. In some cases, the interviewee offered insights into more than one project. To reflect upon the experiences associated with each case, encodings are associated with cases and not interviews. Table 5.2 illustrates the described development.

| Number | Classification | Role | Employees | Duration | Participants |
|---|---|---|---|---|---|
| 1 | Multi-banking startup | Backend Developer | 11-50 | 00:22:52 | IV1 |
| 2 | Industrial manufacturing | Internal Consultant | >100,000 | 00:44:09 | IV2 |
| 3 | Automotive | Product Owner | >100,000 | 00:48:49 | IV3, IV4 |
| 4 | Software & IT service provider | Software Architect | 1001-5000 | 00:42:25 | IV5 |
| 5 | IT service subsidiary | Portfolio Manager | 1001-5000 | 00:51:12 | IV6 |
| 6 | Insurance subsidiary | Software Architect | 51 - 250 | 00:59:28 | IV7 |
| 7 | Industrial manufacturing | Technical Lead | >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 | Software & IT service provider | Internal Consultant | 5001 - 10,000 | 00:50:49 | IV11 |
| 11 | Software & IT service provider | Integration Architect | 11-50 | 00:56:29 | IV12 |
| 12 | Automotive | Product Owner | >100,000 | 00:51:48 | IV3, IV4 |
| 13 | Software & IT service provider | Technical Lead, Product Owner | >100,000 | 00:55:25 | IV13, IV14 |
| 14 | Software & IT service provider | Software Architect | 1001-5000 | 00:50:49 | IV5 |
| 15 | IT service subsidiary | Portfolio Manager | 1001-5000 | 00:31:58 | IV6 |
| 16 | IT service subsidiary | Internal Consultant | 1001-5000 | 00:45:44 | IV15 |

Table 5.1: Interviews

| Number | # Interview | Architectural Openness | Maturity | Case |
|---|---|---|---|---|
| 1 | 2 | Partner | Pilot | C1 |
| 2 | 3, 12 | Public & Partner | Production | C2 |
| 3 | 4, 14 | Public | Production | C3 |
| 4 | 4, 14 | Partner | Production | C4 |
| 5 | 5, 15, 16 | Group | Production | C5 |
| 6 | 6 | Group | Pilot | C6 |
| 7 | 7 | Private | Development | C7 |
| 8 | 8 | Public & Partner | Production | C8 |
| 9 | 9 | Partner | Production | C9 |
| 10 | 9 | Public & Partner | Production | C10 |
| 11 | 10 | Partner | Production | C11 |
| 12 | 11 | Public & Partner | Production | C12 |
| 13 | 13 | Public & Partner | Production | C13 |
| 14 | 13 | Private | Development | C14 |

Table 5.2: Cases

As laid out in chapter API Management, API providers utilize API management platforms. For the companies interviewed, the API gateway is demonstrated to be the most fundamental of the platforms. This aligns with the API platform hierarchy from De (2017) which is shown in figure 2.1. The gateway can be reused internally for several developer portals and thus, API management systems. The developer portal builds on top of the API gateway (De, 2017, p. 17). Within the studied cases, developer portals are used to offer a set of API products or services of which each contains a set of endpoints. To capture influence factors and context of the known usage for each pattern, interviews are split into cases on a developer portal level. Since API gateways are infrastructure and may serve as a baseline for several projects or initiatives, it is difficult to narrow down context. On the other hand, interviewees are managing multiple API products that all shared general context. The case creation on an API portal level allows for the best substantive significance based on granularity and available sufficient information (Dube & Pare, 2003; Löhe & Legner, 2010a). Thus, the interviews are broken down into distinct API portals on a developer portal level. Table 5.2 links each interview to its associated cases. The first interview is not connected to any case. It has been foremost conducted to explore the domain of API management.

Table 5.2 hints the architectural openness and maturity of the platform in focus for each case. The architectural openness describes the degree of visibility and access of the supply and demand side of the API platform to third-party developers (public), partnering organizations (partner), subsidiary firms (group), or within the boundaries of the organization (private). It is based on the differentiation of visibility and access from De (2017) which differentiate between public, partner, and private APIs (De, 2017, p. 7). The maturity level documents the current stage of the API platform. We identified three different maturity levels in the studied cases: production, in pilot phases with internal and external partners, and in early stages of development. These attributes further illustrate the differences between cases derived from the same interviews. For instance, interview 13 provides data about two API platforms that are captured in cases C13 and C14. C13 describes a production API platform that is opened to external partners and the public while C14 documents a private developer portal that is still under development.

In the following, interviewees are referenced to underline collected experience and referenced by their number, e.g. IV1 to reference interviewee one. Cases are linked to refer to known uses of solution approaches and are referenced by their case identifier, e.g. C1 to refer to case number one.

Publicly visible API platforms are utilized as further known uses of detected solution approaches. For example, *Pattern 10: Tailoring APIs to products* references Twilio and Stripe as organizations that utilize the solution approach. Twilio is also used to illustrate the example of *Pattern 10: Tailoring APIs to products*. Captured screenshots and references to the Twilio website illustrate the visible implementation. Similar to De (2017) which list popular public APIs, this study identified several popular API platforms (De, 2017, p. 8-10). In the following, utilized public developer portals are listed.

- Mercedes-Benz[1]
- SendGrid[2]
- Stripe[3]
- Twilio[4]

The Mercedes-Benz developer portal is referenced as an example for good design[5]. Twilio and Stripe are referenced as examples for good product documentation[6]. SendGrid is used as an example for role management. It is also considered a popular and well created developer portal[7].

In the following section, the pattern language is presented.

## 5.2 Pattern Language

This chapter specifies a pattern language that is used to create the pattern catalog. It documents all elements of the pattern language, their properties, and their relationships. An overview of the pattern language is given in figure 5.1. It is built on best practices derived from the literature. A summary of the pattern literature is given in chapter Related Work.

As visible in figure 5.1, the pattern language consists of five different elements: Influence Factors, Concerns, Stakeholders, Pattern Candidates, and Patterns. In the following, each element is defined in detail.

**Stakeholders** apply solution approaches to solve their concerns. Uludağ et al. (2019) define stakeholders as the persons that are involved, affected, or influenced by the domain (Uludağ et al., 2019). A map of the API provider-consumer relationship and naming conventions are introduced in section Roles and Stakeholders.

**Concerns** describe the goals, responsibilities, or risks of the stakeholders (Uludağ et al., 2019). Each concern is raised by at least one stakeholder (Uludağ et al., 2019). Therefore, a concern can always be traced back to a stakeholder that voiced the concern. They are phrased as questions that require an answer (Uludağ et al., 2019).

**Influence Factors** are utilized to put solution patterns into perspective (Buschmann et al., 2007a, p. 98). Sophisticated patterns might fit better to mature API platforms while starter patterns fit to API platforms in development (Khosroshahi

[1]https://developer.mercedes-benz.com/
[2]https://sendgrid.com/
[3]https://stripe.com/
[4]https://twilio.com/
[5]https://pronovix.com/blog/best-developer-portals-2020#hook08
[6]https://documentor.in/2148/best-examples-product-documentation-guides/
[7]https://www.quora.com/Which-companies-have-the-best-developers-website-and-API-documentation

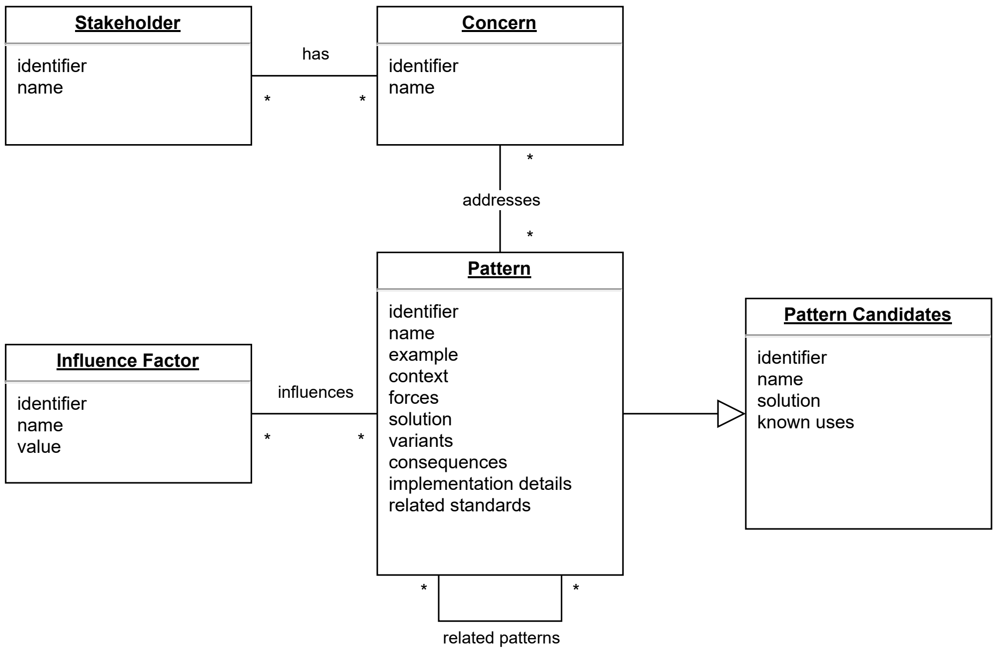


Figure 5.1: Meta-Model of the Pattern Language

et al., 2015). Thus, the influence factors support the API management in their decision making. They are derived from discovered context attributes. Context attributes, values, and their distribution across the cases are presented in section Influence Factors.

**Pattern Candidates** are solution approaches that are identified within the studied cases. Each pattern is a pattern candidate. A pattern candidate is validated if it fulfills the rule of three known uses as established by Coplien (1994) (Buckl et al., 2008; Coplien, 1994). Thus, a pattern candidate is validated by its known uses. Three known uses are required to make a pattern candidate a pattern (Buckl et al., 2008; Coplien, 1994). The rule of three is also practiced by Uludağ et al. (2019) as a means to confirm patterns (Uludağ et al., 2019).

**Patterns** are documented solutions for recurring concerns based on a particular context (Buckl et al., 2008; Gamma et al., 1994). Patterns build the core of the pattern language. They are linked to concerns and thereby, also to stakeholders. Patterns are put into perspective by influence factors.

Pattern languages utilize additional elements to capture solutions such as principles or anti-patterns. Uludağ et al. (2018) define principles as "enduring and general guidelines that address given concerns by providing a common direction for action" (Uludağ et al., 2019, p. 4). They are less concrete than patterns and provide overall guidelines (Uludağ et al., 2019). Anti-patterns offer revised solutions to common mistakes (Uludağ et al., 2019). The goal is to avoid typical pitfalls and transform problems into opportunities (Brown et al., 1998; Uludağ et al., 2019). Patterns provide a fitting framework to document the findings of this thesis. Principles and anti-patterns are not utilized to simplify the catalog.

In the following, the forms of all elements are explained. All elements have an identifier and a name to ease referencing (Uludağ et al., 2019). Influence factors have a set of values. Each influence factor describes one context attribute and captures context values within the known cases. Each pattern candidate has a short description of its solution approach. It is used to describe the overall idea of the candidate. Additionally, each pattern candidate has a list of known uses. It references the studied cases that apply the solution approach.

Patterns are the most complex element within the pattern language. The pattern literature provides different popular forms (Buschmann et al., 2007a; Fowler, 2006; Uludağ et al., 2019). Each form provides a different presentation to communicate patterns (Buschmann et al., 2007a, p. 92). The pattern form should be based on the target audience and the intent of the pattern language (Buschmann et al., 2007a; Fowler, 2006).

The form used in this study follows the guidance from the pattern literature and uses best practices defined by Gamma et al. (1994, Coplien (1994), Brown et al. (1998), Buschmann et al. (2007), Fowler (2006), Buckl et al (2013 ), Khosroshahi et al. (2015), and Uludağ et al. (2018) (Brown et al., 1998; Buckl et al., 2013; Buschmann et al., 2007a; Coplien, 1994; Fowler, 2006; Gamma et al., 1994; Khosroshahi et al., 2015; Uludağ et al., 2019). It builds on top of related API management pattern languages from Lübke et al. (2019), Zimmermann et al (2017), and Zimmermann et al (2020) (Lübke et al., 2019; Zimmermann et al., 2020; Zimmermann et al., 2017). Each pattern is presented using the following sections: Stakeholders, Concerns, Example, Context, Forces, Influence Factors, Solution, Consequences, Implementation Details, Related Standards, Related Patterns, and Known Uses. Additionally, the section Variants is used if necessary. Each section is explained below.

Sections **Stakeholders** and **Concerns** reference linked stakeholders and concerns. Each pattern provides a solution approach to one or more concerns and each concern is raised by a set of stakeholders. The two sections give an overview of stakeholders and concerns that are associated with the pattern. In the literature, the section Concerns is also referred to as 'Problems' (Gamma et al., 1994; Lübke et al., 2019; Uludağ et al., 2019; Zimmermann et al., 2020).

Section **Example** is used to illustrate the solution approach based on a real-world example (Lübke et al., 2019; Uludağ et al., 2019; Zimmermann et al., 2020). The examples are either derived from the studied cases or publicly available developer portals as listed in section Data Collection.

Section **Context** briefly describes the setting in which the solution approach is utilized. Lübke et al. (2019) use 'Context' in their work to describe the point in time in which the pattern should be applied (Lübke et al., 2019). Uludağ et al. (2018) and Zimmermann et al. (2020) use 'Context' to present the knowledge foundations on which the concern and pattern meet (Uludağ et al., 2019; Zimmermann et al., 2020).

Section **Forces** is used to specify a bulleted list of impediments and challenges of the status-quo that are resolved and balanced by the pattern (Lübke et al., 2019;

Uludağ et al., 2019; Zimmermann et al., 2020). It specifies the forces that play into the concerns and provides an understanding about why the solution approach is a solution to the concerns (Buschmann et al., 2007a, p. 37).

Section **Influence Factors** documents the most important context variables and their values as identified in the known cases. The influence factors put the pattern into perspective and describe the most important context variables and their distribution (Khosroshahi et al., 2015). They support the stakeholders in the assessment of the context (Buschmann et al., 2007a, p. 92).

Section **Solution** "describes the elements that make up the design, their relationships, responsibilities, and collaborations" (Gamma et al., 1994, p. 3). It does not explain specific implementation details but offers an abstract description (Gamma et al., 1994, p. 3). This ensures the reusability of the pattern (Gamma et al., 1994, p. 3).

Section **Variants** is optional and documents different variants of the same solution approach (Lübke et al., 2019; Uludağ et al., 2019). Variants are used to emphasize the differences in the influence factors, stakeholders, or alternating elements. Variants provide a way to add additional insights to a pattern by highlighting alternatives and variations. Alternatively to variants, the pattern can be split up into related patterns. This decision is made for each pattern individually based on similarities and differences between the variants.

Section **Consequences** lists both benefits and liabilities of a pattern (Gamma et al., 1994, p. 3). Consequences are essential to the evaluation of a solution approach (Gamma et al., 1994, p. 3).

Section **Implementation Details** provides additional guidance for the implementation of a solution approach. Lübke et al. (2019) and Zimmermann et al (2020) utilize similar sections 'How it works' and 'Implementation hints' to guide the observer through technical implementation details (Lübke et al., 2019; Zimmermann et al., 2020). In this study, the section is used to document recurring implementation approaches identified in the studied cases and public visible instances. Implementation details are used to describe common design decisions for the developer portal, document sequencing implementation steps, and mention common pitfalls.

Section **Related Standards** is used to refer to related and similar best practices, standards, principles, pattern, and conventions in the literature (Lübke et al., 2019; Uludağ et al., 2019; Zimmermann et al., 2020). They support observers by pointing them to additional material. The ITIL (2019) offers a set of guiding principles for service value systems (Limited & Office, 2019, p. 39). Other best practices are derived from the related literature or common state-of-the-art software engineering practices.

Section **Related Patterns** is used to describe relationships between patterns within the pattern catalog. Patterns can work in compliment, act as alternatives, or depend on each other (Buckl et al., 2013).

Section **Known Uses** lists the studied cases that utilize the solution approach.

It also references the publicly visible instances of the solution approach that are defined in section Data Collection. Known uses ensure the reusability of a pattern (Buckl et al., 2013).

Related pattern languages use additional sections such as 'Non-solution', 'Resolution of forces', 'Further discussion', and 'General form' (Lübke et al., 2019; Uludağ et al., 2019; Zimmermann et al., 2020). They are left out to reduce the complexity, because they were found to be too abstract, or not applicable to the management patterns of this thesis.

In the following section, the identified stakeholders and their relationships are mapped.

## 5.3 Roles and Stakeholders

This section presents a map of the relationships between stakeholders and software artifacts within API management. The map is derived from the interview data and aims to provide an overview of identified roles, their collaboration, and their use of software artifacts. Additionally, the map is used to introduce naming conventions for stakeholders used in the pattern catalog. As argued by Buckl et al. (2013), synonyms and homonyms have to be identified and resolved (Buckl et al., 2013). For this, discovered stakeholders and roles are discussed and similarities and differences with the literature are outlined.

Figure 5.2 maps relationships between stakeholders and software artifacts. Most entities are either optional or can be merged, replaced, or rearranged. The map aims to abstract the relationships to reflect the most common constellations. The following roles, teams, and stakeholders have been identified.

- End user
- Application provider
- Customer support
- Portal provider
- Gateway provider
- Backend provider
- API governance
- Legal
- Sales and marketing
- CIO

Software applications implement web APIs to communicate with remote services (Tan et al., 2016). The application provider integrates the web API calls into the application's code base. In the application runtime, requests are sent to web APIs to provide the end user composite services (Tan et al., 2016). The end user repre-

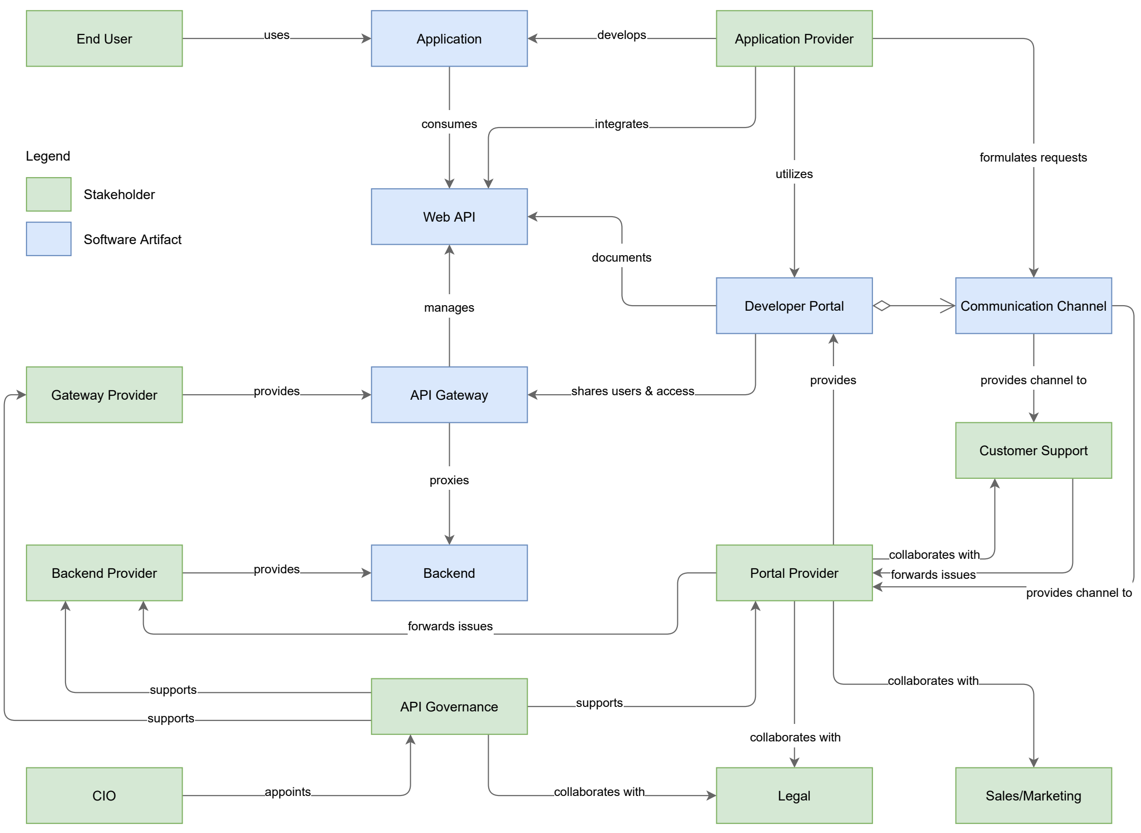


Figure 5.2: Relationships between Different Roles, Teams, Stakeholders, and API Platforms of API Management

sents the customer that is using the application. Web APIs are commonly managed by API platforms (De, 2017, p. 11). Figure 5.2 connects the application provider to the developer portal and its communication channels. This illustrates the utilization of developer and social boundary resources as described in section Boundary Resources. The term API consumer is used to describe both the application and application provider. Indirectly, it also refers to the end user. The web API can be provided by the same team, the same organization, a subsidiary firm, a partner organization, or a third-party provider. The providing entity is called API provider for the remainder of this thesis.

The API provider is further divided into several API management platforms, teams, and roles. Figure 5.2 defines two general paths of collaboration between API consumer and API provider. In both paths, the collaboration happens on an API management platform, the API gateway or the developer portal. The first path starts from the application. The application consumes the web API. Each web API call from the application is channeled through the API gateway. The API gateway is provided by an infrastructure team which is called gateway provider for the remainder of this thesis. Section API Gateway characterizes the API gateway as a reverse proxy. The API gateway platform fetches requested capabilities from offered backend services and then returns the fetched data to the requester.

In the remainder of this study, the term backend is used as defined in section Cloud Computing to reference any software component, like microservices or web services, that provide an interface and access to capabilities through the web. The requested capabilities are provided by backend services. The entity that provides a backend service is named backend provider. The backend provider acts as the supply side of the API platform. The openness of the supply and demand sides of a platform characterize it as either internal, many-to-one, or multi-sided. If the backend provider is based in a partnering or third-party organization, it is itself an API provider that acts as an orchestration layer for other backend services. The developer portal becomes an API marketplace if both the supply and demand side of the platform are open for partnering firms or the public.

The second path of collaboration is drawn between the application provider and the developer portal. The developer portal is defined both as a web client of the API gateway and a portal that offers documentation, specifications, and further artifacts to guide the application provider. It is maintained by the portal provider. The portal provider manages the API brokering. The developer portal has to market, sell, and document the APIs, and serve as a tool for the API consumer to find them and manage their integrations. The portal offers communication channels. This can be as simple as an email address provided in the imprint, contact forms, or more complex communication tools like forums. The contact inquires are forwarded through the developer portal to either a dedicated customer support or to the portal provider. Additionally, the customer support team or portal provider will forward the issues to the backend provider if necessary. The portal provider collaborates with the customer support, legal, and sales and marketing teams to manage customer requests, contracts, marketing information, pricing, and more. Figure 5.2 illustrates this central role of the portal provider.

Overall, three different API provider entities have been identified. The portal provider, gateway provider, and backend provider. All three provider entities might be the same team, one organization, subsidiary firms, partner organizations, or different third-party providers. The constellation of the different teams and additional roles varies across organizations. It can be noted that the utilization of API management platforms is not always the case. Web APIs can also be provided directly by backend services. Commonly, the API gateway provider and portal provider form the inner core of API management while the backend providers provide interfaces to their respective services. In the remainder of this thesis, the term API management is used to capture the portal provider, gateway provider, and all additional management roles that are included in the provision management. In bigger organizations, the gateway platform might be shared across several API management initiatives. In this case, the portal provider acts as the central API management entity of the platform in focus.

The API governance acts as a central authority of quality assurance within the API provider organization. If utilized, it is commonly instigated by the CIO or upper management. It supports the API management and the backend providers. It issues policies that have to be implemented into the API gateway by the gateway provider.

Overall, the API provision management includes several stakeholders and roles possibly distributed across several organizations. Figure 5.2 aims to map the most common relationships and flow of collaboration. Instantaneous collaborations have been left out. For instance, the API management might reach out to the application provider for pilot projects or for closer collaboration.

In the following, introduced naming conventions are linked to the literature. Zhu et al. (2014) define common roles that interact with the API management software from IBM (Zhu et al., 2014). They describe API administrators, developers, product managers, and IT operations. Medjaoui (2018) differentiates between technical and business roles in an API team (Medjaoui et al., 2018). In this study, those roles are shared or divided between the three providing entities. The naming conventions of API provider and consumer are common practice and included in the web services definition from the W3C[8] (Basole, 2016; Bonardi et al., 2016; Bondel et al., 2020; De, 2017; Krintz & Wolski, 2013; Mathijssen et al., 2020; Yu & Woodard, 2009; Zhu et al., 2014; Zimmermann et al., 2020). The naming introduced for end user, application developer, and application also follow common practices (Brown et al., 1998; De, 2017; Zhu et al., 2014). De (2017) also utilizes the term backend service to describe the origin of remote capabilities (De, 2017, p. 17, 22).

De (2017) defines the API team as the provider of the API (De, 2017, p. 13). In this study, the API provider entity is further split to better reflect the findings from the interview data. De (2017) also defines the API product owner as the person or organization that manages the API as a product (De, 2017, p. 175). In figure 5.2, this responsibility is shared and divided between the backend provider and the portal provider. In the studied cases, different constellations have been identified. In some cases, the portal provider tailored API products. In other cases, one team is acting as both the backend provider and portal provider and the responsibility is shared. In all cases, a collaboration between the two entities is required to provide API products to the API consumers.

As described in chapter Objectives, this study focuses on an API provider perspective. Each pattern captures applicants and potential collaborators. Applicants raise concerns and implement the solution approaches. Potential collaborators can be engaged in the solution approach. All stakeholders that act as applicants for patterns are listed below.

- S1: API management
- S2: Portal provider
- S3: Backend provider
- S4: API governance

As illustrated in figure 5.3, the term API management captures both the portal provider and gateway provider responsibilities. It is used to emphasize that both the portal and gateway provider share concerns and have to apply the solution

[8]https://www.w3.org/TR/ws-arch/#whatis

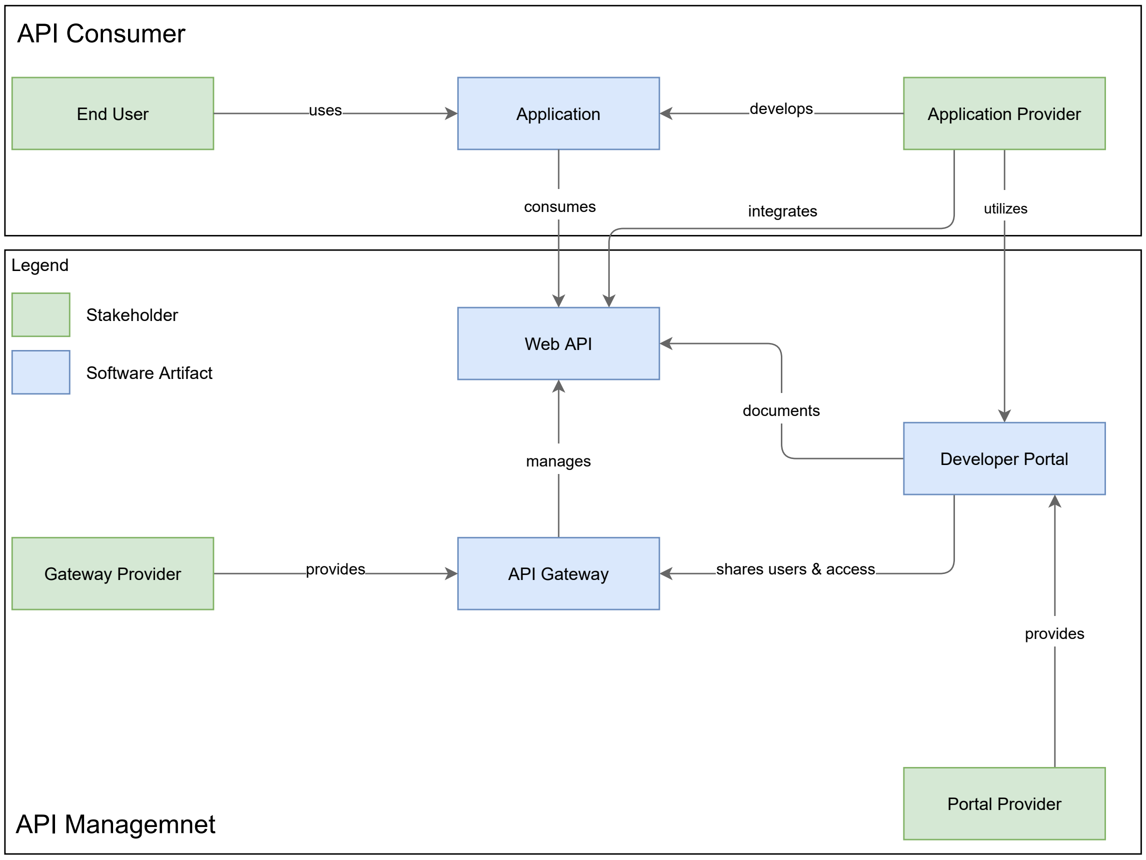


Figure 5.3: Core API Management Responsibilities

approach in close collaboration. The portal provider manages the API brokering and provides supporting material to the API consumer. Thus, multiple API management concerns are raised by the portal provider. Other concerns are raised by the backend provider, mostly in addition to the API management or portal provider. The API governance acts as a central authority. It is not providing IT artifacts but support and guidance. To conclude, documented patterns target stakeholders within the API provider entity. The API consumer includes the application, the application provider, and the end user of the applications. The API consumer is not applying the solution patterns documented in this study. Solution approaches will however affect them. Both the end user and the application provider are treated as the customers of the web API. It is the responsibility of the API provider roles, teams, and organizations to find adequate solutions and implement them into their systems. In some patterns, the API consumer is listed as a potential collaborator.

## 5.4 Influence Factors

Influence factors are introduced as part of the pattern language. Khosroshahi et al. (2015) utilize them to enhance their pattern catalog and state: "Influence factors determine which stakeholders, concerns and patterns have to be chosen" (Khosroshahi et al., 2015, p. 3). Thus, influence factors describe the setting of the analyzed cases in which a pattern is applied and support organizations with the pattern selection process (Khosroshahi et al., 2015). In this study, the term 'context' is used to describe all detected context variables and their values. The term influence factors is used to capture those variables and their values that actively influence one specific solution approach and make a pattern viable to solve the linked concerns. Each pattern can be dependent on a different set of influence factors.

In the following, the selection process of context variables is documented and the overall distribution of values presented. Khosroshahi et al. (2015) conducted surveys to collect influence factors (Khosroshahi et al., 2015). This study uses semi-structured interviews to collect data. First, the relevant literature was analyzed to derive potential influence factors that have an impact on the API management. Second, we developed additional context variables and values based on the encodings of the interviews. Overall, 20 context variables are used. Table 5.3 presents the list of context variables and their sampling across the 14 studied cases. The table follows the design of Löhe and Legner (2010) (Löhe & Legner, 2010a, 2010b). They document 24 context attributes that were derived from the related literature (Löhe & Legner, 2010a, 2010b). Since SOA and API management are related streams of research, this study utilizes attributes defined by Löhe and Legner (2010). Six context attributes and associated values are taken over from Löhe and Legner (2010) and five more have been adapted or merged to better fit the API management perspective. Next, all context attributes are presented in detail.

The attribute *Architectural Openness* fulfills a similar purpose to the attribute *SOA Scope* from Löhe and Legner (2010) but is derived through the encodings. The attribute values are based on the definitions from Hussain et a. (2020) (Hussain et al., 2020). Private API platforms are only used internally (Hussain et al., 2020). The attribute value *Group* is added based on the encodings and emphasizes the usage of an API platform between subsidiaries and potentially a parent company. This comes with additional complexity in communication, billing, and contractual work. The value *Partner* describes collaboration between a known set of close partner firms e.g. based on strategic projects (Hussain et al., 2020). Public API platforms are opened to the public (Hussain et al., 2020). They can still be restricted through onboarding processes but generally everybody can register or apply for it as the developer portal is visible over the web. Cases are counted multiple times if a platform differentiates its architectural openness for different products. Notably, most studied cases offer APIs to partner organizations or the public at the point in time the interviews were conducted.

*Maturity* describes the platform's current stage within its life-cycle. The literature defines several possible maturity models for APIs (European Commission. JRC., 2019). This study uses a simple model based on the study data where 71% of the researched platforms are in production while 14% are either in development or in a pilot phase. *Number of API Consumers* captures the approximate number

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private [#2, 14%]* | Group [#2, 14%]* | Partner [#9, 64%]* | Public [#6, 43%]* |
| Maturity | Development [#2, 14%] | Pilot [#2, 14%] | Production [#10, 71%] | |
| Number of API Consumers | <20 [#6, 43%] | >20 [#3, 21%] | >10,000 [#3, 21%] | na [#2, 14%] |
| Partner Type | B2B [#12, 86%]* | Business to Consumer (B2C) [#3, 21%]* | Business to Government (B2G) [#1, 7%]* | none [#2, 14%]* |
| Type of Platform | Marketplace [#2, 14%] | Developer Portal [#9, 64%] | Backend APIs [#2, 14%] | na [#1, 7%] |
| Network Topology | 1:1 [#0, 0%] | 1:n [#6, 43%] | m:n [#8, 57%] | |
| Service Granularity | Business Process [#2, 14%]* | Activity & Task [#10, 71%]* | Utility & Entity [#3, 21%]* | na [#2, 14%]* |
| Offered API Capabilities | Data [#11, 79%]* | Function [#14, 100%]* | | |
| API Consumer Heterogeneity | Homogenous [#4, 29%] | Heterogeneous [#10, 71%] | | |
| Monetization | Free [#3, 21%]* | In Product [#2, 14%]* | Contractual [#8, 57%]* | Per API call [#6, 43%]* |
| Initial Driver / Trigger | Top down [#7, 50%]* | Bottom up [#7, 50%]* | na [#3, 21%]* | |
| Number of API calls | Many [#9, 64%] | Few [#6, 43%] | | |
| Value Chain Integration | Vertical [#7, 50%]* | Horizontal [#6, 43%]* | Internal [#2, 14%]* | |
| Number of API Products | <20 [#7, 50%] | >20 [#2, 14%] | na [#5, 36%] | |
| Onboarding Process | Manual onboarding [#9, 64%]* | Self-service [#6, 43%]* | na [#3, 21%]* | |
| Network Governance | Focal [#14, 100%] | Polycentric [#0, 0%] | | |
| Networking Target | Efficiency [#5, 36%]* | Innovation [#3, 21%]* | Channel Extension [#6, 43%]* | Venture [#5, 36%]* |
| Process Output | Virtual [#12, 86%] | Physical [#2, 14%] | | |
| Initial Trigger Motivation | Strategic Pressure [#10, 71%]* | Process Pressure [#0, 0%]* | IS Pressure [#7, 50%]* | |
| Type of Gateway | Commercial [#8, 57%] | Open source [#2, 14%] | none [#2, 14%] | na [#2, 14%] |

Table 5.3: Context Attributes and Values with [# of occurrence in cases, percentage of cases] n=14, * denotes multiple counting of cases

of organizations that integrate with the platform's APIs. It is derived from the attribute *Number of Partners* used by Löhe and Legner (2010) (Löhe & Legner, 2010a, 2010b). The value *na* captures cases where no data was available. Both attributes *Maturity* and *Number of API Consumers* are derived from the interview encodings.

The attribute *Partner Type* and its values are taken over from Löhe and Legner (2010) (Löhe & Legner, 2010a, 2010b). It describes the type of organizations that operate on the demand side of the platform. As visible in table 5.3, one case can be multiple counted. Most (86%) of the studied platforms operate in a B2B environment. *Type of Platform* is derived from the encodings and characterizes the platform in question. The platform in focus is most commonly (64%) a developer portal. In two cases (14%), no API management platform is utilized and the backend services that provide the API products and services are consumed directly. The difference between a marketplace and developer portal is defined in section Roles and Stakeholders and based on the architectural openness of the supply side. Both API marketplaces and developer portals utilize API gateways. Thus, 12 out of 14 cases operate an API gateway.

The attribute *Network Topology* is taken over from Löhe and Legner (2010) (Löhe & Legner, 2010a, 2010b). It is important to emphasize the difference between *Network Topology* and *Type of Platform*. A platform of type *Marketplace* leads directly to a m:n topology. Nevertheless, developer portal based platforms can also be granted a m:n network topology if the backend services are provided by different business units or subsidiaries. The reasoning behind this design decision is to characterize the level of complexity within the platform's API provision management. An international cooperation with more than 100,000 employees might generate a high level of complexity even without opening the supply side to external parties. To conclude, m:n denotes the highest level of complexity within the API brokering that can be both achieved by developer portal and marketplace based platforms. As visible in table 5.3, the interview cases are spread relatively evenly between 1:n (43%) and n:m (57%) topologies.

The attribute *Service Granularity* is used by Löhe and Legner (2010) to give insights about the kind of services that are offered within the SOA-based business network (Löhe & Legner, 2010a, 2010b). Similarly, this study uses the attribute to give insights about offered API capabilities. As defined in section Web Service, API platforms can offer capabilities of different granularities. 14% of studied platforms offer at least one business process. 21% serve, among other things, utilities and entities. 71% of all platforms provide activities or tasks. This illustrates the general idea of the API Economy where applications utilize a composition of different APIs to create new value through mashups.

Löhe and Legner (2010) utilize *Integration Approach* to characterize the layer of integration (Löhe & Legner, 2010a, 2010b). They differentiate between business process, presentation, and data and function layer (Löhe & Legner, 2010a, 2010b). This attribute is replaced by *Offered API Capabilities* to better specify the integration on an API-based level. The attribute *Offered API Capabilities* is based on the interview encodings and categorizes the platforms capabilities. All studied platforms provide functionality and in most cases (71%), both functionality and data, are provided. Thus, no studied API platform offers read-only data consumption exclusively. In three cases, functionality but no data is provided. For those platforms, the API consumer is required to bring its own data to utilize the service functions. *API Consumer Heterogeneity* is derived from the attribute *Partner Heterogeneity* by Löhe and Legner (2010). It illustrates the composition of API consumers with regards to their industry. The majority (71%) of cases offers a heterogeneous set of consumers. Thus, the consumers are associated to different industries (Löhe & Legner, 2010a).

*Monetization* describes how the API calls are charged. One API platform may offer different monetization strategies for different offered products. 21% of the platforms provided API capabilities for free, for instance, to gain more market penetration. The attribute value *In Product* (14%) defines API capabilities that are included within an overall product pricing. In those two cases, the API consumer purchases a software application license which also includes access to the API platform. 57% of the cases include contractual agreements. 43% of the interviewed API providers bill some capabilities per API call.

*Initial Driver / Trigger* describes the initial forces at work that pushed for the API platform. Bottom-up initiatives can be incentivised by management and top-down programs can be supported by bottom-up forces. Therefore, the attribute is multiple counted. In three out of the 14 cases, the data is not available. In the remaining 11 cases, seven were triggered in a top-down and seven in a bottom-up manner.

*Number of API calls* aims to categorize the traffic of the API platform. Since specific data is not available for most interviewed cases, a rough characterization of many and few is used to give an basic idea of the traffic. *Many* denotes that API call based billing might be possible while *Few* stresses that other ways of monetization might be more suited since the APIs are not called regularly. Furthermore, API platforms that are in development or pilot phases might also have low traffic. In the studied cases, 64% of the API platforms are associated to many API calls while 43% of the platforms are associated with a low volume of API requests. The attributes *Monetization*, *Initial Driver / Trigger*, and *Number of API calls* are all derived from the encodings and define important influences for the API provision management.

The attribute *Value Chain Integration* is obtained from Löhe and Legner (2010). It categorizes the API consumer organizations and the platform's position within the value chain (Löhe & Legner, 2010a, 2010b). The cases are spread relatively evenly between vertical integration (50%) and horizontal integration (43%). The attribute value *Internal* (14%) is added to illustrate the value chain integration of internal platforms. The attribute *Number of API Products* is derived from the encodings and characterizes the amount of API products offered through the studied platform. 50% of all platforms offer less than 20 distinct API products. Only two platforms (14%) offer more than that. For 36%, this study lacks data for estimation.

The category *Onboarding Process* is developed based on the encodings. It illustrates how API consumer sign up and register applications. Manual onboarding is utilized in 64% of all cases. 43% of all cases support some level of self-service within their developer portal or marketplace.

The attribute *Network Governance* is inherited from Löhe and Legner (2010). Löhe and Legner (2010) identify 21% polycentric-based network governance. In this study, platforms are provided by one API provider. Thus, all platforms are based on focal governance. Löhe and Legner (2010) also document *Network Target* and *Network Purpose* (Löhe & Legner, 2010a, 2010b). This study characterizes the target of the business network via Ansoff's two by two matrix as utilized by Kambil (2008) (Kambil, 2008). The study cases are distributed across the targets of efficiency, innovation, channel extension, and venture.

The attribute *Process Output* from Löhe and Legner (2010) renders the final result of the web API integration. The final result can either be a digital service or product, thus, virtual or tied to a physical service or product (Löhe & Legner, 2010a). In 86% of the studied cases, a virtual end result is achieved while two API platforms (14%) enable a physical outcome.

Löhe and Legner (2010) document strategic, process-related, and IS-related pressure to describe different levels of influences (Löhe & Legner, 2010a, 2010b). This study combines those three attributes into *Initial Trigger Motivation* and uses them as attribute values to categorize what type of pressure is mentioned in the interviews. Thus, *Initial Trigger Motivation* describes the motivation behind the API platform creation. Studied cases are motivated by *Strategic Pressure* (71%) and *IS Pressure* (50%) while no case is based on process-related pressure. Strategic pressure includes the factors *Customer Access*, *Improvement*, and *Know-how* (Löhe & Legner, 2010a, 2010b). IS-related pressure includes, among others, the attribute values *Missing Interoperability*, *Heterogeneity*, *Redundancy*, *Legacy & Monolithic*, and *Costs* (Löhe & Legner, 2010a, 2010b). Process-related pressure lies between the strategic and IS-related forces on a process level and includes factors such as *Redundancies* (Löhe & Legner, 2010a, 2010b).

*Type of Gateway* is derived from the encodings and characterizes the underlying API gateway. 57% of studied platforms are based on commercial API gateway platforms. 14% utilize open source gateways. Two platforms (14%) do not utilize an API gateway. This number matches the amount of platforms of type *Backend APIs*.

Following attributes from Löhe and Legner (2010) are left out and could not be instantiated based on the interview data: *Network Stability*, *Cooperation Process*, *Cooperation Span*, *SOA affected Application*, *Coupling Approach*, *SOA Implem. Strategy*, *Info. Exchange Style*, *Coupling Intensity*, and *Standardiz. Scope*. In this study, *Coupling Intensity* can be derived from *Onboarding Process* and *Monetization*. Those attributes illustrate how intense the coupling between API provider and consumer are. The attribute *Communic. Type* is left out since it does not fit the API provision context. APIs are consumed by applications while API platforms are utilized by both applications and developers.

## 5.5 Concerns

Concerns describe the goals, responsibilities, or risks of the stakeholders (Uludağ et al., 2019). As described in section Pattern Language, each concern can be linked to several stakeholders and can be answered by different patterns. The patterns can work complimentary or as alternatives to answer the concerns. Overall, 32 concerns have been discovered by analyzing the interview data. In this section, an overview of identified concerns is presented. The concerns are categorized in seven categories: API as a Product, API management collaboration, Support management, Incident management, Quality management, Internal API platform initiatives, and Venture. The categorizes capture the main focus of the concerns.

**API as a Product** is a common term used to promote the treatment of APIs as digital products[9]. Following concerns are related to API product design and management:

- Q1: Who will be using the APIs?
- Q2: Which APIs should be offered?
- Q3: How to tailor backend services to APIs that fit the API consumer's needs?
- Q4: How to aggregate backend services?
- Q5: How to onboard a new API product onto the platform?
- Q6: How to fit the APIs to consumers' requirements?
- Q7: How to ensure market-fit?
- Q8: How to validate API offerings?
- Q9: How to trigger feedback from API consumers?
- Q10: How to offer a high-quality user experience for both business and developer roles?
- Q11: How to engage business roles of the API consumer?
- Q12: How to market API offerings to non-technical roles?
- Q13: How to market API offerings to application developers?
- Q14: How to notify API consumers about new API products?
- Q15: How to onboard new API products onto the platform?

**API provider collaboration** includes all concerns that focus on the collaboration between different API provider roles, teams, and stakeholders. The collaboration can be linked to other categories such as support, incident, or quality management but the focus is not on the API consumer. The following concerns are linked to this category:

- Q16: How to effectively and efficiently collaborate with other API provision

[9]https://developers.redhat.com/blog/2019/12/02/apis-as-a-product-get-the-value-out-of-your-apis/

teams?

- Q17: How to effectively and efficiently collaborate with first-level support?
- Q18: How to manage APIs within a group of subsidiary or partnering firms?
- Q19: How to centralize but allow distributed control of API management?

**Support management** captures all API management activities that aim to support the API consumer. This includes goals, responsibilities, and risks of service request management and the management of social boundary resources. Following concerns are categorized as support management concerns:

- Q20: How to communicate with API consumers?
- Q21: How to document API products?
- Q22: How to support developers with API integration?
- Q23: How to support potential API consumers without technical capabilities?
- Q24: How to manage non-complex, routine API consumer requests?
- Q25: How to onboard API consumers efficiently?
- Q26: How to support a growing number of API consumers?
- Q27: How to provide efficient support for API consumers?

**Incident management** aims to reduce the impact of defects and resolve issues as quickly as possible (Limited & Office, 2019, p. 121). It is closely related to support management. The following concern is linked to incident management:

- Q28: How to resolve bug reports effectively and transparently?

**Quality management** focuses on the reduction of incidents and includes activities such as analysis and motioning. One concern is linked to this category:

- Q29: How to ensure API service quality?

**Internal API platform initiatives** differ from their public counterparts in their trigger motivation, strategic goals, and other context attributes. The following concerns are raised specifically within internal API platform initiatives:

- Q30: How to onboard APIs to a private API platform?
- Q31: What is a good first step of a private API platform initiative?

**Venture** opportunities are created by answering concerns outside the boundaries of the API management. The following concern is answered in such a way that it creates a venture opportunity:

- Q32: How to target potential API consumers without technical capabilities?

In the following section, an overview of the pattern catalog is presented. The catalog links identified concerns to stakeholders and associated solution patterns.

## 5.6 Taxonomy

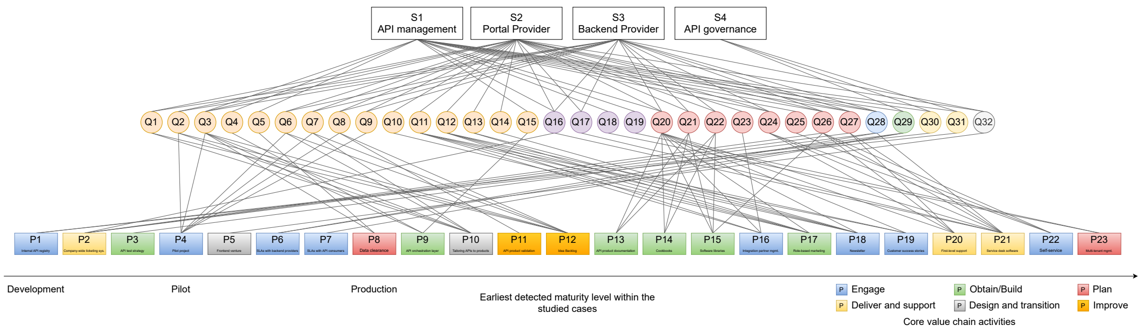

S1 API management
S2 Portal Provider
S3 Backend Provider
S4 API governance
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10 Q11 Q12 Q13 Q14 Q15 Q16 Q17 Q18 Q19 Q20 Q21 Q22 Q23 Q24 Q25 Q26 Q27 Q28 Q29 Q30 Q31 Q32
P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15 P16 P17 P18 P19 P20 P21 P22 P23
Development
Pilot
Production
Earliest detected maturity level within the studied cases
Engage
Obtain/Build
Plan
Deliver and support
Design and transition
Improve
Core value chain activities

Figure 5.4: Pattern Catalog Taxonomy

This section provides an overview of the pattern catalog. First, figure 5.4 is used to present identified solution patterns. Next, the pattern sequence and categories are explained in detail.

The pattern catalog consists of stakeholders, concerns, influence factors, pattern candidates, and patterns. Figure 5.4 offers a taxonomy of the pattern catalog by linking stakeholders to their concerns and concerns to their appropriate solution patterns. It utilizes the identifiers introduced in sections Roles and Stakeholders and Concerns to reference stakeholders and concerns respectively. A bigger version of the figure is attached to the appendix of this study. In total, four stakeholders are identified applicants for solution approaches. They are linked to 32 concerns. Each concern is linked to one or more patterns. The pattern catalog consist of 23 solution patterns. Since the pattern language does not map pattern candidates to concerns and stakeholders, they are not part of the taxonomy.

In figure 5.4, the patterns are sorted from left to right by the context attribute 'maturity level'. The earliest detected maturity level of the API platform within the associated cases is used to sort the patterns across the x-axis. Patterns that are derived from cases with API platforms in development are listed on the left, followed by patterns linked to pilot phases, and patterns linked to API platforms in production.

Concerns are colored based on the concern categories (from left to right): API as a Product, API management collaboration, Support management, Incident management, Quality management, Internal API platform initiatives, and Venture.

Patterns are categorized following the service value chain activities from ITIL (Limited & Office, 2019, p. 58). The service value chain activities are used to map each solution approach to one main value chain service activity. Figure 5.5 provides an overview of the activities. In the following, the service value chain categories are described.

As visible in figure 5.5, the service value chain consists of six activities.

- **Plan:** Planning activities establish a shared understanding across the organization (Limited & Office, 2019, p. 61).

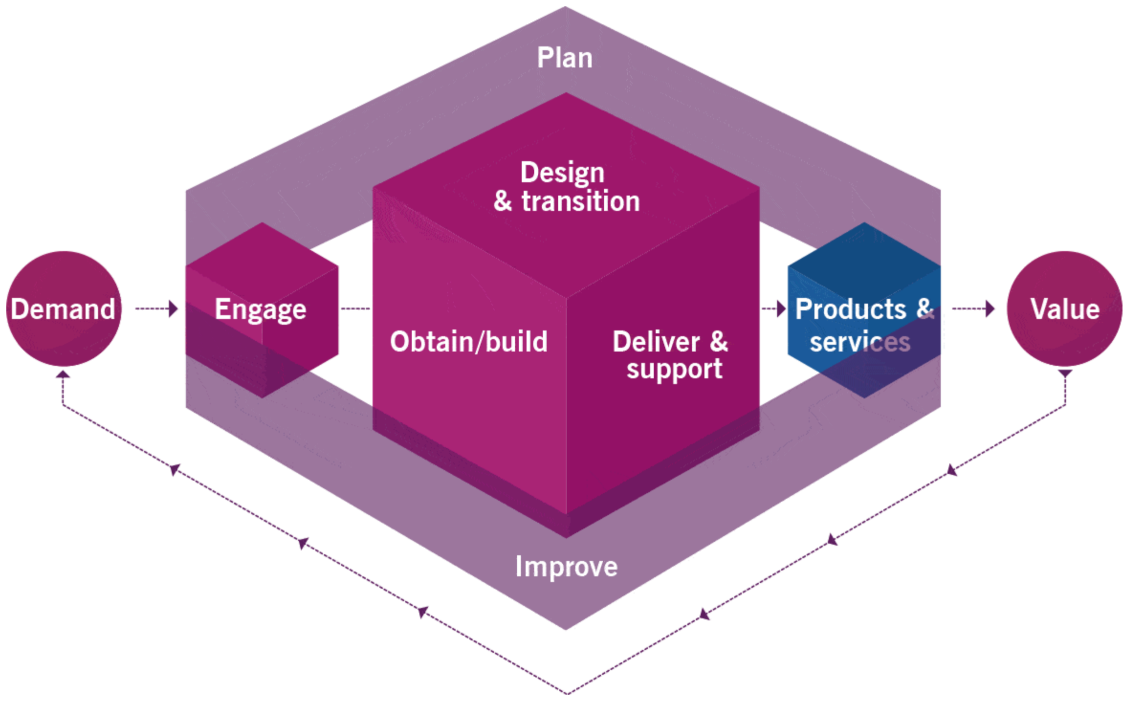


Figure 5.5: Pattern Categories Based on the Service Value Chain Activities from ITIL (Limited & Office, 2019, p. 58)

- **Improve:** Improvement focuses activities ensure continual improvement of services, products, and processes (Limited & Office, 2019, p. 62).
- **Engage:** Engagement with all stakeholder is used to maintain good relationships and identify needs (Limited & Office, 2019, p. 63).
- **Design and Transition:** Design and transition activities utilize identified room for improvement to create and adapt products and services (Limited & Office, 2019, p. 64).
- **Obtain/Build:** Obtain and build activities develop and maintain products and services based on agreed specifications (Limited & Office, 2019, p. 64).
- **Deliver and Support:** Delivery and support activities ensure delivery of products and services and agreed level of support for the stakeholders (Limited & Office, 2019, p. 65).

The engagement of API consumers is associated to be the main value chain activity of eight solution patterns. Delivery and support activities are linked to three patterns. Six patterns focus on obtain and build activities. Two patterns each center around design and transition, planning, and improvement activities.

Figure 5.4 can be used to follow concerns listed in section Concerns to associated patterns. In the following, all patterns and pattern candidates are listed.

## 5.7 API Management Patterns

This chapter presents all patterns and pattern candidates. Each pattern documents the following fields. The associated stakeholders and concerns are listed. A short example is used to illustrate the solution approach. The context and influencing forces are defined. Influence factors from associated cases are used to put the pattern into context. The solution is explained and derived consequences are listen. Implementation details further document implementation steps and notable details. The pattern is put into perspective by referencing related standards, related patterns, and the associated known uses. Next, all detected pattern candidates are listed. As defined in section Pattern Language, each pattern candidate has a name, a solution approach, and lists its known uses.

The following patterns are sorted by the earliest detected maturity level within the associated known uses. The order is also illustrated in figure 1 of the appendix. The pattern candidates do not follow a specific order but reference related patterns and pattern candidates in the solution description.

### 5.7.1 Pattern 1: Internal API registry

#### *Stakeholders*

The following applicants are derived from the study data:

- API management
- API governance

The following potential collaborators are derived from the study data:

- Backend provider

#### *Concerns*

- How to onboard APIs to a private API platform?
- What is a good first step of a private API platform initiative?

#### *Example*

The API management of C7 develops an internal API platform. The bottom-up initiative aims to register as many internal API offerings as possible onto one platform. The overall goal is to improve discoverability and reusability. The onboarding of APIs on the developer portal and API gateway is a time consuming effort. The API has to be created on the developer portal and its documentation moved. API consumers have to be notified about the new developer portal. The integration with the API gateway might also change the Uniform Resource Locator (URL) of the endpoints. This breaking change has to be communicated to all API consumers. The API management of C7 distinguishes different levels of integration. The first level of integration describes an internal API registry. The internal API registry lists all internal API offerings and platforms and links to their independent websites. This takes a fraction of time compared to the full integration and already improves the discoverability of APIs within the organization.

#### *Context*

Internal API platforms are used to improve API discoverability and concentrate API management tasks. An API platform initiative has to onboard API providers and backend providers onto its API platform.

### *Forces*

Forces that have to be resolved and balanced:

- API onboarding efforts
- Centralization efforts

### *Influence Factors*

Table 5.4 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 1: Internal API registry*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | Partner | Public |
| Maturity | Development | Pilot | Production | |
| Type of Platform | Marketplace | Developer Portal | Backend APIs | na |
| Networking Target | Efficiency | Innovation | Channel Extension | Venture |

Table 5.4: Influence Factors for Pattern 1

As visible in table 5.4, the solution approach has been associated with private API platforms. In the identified cases, the platform is still under development. *Pattern 1: Internal API registry* is associated to developer portals only. The target of the API initiatives is documented to be efficiency and innovation. It can be noted that C9 and C10, which have been listed as known uses, do not offer an API registry on their platforms but utilize an internal API registry within their organization. The influence factors for C9 and C10 have been left out of table 5.4, which aims to describe the context of the platform that offers the private API registry.

### *Solution*

An API registry catalogs API offerings and enables discoverability (De, 2017, p. 25-26). The API registry is part of the developer portal and lists all offered APIs (De, 2017, p. 25-26). An internal developer portal can utilize an API registry to maintain a list of all internal API offerings. Those API offerings do not necessarily need to host the documentation on the developer portal. The registry can link to the external sources instead. Additionally, the API does not need to be integrated with the API gateway of the API platform. The creation of an API registry offers an easy way to provide value to API consumers and proofs the usefulness of the API initiative. The API registry can be utilized to list internal API offerings as a first onboarding step and iteratively onboard the documentation of the internal APIs.

### *Variants*

In two cases, the API registry is maintained by the API governance and not by an API platform initiative. In this variant, the API governance maintains a list of all internal API platforms and offerings to ease its own governance activities but also to support API consumers in the detection of API offerings.

### *Consequences*

The following benefits are derived from the study data:

- Focus on value for the API consumers
- Improved visibility and discoverability of internal APIs
- Improved visibility of the API initiative

The following liabilities are derived from the study data:

- Curation efforts required

### *Implementation Details*

The API registry should be searchable to improve discoverability (De, 2017, p. 25-26). Each API should be associated with tags and meta data (De, 2017, p. 25-26). These should be indexed and integrated with the search feature (De, 2017, p. 25-26).

The applicant should curate registry entries and review proposed changes by the different API providers. This ensures the quality of the entries and enforces a common structure for each entry in the catalog. A good starting point is to set up an initial registry with known internal API offerings to promote the general idea. Growing popularity and usage of the registry can be used as an incentive for other API providers to create or requests entries.

### *Related Standards*

- Collaborate and Promote Visibility (Limited & Office, 2019, p. 39)
- Focus on Value (Limited & Office, 2019, p. 39)

### *Related Patterns*

*Pattern 1: Internal API registry* documents the first step of a three-level scale which can be utilized to manage the onboarding of API and backend providers onto an internal API platform. The three-level scale is documented in *Pattern Candidate 55: Integration levels*.

The solution approach explained in *Pattern Candidate 42: Declarative API platform* can be used to ease the collaboration with API and backend providers to add, maintain, and review registry entries.

### *Known Uses*

- C7, C9, C10, C14

### 5.7.2 Pattern 2: Company-wide ticketing system

#### *Stakeholders*

The following applicants are derived from the study data:

- API management

The following potential collaborators are derived from the study data:

- Backend provider
- CIO

#### *Concerns*

- How to support a growing number of API consumers?
- How to provide efficient support for API consumers?
- How to resolve bug reports effectively and transparently?
- How to effectively and efficiently collaborate with other API provision teams?

#### *Example*

Mercedes-Benz utilizes a custom company-wide ticketing system to manage defects across enterprise boundaries[10]. The latest version of the tool is called STARC[11]. Mercedes-Benz cooperates with third-party service providers to enable suppliers to integrate defect management and application life-cycle management tools with STARC. Defects are raised and forwarded to the team that has ownership over the defect component. This enables effective quality management throughout the development processes.

#### *Context*

API management has to effectively and efficiently integrate backend providers to its API platform. Backend providers commonly integrate with services on their own. Defect management within a complex distributed system requires transparent issue tracking and quality management processes.

[10]https://agosense.com/en/solutions/data-exchange/defect-management-automotive/
[11]https://agosense.com/en/ressources/blog/from-dante-to-starc-the-easy-way-with-agosensesymphony/

### *Forces*

Forces that have to be resolved and balanced:

- Efficient and effective management of defects
- Responsibility and prioritization management between the API provider teams
- Scalability of quality management activities

### *Influence Factors*

Table 5.5 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 2: Company-wide ticketing system*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | **Private** | Group | **Partner** | **Public** |
| Maturity | **Development** | Pilot | **Production** | |
| Type of Platform | **Marketplace** | **Developer Portal** | Backend APIs | na |
| Partner Type | **B2C** | **B2B** | **B2G** | **none** |
| Monetization | **Free** | In Product | **Contractual** | **Per API Call** |

Table 5.5: Influence Factors for Pattern 2

Table 5.5 shows that the solution approach has been applied with architectural openness of types private, partner, and public. The platform in focus is documented to be in development or production. We think *Pattern 2: Company-wide ticketing system* can be applied to any level of architectural openness and any maturity level. This is a notable difference to the influence factors detected for *Pattern 21: Service desk software* where the platform is in production only and only applied to API platforms that are open for partnering and public API consumers. Both marketplaces and developer portals are used in the context of *Pattern 2: Company-wide ticketing system*. The pattern is not associated with a product-based monetization strategy.

### *Solution*

A company-wide ticket system can be used to manage defects across business unit and subsidiary firm boundaries. It enables defect and application life-cycle management across a complex distributed system of backend services and is used to handle the communication and quality management in case of a defect. In contrast to a customer-facing service-desk system, the defects are not created by the public API consumers but initiated by the internally affected stakeholders. All requests are handled in an unified manner. Each ticket tracks the history, progress, and relevant people. If a ticket reaches the service provider, the provider team prioritizes the ticket within the context of their current work.

### *Consequences*

The following benefits are derived from the study data:

- Focus on value creation and relevance for the customer
- Reduction and management of complexity
- Support stakeholders and strategic partners efficiently

The following liabilities are derived from the study data:

- Ticketing system integration project required
- Training required to utilize the ticketing system

### *Implementation Details*

Company-wide ticketing systems can be implemented based on a variety of different software solutions. Since the integration of an universal defect management tool requires company-wide efforts, the API management should utilize whatever software is already available. If no such software is used, it should cooperate with the CIO and top management to push for a company-wide solution.

Common code hosting platforms such as GitHub[12] or GitLab[13] offer a variety of features for collaboration across project boundaries. Issues can be created for a specific code repository and assigned to the associated team. In case all distributed backend services have access to the same code hosting platform, it can be used as a company-wide ticketing system. Alternatively, custom application life-cycle management or defect management software can be utilized. For instance, service desk software can be configured for internal use.

All stakeholders of the API management have to be onboarded onto the ticketing system. This can include training courses and workshops. Central authorities within the organization should push for the creation of associated defect escalation processes. In case of a detected defect, the affected team creates a new issue for the team that has ownership over the defect component. This way, all defects are documented and communicated across team boundaries. The assigned team can convert the defect issue into sprint tickets and prioritize the work against their current workload. The ticket is resolved when the defect has been fixed.

[12]https://github.com/
[13]https://about.gitlab.com/

### *Related Standards*

- Collaborate and Promote Visibility (Limited & Office, 2019, p. 39)
- Focus on Value (Limited & Office, 2019, p. 39)
- Optimize and Automate (Limited & Office, 2019, p. 39)
- Service Request Management (Limited & Office, 2019, p. 156)

### *Related Patterns*

Defect management in production involves the API consumer and includes additional activities such as support request management. A service desk software should be utilized to communicate with the API consumer and collaborate with first-level support teams. The onboarding of a first-level support team is described in *Pattern 20: First-level support*. The integration of a service desk software is documented in *Pattern 21: Service desk software*.

As described in *Pattern Candidate 41: Inner source-based platforms*, inner sourcing can further increase collaboration between internal stakeholders. It can be utilized to collaborate on defect resolution.

SLAs document responsibilities of service providers. Incident management escalation processes can be defined and agreed upon. This is further examined in *Pattern 6: SLAs with backend providers*.

### *Known Uses*

- Mercedes-Benz
- C2, C7, C8

### 5.7.3 Pattern 3: API testing strategy

#### *Stakeholders*

The following applicants are derived from the study data:

- API governance
- API management

The following potential collaborators are derived from the study data:

- Backend provider

#### *Concerns*

- How to ensure API service quality?

#### *Example*

The API management of C3 and C4 provides API platforms for external partners and public API consumers. The API platforms are tied to software products provided to the end user. The software ecosystem includes APIs for external use and internal backend services that connect to the software products directly. A holistic staging and testing strategy is utilized to ensure quality and compatibility within the distributed system. Each backend service uses unit tests in isolation. Changes are deployed to a test environment. The deployment to the production stage includes test pipelines that ensure the compatibility between the remote software components. Additionally, the API management maintains API tests that test the API platforms and connected backend services end-to-end from an API consumer perspective.

#### *Context*

API platforms are part of a distributed system which involves several different backend services and provision teams. Testing and staging efforts have to be agreed upon with all stakeholders and have to be managed across team boundaries.

### *Forces*

Forces that have to be resolved and balanced:

- Automation efforts
- Complexity of distributed systems
- Testing efforts
- Service quality

### *Influence Factors*

Table 5.6 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 3: API testing strategy*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | [✓ Private] | Group | [✓ Partner] | [✓ Public] |
| Maturity | [✓ Development] | Pilot | [✓ Production] | |
| Number of API Consumers | [✓ <20] | [✓ >20] | [✓ >10,000] | na |
| Type of Platform | [✓ Marketplace] | [✓ Developer Portal] | [✓ Backend APIs] | na |
| Monetization | [✓ Free] | [✓ In Product] | [✓ Contractual] | [✓ Per API Call] |

Table 5.6: Influence Factors for Pattern 3

As visible in table 5.6, the solution approach has been applied with architectural openness of types: private, partner, and public. The platform in focus is documented to be in development or production. The number of API consumers, the type of the platform, and the monetization strategy vary across the associated cases. We think that *Pattern 3: API testing strategy* should be applied to all API platforms.

### *Solution*

An API testing strategy specifies the testing efforts that are taken to ensure the service quality of the API platform and all connected software artifacts. It aligns the testing efforts of the individual teams and requires collaboration between all stakeholders including the API gateway provider, portal provider, and backend provider teams. The effort should be centralized and managed by either the API management directly or a company-wide API governance authority.

### *Consequences*

The following benefits are derived from the study data:

- Increased confidence through test coverage
- Increased service quality
- Reduction of manual testing efforts through automation

The following liabilities are derived from the study data:

- Collaboration efforts required
- Constant test maintenance required
- Test data management required

### *Implementation Details*

A central API governance authority should support the definition and management of all testing efforts (De, 2017, p. 182). If no API governance team exists, the API management should take upon the testing responsibilities. A close collaboration with all backend providers and development teams is required.

Each backend provider has to ensure service quality in isolation. Thereby, each development team should adhere to best practices used in software development such as test driven development or extreme programming. This includes the implementation of functional tests such as unit tests, integration tests, and regression tests (Limited & Office, 2019, p. 158).

A test environment can be utilized to test the compatibility of changes on a system level. For this, a staging strategy has to be developed and agreed upon with all provider teams. Similarly, test automation can be utilized to automate testing efforts. Automated tests can be integrated into deployment pipelines that are triggered by each push of changes.

The API management should also test its software artifacts in isolation. The test domains of API management include: API interface specifications, API documentation, and API security (De, 2017, p. 154). API interface specification tests ensure the soundness of the API endpoints (De, 2017, p. 155). API documentation testing efforts are manual quality checks that ensure the actuality and completeness of the API documentation (De, 2017, p. 156). API security tests focus on the API gateway configuration and the validity of identity management, authorization, and authentication processes (De, 2017, p. 156). Additionally, API management can implement API performance tests (De, 2017, p. 158). API performance tests monitor quality metrics such as response times and latency (De, 2017, p. 162). For instance, API performance tests can be integrated into a testing stage to ensure that changes to the software ecosystem do not decrease the service performance. The API management can utilize performance test tools to follow best practices in the measuring of quality metrics (De, 2017, p. 164).

API management should further implement system tests. System tests test the software system as a whole (Limited & Office, 2019, p. 158). For this, special-purpose software can be utilized that mocks API consumer requests and validates the received responses (De, 2017, p. 153). Those end-to-end tests ensure the soundness of the platform from an API consumer perspective. For this, the tests should call the API endpoints in a way the actual API consumer would, too (De, 2017, p. 153) . System tests require the most effort but can improve the confidence in the service quality (De, 2017, p. 153).

#### *Related Standards*

- Collaborate and Promote Visibility (Limited & Office, 2019, p. 39)
- Extreme Programming (Beck & Andres, 2004)
- Optimize and Automate (Limited & Office, 2019, p. 39)
- Test-Driven Development (Beck, 2002)

#### *Related Patterns*

SLAs with API providers can be utilized to specify quality standards for backend services. This is documented in *Pattern 6: SLAs with backend providers*.

A solution approach for API security testing is described in *Pattern Candidate 54: Penetration tests*.

The API provider can also assist the API consumer in testing the API integration by providing mock responses as further illustrated in *Pattern Candidate 53: API test values*.

#### *Known Uses*

- C2, C3, C4, C11, C13, C14

### 5.7.4 Pattern 4: Pilot project

#### *Stakeholders*

The following applicants are derived from the study data:

- Backend provider
- Portal provider

The following potential collaborators are derived from the study data:

- API consumer

#### *Concerns*

- Which APIs should be offered?
- How to tailor backend services to APIs that fit the API consumer's needs?
- How to ensure market-fit?
- How to fit the APIs to consumers' requirements?
- How to validate API offerings?
- How to trigger feedback from API consumers?

#### *Example*

The portal provision management captured in C2 manages pilot projects with different groups of API consumers. The portal provider utilizes pilot projects both to trial strategic partnerships and to validate API products. The initial idea for pilot projects comes from partners, third-party organizations, different stakeholders within the API management, or other teams. The development phase and intensity of collaboration is based on a multitude of factors and varies between each project. Each project is owned by a product owner within the portal provider team. The timeline for pilot projects commonly exceeds six months.

#### *Context*

To validate API offerings, the portal provider has to receive customer feedback. An effective and efficient collaboration with external stakeholders such as public API consumers can be hard to achieve [IV3, IV4, IV5].

### *Forces*

Forces that have to be resolved and balanced:

- Collaboration with external stakeholders
- Uncertain needs and problems of API consumers
- Unknown user stories of API consumer
- Required validation of API consumer needs
- Required validation of user stories
- Integration of asynchronous customer feedback into development process

### *Influence Factors*

Table 5.7 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 4: Pilot project*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | **Partner** | **Public** |
| Maturity | Development | **Pilot** | **Production** | |
| Number of API Consumers | **<20** | **>20** | >10,000 | na |
| Type of Platform | **Marketplace** | **Developer Portal** | **Backend APIs** | na |
| Partner Type | **B2B** | B2C | **B2G** | none |
| Monetization | Free | In Product | **Contractual** | Per API Call |

Table 5.7: Influence Factors for Pattern 4

As visible in table 5.7, the solution approach is applied with architectural openness of types partner and public. It can be noted that internal pilot projects within private or group-based environments have not been detected but should also be considered. The platform in focus is documented to be in pilot phases and production. This is a notable difference to the documented maturity level of the alternative *Pattern 11: API product validation* where the API consumer only gains access to the APIs after the release. Additionally, *Pattern 4: Pilot project* is not applied with more than 10,000 API consumers. This can be interpreted in a way that the efforts of pilot projects exceed the advantages in case API management has to support too many API consumers. All three types of platforms have been identified in conjunction with *Pattern 4: Pilot project*. This is another difference to *Pattern 11: API product validation* which was not discovered with backend APIs. Furthermore, pilot projects are utilized together with other businesses and government organizations in B2B and B2G relationships. The monetization is detected to be contractual. Other monetization strategies are not associated with *Pattern 4: Pilot project*.

### *Solution*

Pilot projects describe the closest form of collaboration that was detected with third-party consumers. Pilot experiments are used to test new ideas together with a set of known public or private stakeholders (Billé, 2010). They enable guaranteed and direct feedback before, during, and after each product iteration. Pilot projects follow the ITIL (2019) principles: 'progress iteratively with feedback' and 'collaborate and promote visibility' (Limited & Office, 2019, p. 39).

A pilot project can be initiated by both the API provider and the consumer. After following *Pattern 10: Tailoring APIs to products*, a pilot phase can be started to validate the product ideas. To initiate a pilot project, the API provider can either reach out to potential consumers directly or collaborate with functional teams such as account management in sales and procurement. The development phase can be supported by agile methods that emphasize incremental improvements, ongoing collaboration, and adaptation to an evolving environment (Limited & Office, 2019, p. 40). The pilot project runs until the API provider is satisfied with the state of the API product. It ends with the release of the product.

### *Consequences*

The following benefits are derived from the study data:

- Better understanding of API consumers and their needs
- Customer engagement and buy-ins
- Focus on value creation and relevance for the customer
- Generation of trust, understanding, and greater visibility
- Product quality through continuous improvement
- Learn which use cases are needed and which endpoints are required

The following liabilities are derived from the study data:

- Asynchronous close collaboration can be difficult to manage
- Efforts of collaboration between portal provider and backend provider
- Potentially delayed publishing of APIs
- Project collaboration with third-parties comes with risks such as scope creeping (Kendrick, 2015, p. 52)
- Time consuming process

### *Implementation Details*

A pilot project aims to better understand the requirements of the API consumer. First, the design steps of *Pattern 10: Tailoring APIs to products* can be followed to identify the potential customer, their needs, and use cases. The initial product design can help to pitch the idea to targeted API consumers. The API provider can reach out to the API consumer directly or request support from account management or similar customer and partner facing teams. In case the API consumer starts the initiative, the API provider has to evaluate the business case and technical requirements.

The pilot project can be started with a kick-off meeting or workshop. The collaborators have to get to know each other, create common goals, timelines, manage expectations, and plan the closeness of collaboration. Based on this, the level of collaboration has to be carefully evaluated. Agile methods can offer state-of-the-art practices for iterative and ongoing collaboration (Limited & Office, 2019, p. 40).

### *Related Standards*

- Collaborate and Promote Visibility (Limited & Office, 2019, p. 39)
- Domain Driven Design (Evans, 2003)
- Extreme Programming (Beck & Andres, 2004)
- Focus on Value (Limited & Office, 2019, p. 39)
- Manifesto for Agile Software Development[14]
- Minimal Viable Products (Ries, 2011)
- Pilot Experiment (Billé, 2010)
- Progress Iteratively with Feedback (Limited & Office, 2019, p. 39)
- Rapid Application Development (Kerr & Hunter, 1994)
- Scrum (Takeuchi & Nonaka, 1986)

### *Related Patterns*

The solution approach *Pattern 10: Tailoring APIs to products* offers complementary steps to solve prerequisites.

An alternative or parallel solution approach with loosely coupled collaboration is presented by *Pattern 11: API product validation*.

### *Known Uses*

- C1, C2, C11, C12

[14]http://agilemanifesto.org/

### 5.7.5 Pattern 5: Frontend venture

#### *Stakeholders*

The following applicants are derived from the study data:

- Backend provider
- Portal provider

The following potential collaborators are derived from the study data:

- Sales and marketing

#### *Concerns*

- How to target potential API consumers without technical capabilities?

#### *Example*

The portal provision management captured in case C2 develops frontend tools for its API consumers. Initially, prototypical frontend tools are developed to validate the use case and receive feedback. Based on the results of those pilot phases, the frontend development is delegated to dedicated development teams. This creates ventures that are based on the offered API products and services.

#### *Context*

The integration of APIs into current workflows and tools can come with technical complexity and high costs for the API consumer. Alternatively, potential API consumers might choose to implement the APIs into stand-alone frontends instead. Consumers without technical expertise might request the creation of those frontend tools from the API provider.

#### *Forces*

Forces that have to be resolved and balanced:

- Business case
- Cost-benefit ratio
- Strategic relevance of project

#### *Influence Factors*

Table 5.8 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 5: Frontend venture*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | Partner | Public |
| Maturity | Development | Pilot | Production | |
| Number of API Consumers | <20 | >20 | >10,000 | na |
| Type of Platform | Marketplace | Developer Portal | Backend APIs | na |
| Partner Type | B2B | B2C | B2G | none |
| Monetization | Free | In Product | Contractual | Per API Call |
| Networking Target | Efficiency | Innovation | Channel Extension | Venture |

Table 5.8: Influence Factors for Pattern 5

As visible in table 5.8, the solution approach is applied with architectural openness of types group, partner, and public. Hence, in the documented cases, it is found suitable to interact with external consumers. The platform in focus is documented to be in pilot phases or production. The number of API consumers does not exceed 10,000 in the studied cases. Both backend APIs and developer portals are used in the context of *Pattern 5: Frontend venture*. The type of partner is found to be either B2B or B2G. *Pattern 5: Frontend venture* is not applied with API consumers of type B2C. The monetization strategy is both contractual and API call-based. The networking target varies. I It can be noted that, among others, all known cases shared the networking target 'venture'.

### *Solution*

The development of custom frontends for API consumers is a venture opportunity for the providing organization. In case a potential service consumer requests a custom frontend solution based on the API offerings, the API provider has to calculate the business case of the venture. Forces that should be part of the decision making are cost-benefit ratios, the strategic relevance of the project, and internal resources and capabilities.

The API provider should utilize a pilot phase to iteratively create prototypes. The development of the frontend can be supported by agile methods. A simple first version could offer simple web forms to post data and utilize download buttons to get data from the endpoints.

Continuous validation over the development phase supports the decision making if the business model is worthwhile. In this case, the initial prototype can be delegated to a dedicated development team. The development team then acts as an IT provider to the API consumer while the API platform continues to provide the consumed API services.

### *Variants*

An alternative variant was detected in the studied cases. Early stage API platforms should pivot to provide frontend tools if the target consumers prefer those over API provision. In case that a small group of strategic partners is targeted and those partners would rather be provided with frontend tools altogether, the API provider should change its business model.

The API provider should still use API platforms internally but changes the pilot phase provision management from API provision to frontend provision. A well managed internal platform can later on be opened in case future partners request API services.

### *Consequences*

The following benefits are derived from the study data:

- Customer engagement and buy-ins
- Focus on value creation and relevance for the customer
- Lessons learned from using the own platform as a consumer
- Possibility to venture out of API provisioning
- Promotes API platform and products
- Support for key costumers and strategic partners

The following liabilities are derived from the study data:

- Asynchronous close collaboration can be difficult to manage
- Overhead for API provision management

### *Implementation Details*

Kick-off meetings and workshops together with the requesting API consumer should be used to learn more about the requested features. The budget, needs, and overall workflow should be analyzed. Tools such as the Value Proposition Canvas can support the API provider in the analysis (Osterwalder et al., 2014). Factors like increased publicity or strategic relevance should influence the validity of the business case. In case the business case is approved, a simple first version can be developed in hackathons. The first version should be iteratively improved in a pilot phase. A minimal viable version of the frontend tool can utilize the API consumer's API key and API call-based monetization.

The frontend tool can be provided to the requesting API consumer directly or integrated in the developer portal. If the functionality is part of the developer portal, additional material should be created to explain the functionality to other API consumers. A collaboration with sales and marketing can support the creation of marketing material.

It can be noted that the development of custom frontends requires additional overhead on top of the daily API management. It should only be considered if it does not negatively affect the quality of the API products and services.

### *Related Standards*

- Collaborate and Promote Visibility (Limited & Office, 2019, p. 39)
- Extreme Programming (Beck & Andres, 2004)
- Focus on Value (Limited & Office, 2019, p. 39)
- Manifesto for Agile Software Development[15]
- Minimal Viable Products (Ries, 2011)
- Progress Iteratively with Feedback (Limited & Office, 2019, p. 39)
- Rapid Application Development (Kerr & Hunter, 1994)
- Scrum (Takeuchi & Nonaka, 1986)

### *Related Patterns*

The development of prototypical frontends can be based on pilot projects as described in *Pattern 4: Pilot project*.

### *Known Uses*

- C2, C6, C8

[15]http://agilemanifesto.org/

### 5.7.6 Pattern 6: SLAs with backend providers

#### *Stakeholders*

The following applicants are derived from the study data:

- API management

The following potential collaborators are derived from the study data:

- Backend provider
- Legal

#### *Concerns*

- How to ensure API service quality?
- How to resolve bug reports effectively and transparently?

#### *Example*

C12 documents an API marketplace with a homogeneous set of known API consumers from the same industry. The backend services are provided by a set of heterogeneous third-party API providers. Each API provider aims to provide services for API consumers from that industry. The marketplace offers the API providers services such as discoverability, a marketing platform, and gateway-based utilities like monetization, monitoring, analytics, security, and others. The marketplace provider further ensures the quality of service provided by the backend providers. For this, continuous quality management is required. New API providers are guided through a strict onboarding process. During the onboarding, the API providers create the pricing model for their API offerings. For its services, the marketplace provider is granted a percentage of the price of each API call. The contract between the API provider and marketplace provider includes SLAs that specify the quality of services provided by each party based on a set of defined parameters. The SLAs define the time of availability, the minimum success rate, and incident resolution times. Contractual punishments and escalation levels are documented in case the services do not meet the KPIs.

#### *Context*

API management has to ensure that the offered API services adhere to the quality guarantees given to the API consumers. The API management has to collaborate with the different backend providers to agree upon service levels and quality standards for the API offerings. Quality KPIs can include high availability and low latency for the API consumers.

### *Forces*

Forces that have to be resolved and balanced:

- Business and operational metrics
- Collaboration with third-party providers
- Efficient and effective management of customer requests
- Scalability of support activities
- Service management

### *Influence Factors*

Table 5.9 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 6: SLAs with backend providers*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | Partner | Public |
| Maturity | Development | Pilot | Production | |
| Number of API Consumers | <20 | >20 | >10,000 | na |
| Type of Platform | Marketplace | Developer Portal | Backend APIs | na |
| Partner Type | B2B | B2C | B2G | none |
| Monetization | Free | In Product | Contractual | Per API Call |

Table 5.9: Influence Factors for Pattern 6

Table 5.9 shows that the solution approach is applied with architectural openness of types group, partner, and public. The platform in focus is documented to be in pilot phases and production. The number of API consumers, the type of platform, and the monetization strategy vary across the studied cases. The type of partner is found to be B2B only.

### *Solution*

A SLA identifies all services provided to the service consumer and documents the level of service which is agreed on (Limited & Office, 2019, p. 152). SLAs specify performance parameters from a customer perspective (Limited & Office, 2019, p. 152). API providers can utilize SLAs to define quality standards for the different backend services and thereby hold backend providers accountable for their service provisions. SLAs can also be embedded in contracts that specify further components such as monetization.

The overall goal of SLAs with backend providers is the efficient and effective provision for the API consumer. They document important aspects of the incident management and draw a clear path for error resolution and incident escalation.

### *Variants*

In the studied cases, two different kinds of SLAs between backend providers and the API management providers are documented. The more common one describes the service level that the backend providers agree upon to provide their services for the API platform. A different variant implements the SLA in a way that guarantees the service level upon which the API management provides the infrastructure services for a backend provider. This way around, the API management provider guarantees the service level of the API platform to the backend providers. The two variants are based on different monetization strategies of the API management and API platform. Both can be used together.

### *Consequences*

The following benefits are derived from the study data:

- Efficient and effective management of customer requests
- Focus on value creation for the customer
- Increased transparency for the customer
- Support costumers and strategic partners efficiently

The following liabilities are derived from the study data:

- SLA negotiation efforts

### *Implementation Details*

If the backend services are provided by different teams, subsidiary firms, partnering organizations, or third-party providers, SLAs can be utilized to agree upon a set of quality principles. They can be embedded within a contract. SLAs document the quality parameters of the provided services. Further components, such as monetization, can build upon the SLAs. API management should collaborate with legal to create a reusable framework for service contracts and agreements.

The SLA is negotiated in collaboration by all relevant stakeholders (Limited & Office, 2019, p. 152). A scale of service levels can be utilized to standardize the SLA. API management should investigate if service level management is standardized within its organization.

One important aspect of the SLA is incident management. The agreement should document incident resolution metrics, a clear path for issue escalation, and potential contractual punishment in case quality metrics are not met. All metrics have to be based on measurement standards. The API gateway serves as a single point of truth for monitoring. The formulas and underlying measurements should be part of the SLA to prevent any room for interpretation.

### *Related Standards*

- Collaborate and Promote Visibility (Limited & Office, 2019, p. 39)
- Focus on Value (Limited & Office, 2019, p. 39)
- Incident Management (Limited & Office, 2019, p. 121)
- Optimize and Automate (Limited & Office, 2019, p. 39)
- Service Request Management (Limited & Office, 2019, p. 156)

### *Related Patterns*

A notification system can be utilized to automate reports in case quality parameters are not met. The solution approach is documented in *Pattern Candidate 58: Notification system*.

SLAs can also be singed with the API consumer as described in *Pattern 7: SLAs with API consumers*.

### *Known Uses*

- C4, C5, C6, C8, C9, C11, C12, C13

### 5.7.7 Pattern 7: SLAs with API consumers

***Stakeholders***

The following applicants are derived from the study data:

- API management

The following potential collaborators are derived from the study data:

- API consumer
- Legal

***Concerns***

- How to ensure API service quality?

***Example***

The API management of C5 operates an API platform as a service for a set of subsidiary companies. Each subsidiary can both be backend provider and API consumer. The API management utilizes different SLAs based on the requirements of the subsidiary firm. In some cases, the backend provider pays for the services of the API management. More commonly, the API consumer has to pay for both the backend services and the API management. In latter case, SLAs are used to define the service level that the API consumer is willing to pay for. A higher service level provides faster incident resolution times for the API consumer. Most API consumers negotiate a resolution time that lays within standard business hours. Mission critical API offerings and infrastructure services can utilize a seven days a week, 24 hours a day, service level agreement.

***Context***

The API provision management has to ensure that the offered API services adhere to the quality guarantees given to the API consumers. Those quality guarantees have to be agreed upon with the API consumers.

### *Forces*

Forces that have to be resolved and balanced:

- Business and operational metrics
- Collaboration with third-party providers
- Efficient and effective management of customer requests
- Scalability of support activities
- Service management

### *Influence Factors*

Table 5.10 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 7: SLAs with API consumers*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | Partner | Public |
| Maturity | Development | Pilot | Production | |
| Number of API Consumers | <20 | >20 | >10,000 | na |
| Type of Platform | Marketplace | Developer Portal | Backend APIs | na |
| Partner Type | B2B | B2C | B2G | none |
| Monetization | Free | In Product | Contractual | Per API Call |

Table 5.10: Influence Factors for Pattern 7

Table 5.10 shows that the solution approach has been applied with architectural openness of types group, partner, and public. The platform in focus is documented to be in pilot phases and production. The number of API consumers and the type of platform vary across the studied cases. In contrast to *Pattern 6: SLAs with backend providers*, the partner type is not limited to businesses. All three partner types have been identified. The monetization strategy has been documented as product-based, contractual, and API call-based. The API is not offered for free in the associated cases. This is another difference to *Pattern 6: SLAs with backend providers*.

### *Solution*

SLAs document the services offered to the consumers and specify the level of service which is agreed on (Limited & Office, 2019, p. 152). They describe performance parameters from a customer perspective (Limited & Office, 2019, p. 152). SLAs between the API provider and API consumers document the responsibilities of the API management and act as quality guarantees of the API provision.

API providers can utilize SLAs to define quality standards for the customer support and infrastructure provision. The SLA documents important aspects of the incident management and thus, draws a clear path for error handling and incident escalation. SLAs are one component of a contract. Other components can specify the price model of offered services. The overall goal of SLAs is transparency and a measurable agreement about the service quality.

### *Consequences*

The following benefits are derived from the study data:

- Efficient and effective management of customer requests
- Focus on value creation for the customer
- Increased transparency for the customer
- Support costumers and strategic partners efficiently

The following liabilities are derived from the study data:

- SLA negotiation efforts

### *Implementation Details*

The API management provides a set of services around the offered API products. Those services include: customer support, infrastructure maintenance (e.g. API gateway provision), API brokering (e.g. portal provision), and others. The gateway is the core infrastructure platform of the API management. Thus, the gateway provider team provides mission critical services for the API management and plays a key role in the SLA definition.

The SLA is negotiated in collaboration by all relevant stakeholders (Limited & Office, 2019, p. 152). A scale of service levels can be utilized to standardize the SLA. The API management should investigate if service level management is standardized within its organization. Additionally, API management should collaborate with legal to create a reusable framework for service contracts and agreements. The API consumer and API provider have to agree on the terms and conditions of the API provision contract. The SLA defines the responsibilities of the API provider. For this, the service level has to be agreed on. A higher service level is connected to more costs for the API management but might allow for higher charges towards the API consumer.

One important aspect of the **SLA,!** (**SLA,!**) is incident management. The agreement should document incident resolution metrics, a clear path for issue escalation, and potential contractual punishment in case quality metrics are not met. All metrics have to be based on measurement standards. The API gateway serves as a single point of truth for monitoring. The formulas and underlying measurements should be part of the SLA to prevent any room for interpretation. If one API consumer demands to have a high service level, all other API consumer will profit from the higher quality agreement. This is the case because if the API gateway has an outage, it has an outage for all services. One high service level agreement translates to quicker resolution times for all API consumers.

### *Related Standards*

- Collaborate and Promote Visibility (Limited & Office, 2019, p. 39)
- Focus on Value (Limited & Office, 2019, p. 39)
- Incident Management (Limited & Office, 2019, p. 121)
- Optimize and Automate (Limited & Office, 2019, p. 39)
- Service Request Management (Limited & Office, 2019, p. 156)

### *Related Patterns*

A notification system can be utilized to automate reports in case quality parameters are not met. The solution approach is documented in *Pattern Candidate 58: Notification system*.

SLAs can also be singed with backend providers as described in *Pattern 6: SLAs with backend providers*.

### *Known Uses*

- C1, C2, C3, C5, C6, C8, C10, C12, C13

### 5.7.8 Pattern 8: Data clearance

*Stakeholders*

The following applicants are derived from the study data:

- API management

The following potential collaborators are derived from the study data:

- API governance
- Backend provider
- Legal

*Concerns*

- How to onboard a new API product onto the platform?
- How to tailor backend services to APIs that fit the API consumer′s needs?

*Example*

The portal provision management captured in C2 collaborates with a central data clearing office. Each data point that is meant to be published to third-party API consumers has to go through a data clearance process. The data clearing ensures that all API endpoints follow compliance and privacy regulations.

*Context*

Offering an internal API for external use comes with legal, compliance, strategic, and other requirements. API management has to ensure that all data that is made accessible for third-parties is cleared.

*Forces*

Forces that have to be resolved and balanced:

- Data clearance and compliance
- Ease of integration
- Separation of concerns
- Service orchestration
- User experience

## *Influence Factors*

Table 5.11 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 8: Data clearance*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | **Group** | **Partner** | **Public** |
| Maturity | Development | Pilot | **Production** | |
| Number of API Consumers | **<20** | **>20** | **>10,000** | **na** |
| Type of Platform | Marketplace | **Developer Portal** | Backend APIs | **na** |
| Monetization | **Free** | **In Product** | **Contractual** | **Per API Call** |

Table 5.11: Influence Factors for Pattern 8

As visible in table 5.11, the pattern is identified for APIs offered to group, public and partnering API consumers. The solution approach is documented in production only. The number of API consumers varies across the associated cases. It is applied to developer portals only. All variants of monetization were identified in the context of this solution approach. It is notable that the influence factors of this solution approach are shared with *Pattern 9: API orchestration layer*.

## *Solution*

Internal backend services can be reused for external API offerings. The publication of internal capabilities requires a data clearing process. Data clearance ensures that new offered endpoints comply with legal, strategic, and other requirements. Additionally, the process can be used to tailor new offered capabilities towards customer needs.

Before backend services can be published, each new data points that is made public has to be analyzed and cleared. Each data point has to follow privacy regulations. A close collaboration with legal and data privacy teams is required to clear the data.

Some data points might have strategic value or are part of an internal business model. The publication of the backend services requires a close collaboration with API governance, backend provider, and eventual additional data owners. For instance, if the backend service aggregates data from several other internal backend services, those backend providers should be consulted as well.

Additionally, public API endpoints have to be aligned to the needs of the API consumers. Public API consumers have a different perspective than internal API consumers. They may lack domain knowledge or company-specific jargon. The alignment can be improved by filtering, renaming, aggregating, and adjusting the response data.

### *Consequences*

The following benefits are derived from the study data:

- Focus on value and relevance for the customer
- Improves developer experience
- Reduces integration effort

The following liabilities are derived from the study data:

- Compliance and data clearing efforts

### *Implementation Details*

Data clearance is required to comply with privacy regulations such as the General Data Protection Regulation (GDPR). A close collaboration with privacy experts and legal teams is recommended to avoid pitfalls and potential costly privacy breaches. If backend services provide user specific data or protected resources, authorization frameworks are required to grant the end user the ability to authorize third-party applications access to their data. Frameworks, such as OAuth 2.0[16], serve as authorization layers that provide state-of-the-art authorization and authentication processes. Required authorization layers should be implemented on an API gateway level to reuse and centralize security and authentication.

Further strategic alignment with the backend provider and data owner teams is necessary to estimate the strategic value of the data. The estimated business model of the new API products has to be compared to the value of offered data sets. If possible, strategic data points can be filtered from the external API endpoints.

Further filtering is required to reduce unnecessary complexity for the API consumer. Data points not relevant to the public API consumers' needs should be deleted from the API responses. Next, the data keys used to name each data point should be evaluated. Internal jargon should be renamed to make the data understandable for external API consumers. Each data key should follow a consistent naming convention that is used across all public API products. Subsequently, API consumers might lack further contextual knowledge. Data points can be aggregated or derived from several different sources to create new data points that might be redundant but reduce the complexity for the API consumer. The ease of integration and reduction of complexity for the API consumer should be seen as the main goals.

[16]https://tools.ietf.org/html/rfc6749

### *Related Standards*

- API Facade Pattern (De, 2017, p. 86)
- Domain Driven Design (Evans, 2003)
- Extreme Programming (Beck & Andres, 2004)
- Focus on Value (Limited & Office, 2019, p. 39)
- Manifesto for Agile Software Development[17]
- Minimal Viable Products (Ries, 2011)
- Progress Iteratively with Feedback (Limited & Office, 2019, p. 39)
- Rapid Application Development (Kerr & Hunter, 1994)

### *Related Patterns*

*Pattern Candidate 31: Data clearing office* documents a centralization effort for the collaboration required for data clearance.

*Pattern 8: Data clearance* should be integrated in an early stage within the API product development. *Pattern 10: Tailoring APIs to products* offers an overall product design process that can be followed.

*Pattern 9: API orchestration layer* describes an abstraction layer that orchestrates internal backend services before it returns an aggregated response to external API consumers. The orchestration layer can be utilized to fulfill data clearing requirements such as data filtering.

### *Known Uses*

- C2, C3, C4, C5, C8, C9, C10

[17]http://agilemanifesto.org/

### 5.7.9 Pattern 9: API orchestration layer

#### *Stakeholders*

The following applicants are derived from the study data:

- API management

#### *Concerns*

- How to onboard new API products onto the platform?
- How to tailor backend services to APIs that fit the API consumer′s needs?
- How to aggregate backend services?

#### *Example*

The API management documented in C9 implements an API orchestration layer to invoke several internal backend services. The API orchestration layer is managed and maintained by its own API provider team. It utilizes GraphQL as a technology to aggregate data from different sources and provides a GraphQL endpoint for the API consumer-facing API platform.

#### *Context*

API providers utilize a set of backend services to interact with the requested capabilities (De, 2017, p. 23). Usually, endpoints are not mapped in a one-to-one relationship to the API consumer. Instead, an API consumer invokes an abstract API that reflects the user′s point of view. Internally, the API call invokes several backend services to collect data or execute functions (De, 2017, p. 23). The API consumer then receives an aggregated response from the API platform (De, 2017, p. 23).

#### *Forces*

Forces that have to be resolved and balanced:

- Data clearance and compliance
- Ease of integration
- Separation of concerns
- Service orchestration
- User experience

## *Influence Factors*

Table 5.12 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 9: API orchestration layer*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | Partner | Public |
| Maturity | Development | Pilot | Production | |
| Number of API Consumers | <20 | >20 | >10,000 | na |
| Type of Platform | Marketplace | Developer Portal | Backend APIs | na |
| Monetization | Free | In Product | Contractual | Per API Call |

Table 5.12: Influence Factors for Pattern 9

As visible in table 5.12, the pattern is identified for APIs offered to group, public and partnering API consumers. We think that *Pattern 9: API orchestration layer* can also be applied to private APIs. This solution approach is documented in production only. The number of API consumers varies across the associated cases. The pattern is applied to developer portals only. All variants of monetization were identified in the context of this solution approach. It is notable that the influence factors of this solution approach are shared with *Pattern 8: Data clearance*.

## *Solution*

An API orchestration layer enables the aggregation of several internal backend services to one API call for the API consumer (De, 2017, p. 23). It can be used to collect and augment the capabilities of different internal backend services to fit the API consumer's needs (De, 2017, p. 23). The API orchestration layer thereby supports the tailoring of API products that fit the user stories of the API consumers.

Furthermore, it can be used to filter data from internal backend services that is not meant to be accessible for external API consumers. The API orchestration layer can be utilized to support data clearing processes.

## *Consequences*

The following benefits are derived from the study data:

- Focus on value and relevance for the customer
- Improved developer experience
- Improved performance for API consumers
- Reduced latency for API consumers
- Reduced integration effort

The following liabilities are derived from the study data:

- Additional effort for the creation and maintenance of federation layer

### *Implementation Details*

An API orchestration layer can be implemented in various ways. It can be implemented within the API gateway directly (De, 2017, p. 87). Alternatively, a proxy backend service can be created that is invoked by the API gateway and used as an orchestration layer.

Furthermore, the orchestration layer can utilize several different technologies. De (2017) emphasizes that the API gateway and orchestration layer should be kept light-weight and stateless (De, 2017, p. 23). To achieve this, the orchestration layer should utilize state-of-the-art technologies such as REST or GraphQL to aggregate data, call backend services, and return the responses. Thus, the orchestration layer should be built following best practices in the development of backend services to avoid common pitfalls such as stateful service provision.

### *Related Standards*

- API Facade Pattern (De, 2017, p. 86)
- Design Thinking (Plattner et al., 2010)
- Domain Driven Design (Evans, 2003)
- Extreme Programming (Beck & Andres, 2004)
- Focus on Value (Limited & Office, 2019, p. 39)
- Minimal Viable Products (Ries, 2011)
- Rapid Application Development (Kerr & Hunter, 1994)

### *Related Patterns*

An API orchestration layer supports the creation of tailored endpoints based on the API consumers' needs. The tailoring of API products is described in *Pattern 10: Tailoring APIs to products*.

Software libraries can be used to wrap API calls on the API consumer side into function calls. This further improves the API consumer experience. The creation of software libraries is documented in *Pattern 15: Software libraries*.

### *Known Uses*

- C2, C3, C4, C5, C9, C10

### 5.7.10 Pattern 10: Tailoring APIs to products

#### *Stakeholders*

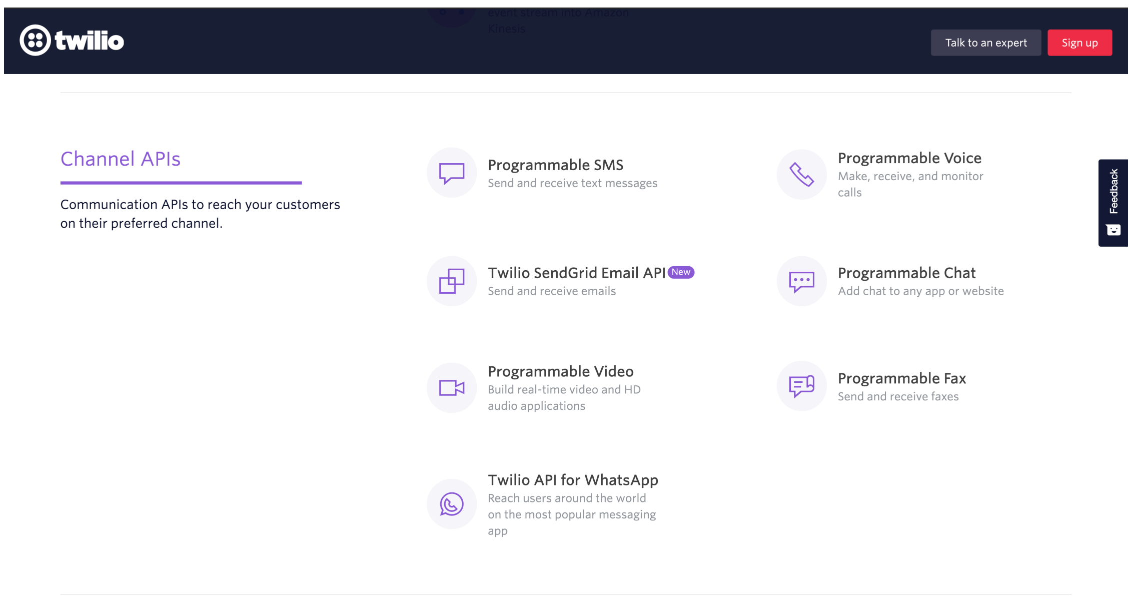
The following applicants are derived from the study data:

- Backend provider
- Portal provider

#### *Concerns*

- Which APIs should be offered?
- How to tailor backend services to APIs that fit the API consumer's needs?
- How to ensure market-fit?
- Who will be using the APIs?
- How to fit the APIs to consumers' requirements?

#### *Example*



Channel APIs

Communication APIs to reach your customers on their preferred channel.

Programmable SMS
Send and receive text messages

Programmable Voice
Make, receive, and monitor calls

Twilio SendGrid Email API New
Send and receive emails

Programmable Chat
Add chat to any app or website

Programmable Video
Build real-time video and HD audio applications

Programmable Fax
Send and receive faxes

Twilio API for WhatsApp
Reach users around the world on the most popular messaging app

Figure 5.6: Twilio Product Overview Page

The following example illustrates the end result of this solution approach. Twilio[18] offers web APIs for API consumers to develop customer experience solutions. Figure 5.6 shows the product overview page[19] of the Twilio website. The user is given an overview of available products and included user stories. Thus, the web APIs are tailored into several independent products. Each product contains a set of specified user stories and is easily differentiable from the other products. For instance, the product 'Programmable SMS' captures the two user stories 'Send text messages' and 'Receive text messages'.

### *Context*

The portal provider and backend provider have to decide what services should be offered through the developer portal.

### *Forces*

Forces that have to be resolved and balanced:

- Uncertain needs and problems of API consumers
- Unknown API consumers
- Unknown user stories of API consumer

### *Influence Factors*

Table 5.13 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 10: Tailoring APIs to products*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | Partner | Public |
| Maturity | Development | Pilot | Production | |
| Number of API Consumers | <20 | >20 | >10,000 | na |
| Type of Platform | Marketplace | Developer Portal | Backend APIs | na |
| API Consumer Heterogeneity | Homogeneous | Heterogeneity | | |
| Monetization | Free | In Product | Contractual | Per API Call |

Table 5.13: Influence Factors for Pattern 10

As visible in table 5.13, the management of API products is identified for APIs offered to public and partnering third-party consumers. We think that the product approach and user-centric design can also be applied to internal APIs. This solution approach is documented in production only. Further, the tailoring of API products is observed in organizations with varying numbers of API consumers.

[18]https://www.twilio.com
[19]https://www.twilio.com/products

Both marketplaces and developer portals are found to market APIs as products. Backend APIs without a developer portal were not discovered to offer API products. All variants of monetization were identified in the context of this solution approach. In the studied cases, product APIs are not offered for free.

### *Solution*

Related APIs should be bundled to API products (De, 2017, p. 149). Each API product fulfills specific business needs and is purchased independently by the API consumers (De, 2017, p. 149). Thus, price models are created on a product level.

Treating web API offerings as products enables the usage of product design processes. They support the creation of offerings that are designed towards identified customer needs (Medjaoui et al., 2018). Each product is specifically tailored to provide the most value for a group of target customers (Limited & Office, 2019, p. 12). The focus on value includes the design of user experience for the customers (Limited & Office, 2019, p. 39). Thus, ease of use, quality, effectiveness, and further factors are part of the API product design (Limited & Office, 2019, p. 148). All those factors should be considered in order to tailor API products that fit customer needs and provide value.

Product design processes are based on agile methods and dedicated tooling. First, the API provider has to identify potential consumers and develop hypotheses about how the products can fit their needs (Limited & Office, 2019, p. 12). Following an agile approach, identified problems and needs can then be translated to user stories[20]. Each user story provides specific value to a customer. Next, related user stories are bundled into a product.

### *Consequences*

The following benefits are derived from the study data:

- Better understanding of API consumers and their needs
- Focus on value creation for API consumers
- Learn which use cases are needed and thus, which endpoints are required
- Provides a clear path to API marketing and user story-based documentation
- Identification of potential API consumers

The following liabilities are derived from the study data:

- Additional documentation and marketing information required to communicate product approach to the consumer (Weir & Nemec, 2019).
- Additional effort of product management and ownership for API provider (Weir & Nemec, 2019)

[20]http://www.agilemodeling.com/artifacts/userStory.htm

### *Implementation Details*

API providers should follow product design processes to create their products. First, the target consumer has to be identified (Limited & Office, 2019, p. 41). Next, the needs and problems of the consumer have to be uncovered (Medjaoui et al., 2018). Finally, user stories have to be created to reflect the use cases that the consumer is tying to achieve. Use cases and user stories can further be derived and combined into products. One product can be used by several groups of consumers if it covers the use cases of each group (Limited & Office, 2019, p. 12).

To support the tailoring of API products, best practices and standard tools can be utilized such as the Business Model Canvas or Value Proposition Canvas (Barquet et al., 2011; Osterwalder et al., 2014). In relation to agile methods, the tailoring of API products can be understood as a design sprint, followed by rapid prototyping.

Products and user stories are based on web APIs and service offerings. Several products and user stories might use the same web APIs. Unused web APIs can be removed from the developer portal. Each product has to be designed with a corresponding price model in mind. The business model of an API product should be based on the associated business goals of the API platform and product (De, 2017, p. 14).

Like every software product, the responsibility of an API product should be assigned to a product owner. The API product owner is responsible for the quality and delivery of the bundled APIs (De, 2017, p. 184).

### *Related Standards*

- Business Model Canvas (Barquet et al., 2011)
- Design Thinking (Plattner et al., 2010)
- Domain Driven Design (Evans, 2003)
- Domain Story Telling [21]
- Extreme Programming (Beck & Andres, 2004)
- Focus on Value (Limited & Office, 2019, p. 39)
- Minimal Viable Products (Ries, 2011)
- Rapid Application Development (Kerr & Hunter, 1994)
- User Stories
- Value Proposition Design (Osterwalder et al., 2014)

[21]https://domainstorytelling.org/

### *Related Patterns*

*Pattern 9: API orchestration layer* illustrates how multiple backend services can be aggregated and altered. An orchestration layer enables adjustments to internal backend services to fit the needs of the API consumers and thus, serves as a perquisite to tailor API products.

*Pattern 8: Data clearance* documents requirements and best practices that should be followed when an internal backend service is exposed to external API consumers.

*Pattern 11: API product validation* describes the validation of products both before and after they are published on the API platform. In this context, *Pattern 10: Tailoring APIs to products* should also be utilized in conjunction with *Pattern 12: Idea backlog*.

After products have been created successfully and validated in collaboration with API consumers, the marketing and documentation of API user stories and products should be approached. They are part of the user experience of the product and have to be tailored to provide the most value towards the customer. This is covered in *Pattern 13: API product documentation*, *Pattern 14: Cookbooks*, and *Pattern 17: Role-based marketing*.

Next to the product dimension, the API management also provides services around the API product provision. The service dimension includes centralized management, brokering, and provision services. Those services can be part of the product pricing or treated separately. *Pattern 6: SLAs with backend providers* and *Pattern 7: SLAs with API consumers* document how the quality of those services can be standardized.

### *Known Uses*

- Twilio, Stripe
- C2, C10, C12, C13

### 5.7.11 Pattern 11: API product validation

***Stakeholders***

The following applicants are derived from the study data:

- Portal provider

***Concerns***

- How to validate API offerings?

***Example***

C3 describes an API platform targeting public API consumers while C4 documents API services provided to partner organizations. Both platforms are managed by the same API provider team. In both cases, a similar process is used to initiate collaboration and trigger customer feedback.

The public API is tied to a software product. Each user of the software product has free access to the public API. Hence, the list of all possible customers is known to the API provider. Each feature request or contact inquiry that reaches the API provider through contact forms, customer support, or other channels, is documented and references the requesting entity. This way, the API provider knows which (potential) public API consumer or partner organization is interested in which topics and contact can be initiated. Additionally, product ideas can be validated against the list of requested features.

When a new API product is published to one of the developer portals, the API management notifies interested consumers via newsletter. To issue follow-up requests or report problems, the API consumer can again utilized the contact forms, customer support, or other channels. This way, potential customers are informed of the API development and are motivated to give feedback.

***Context***

An effective and efficient collaboration with external stakeholders such as public API consumers can be hard to achieve [IV3, IV4, IV5]. Validation aims to ensure the offerings fit the costumers' needs (Limited & Office, 2019, p. 158).

### *Forces*

Forces that have to be resolved and balanced:

- Required validation of API consumer needs
- Required validation of user stories
- Communication with external stakeholders
- Integration of asynchronous customer feedback into development process

### *Influence Factors*

Table 5.14 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 11: API product validation*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | [✓ Group] | [✓ Partner] | [✓ Public] |
| Maturity | Development | Pilot | [✓ Production] | |
| Number of API Consumers | [✓ <20] | [✓ >20] | [✓ >1000] | na |
| Type of Platform | [✓ Marketplace] | [✓ Developer Portal] | Backend APIs | na |
| API Consumer Heterogeneity | [✓ Homogeneous] | [✓ Heterogeneous] | | |
| Partner Type | [✓ B2B] | [✓ B2C] | [✓ B2G] | none |
| Monetization | [✓ Free] | [✓ In Product] | [✓ Contractual] | [✓ Per API Call] |

Table 5.14: Influence Factors for Pattern 11

As visible in table 5.14, the solution approach has been applied with architectural openness of types: group, partner, and public. Thus, in the documented cases, it is found suitable to interact with external consumers. In all cases, the platform has been in production. The number of API consumers, partner type, heterogeneity, and monetization strategy varies across documented cases. This is a notable difference to the alternative solution approach documented in *Pattern 3: API testing strategy*. Both marketplaces and developer-based API management utilize the described product validation approach.

### *Solution*

The portal provider can utilize customer feedback to validate API products. The ITIL (2019) stresses the importance of customer feedback and illustrates the perception of products or services from a customer's point of view (Limited & Office, 2019, p. 47). The developer portal provides communication channels to the API consumers. For instance, a developer portal can include a contact form which allows the consumer to issue different kinds of contact inquiries. Each inquiry contains information about potential or current API consumers and their concerns. Feature requests or business inquiries can be used to either initiate or validate user stories and tailored API products.

After a new API product or service has been launched, incoming feedback reflects the customers' opinions and needs. It can be used to incrementally improve the offerings. This constant validation and iterative development follows agile methodologies (Limited & Office, 2019, p. 40).

#### *Variants*

*Pattern 11: API product validation* can be achieved by utilizing different communication channels and approaches. In the following, identified variants will be listed. Feedback can be communicated through contact forms, service desk software, or other channels such as email or phone to a customer support team. If the number of API consumers is low and can be managed efficiently, e-mail or chat-based communication can be sufficient. In cooperation with internal API consumers or strategic partners, meetings and workshops can be utilized to achieve validation.

#### *Consequences*

The following benefits are derived from the study data:

- Better understanding of API consumers and their needs
- Focus on value creation for the customer
- Improved product quality through continuous improvement
- Learn which use cases are needed and thus, which endpoints are required
- Potential customer engagement

The following liabilities are derived from the study data:

- Close collaboration between portal provider and backend provider required
- Efforts of managing customer requests required

#### *Implementation Details*

Each incoming customer request has to be categorized and managed in an efficient and effective manner. The feedback can be used to create or validate product ideas. A product idea or the result of a product design process can be validated against the list of requested features. A product idea that is not reflected in any current requests can still be valid. Potentially, the list of requests is small or from a different group of customers than targeted by the new product. The chance of market-fit, however, is higher if the product is pre-validated.

After a product launch, new incoming requests such as bug reports and related feature requests can be used for the next iteration of validation. An agile development process can be used to iteratively improve the published product.

### *Related Standards*

- Manifesto for Agile Software Development[22]
- Extreme Programming (Beck & Andres, 2004)
- Rapid Application Development (Kerr & Hunter, 1994)
- Focus on Value (Limited & Office, 2019, p. 39)
- Progress Iteratively with Feedback (Limited & Office, 2019, p. 39)
- Minimal Viable Products (Ries, 2011)

### *Related Patterns*

In conjunction with *Pattern 11: API product validation*, *Pattern 18: Newsletter* describes how to notify potentially interested consumers about new product launches. This can trigger additional feedback. To manage incoming feature requests efficiently and effectively, *Pattern 12: Idea backlog* can be utilized.

An alternative or parallel solution approach with closer collaboration is presented by *Pattern 4: Pilot project*.

Portal providers can take advantage of *Pattern Candidate 28: Service validation workshops*, *Pattern Candidate 35: Hackathons*, and *Pattern Candidate 36: Pilot workshops* to validate API product and service ideas. Especially within organization boundaries direct communication can be easier to achieve and enhance the collaboration between the participants.

### *Known Uses*

- C2, C3, C4, C5, C12, C13

[22]http://agilemanifesto.org/

### 5.7.12 Pattern 12: Idea backlog

#### *Stakeholders*

The following potential collaborators are derived from the study data:

- Portal provider

#### *Concerns*

- Which APIs should be offered?
- How to tailor backend services to APIs that fit the API consumer's needs?
- How to ensure market-fit?
- Who will be using the APIs?
- How to fit the APIs to consumers' requirements?
- How to validate API offerings?

#### *Example*

As laid out in the example of *Pattern 11: API product validation*, the API management of C3 and C4 utilizes incoming customer requests to initiate and validate new API offerings. An idea backlog is used to store all new feature ideas and change requests derived from customer requests. Each incoming request is entered into the idea backlog. The idea backlog enables the API management to cluster customer requests, count how many times a features is requested, and what variants of the same request might exist. New offerings can either be created based on ideas out of the idea backlog or validated against the list of requested features.

#### *Context*

The portal provider and backend provider have to decide what services should be offered through the developer portal.

### *Forces*

Forces that have to be resolved and balanced:

- Communication with external stakeholders
- Integration of asynchronous customer feedback into development process
- Mixing different groups of consumers
- Uncertain needs and problems of API consumers
- Unknown API consumers
- Unknown user stories of API consumer
- Required validation of API consumer needs
- Required validation of user stories

### *Influence Factors*

Table 5.15 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 12: Idea backlog*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | **Partner** | **Public** |
| Maturity | Development | Pilot | **Production** | |
| Number of API Consumers | <20 | **>20** | **>10,000** | na |
| Type of Platform | Marketplace | **Developer Portal** | Backend APIs | na |
| Monetization | **Free** | **In Product** | **Contractual** | **Per API Call** |

Table 5.15: Influence Factors for Pattern 12

As visible in table 5.15, the management of API products is identified for APIs offered to public and partnering third-party consumers. The solution approach is documented in production only. The number of API consumers is above 20 in the related cases. Additionally, the solution approach is only applied within the management of developer portals. All variants of monetization were identified in the context of this solution approach.

### *Solution*

An idea backlog offers a simple and intuitive way to manage incoming feature and change requests. Each request is translated into a ticket within the backlog. It contains information about the requesting (potential) API consumer and a description. Additional fields within the ticket software can be utilized to further enhance the information. Each ticket also provides meta data such as the time and date of the ticket creation that can aid in the analysis of requests. The request can be treated as customer feedback. Thus, the idea backlog can be used to initiate new product ideas or to validate planned or current offerings.

### *Consequences*

The following benefits are derived from the study data:

- Better understanding of API consumers and their needs
- Focus on value creation for the customer
- Identification of consumers opens opportunities to collaboration
- Identification of potential API consumers
- Learn which use cases are needed and thus, which endpoints are required

The following liabilities are derived from the study data:

- Additional effort of request management for portal provider
- Close collaboration between portal provider and backend provider required
- Divided ownership between backend provider and portal provider

### *Implementation Details*

The idea backlog is a dynamic list of tickets similar to the product backlog in Scrum. Common ticket-based project management software offers all tools required to create an idea backlog. If an agile development approach is utilized, the same software can be utilized which is already in use to manage tickets. The idea backlog can be created as an additional backlog within the team space.

Each incoming request should be checked for novelty and then entered into the idea backlog. This requires the creation of a new ticket. It is important that no contextual data and information is lost during ticket creation. Requests can furthermore be translated into one or more user stories. Each user story can be added to the idea backlog separately. In case the requested changes are already reflected within a ticket, a counter within the ticket can be incremented. Different variants of the same request should either be merged to one ticket or saved separately. This ensures that no information is lost.

Tags and categories can be utilized to cluster incoming requests. Possible categories could include 'new API', 'new endpoint', and 'change request'. Additional product or service-based tags can link tickets to a specific product or service. This way, the idea backlog can be sorted in several ways and better utilized to gain an overview of the distribution of requests. Prioritized tickets can then be taken over into the product backlog or directly into the current scope of work. If not done before, the request ticket should be translated into a task or user story based on the best-practices utilized in the team.

***Related Standards***

- Design Thinking (Plattner et al., 2010)
- Extreme Programming (Beck & Andres, 2004)
- Focus on Value (Limited & Office, 2019, p. 39)
- Manifesto for Agile Software Development[23]
- Minimal Viable Products (Ries, 2011)
- Progress Iteratively with Feedback (Limited & Office, 2019, p. 39)
- Rapid Application Development (Kerr & Hunter, 1994)
- Scrum (Takeuchi & Nonaka, 1986)

***Related Patterns***

The idea backlog can be utilized to validate new product ideas as described in *Pattern 11: API product validation*.

***Known Uses***

- C2, C3, C4

[23]http://agilemanifesto.org/

### 5.7.13 Pattern 13: API product documentation

#### *Stakeholders*

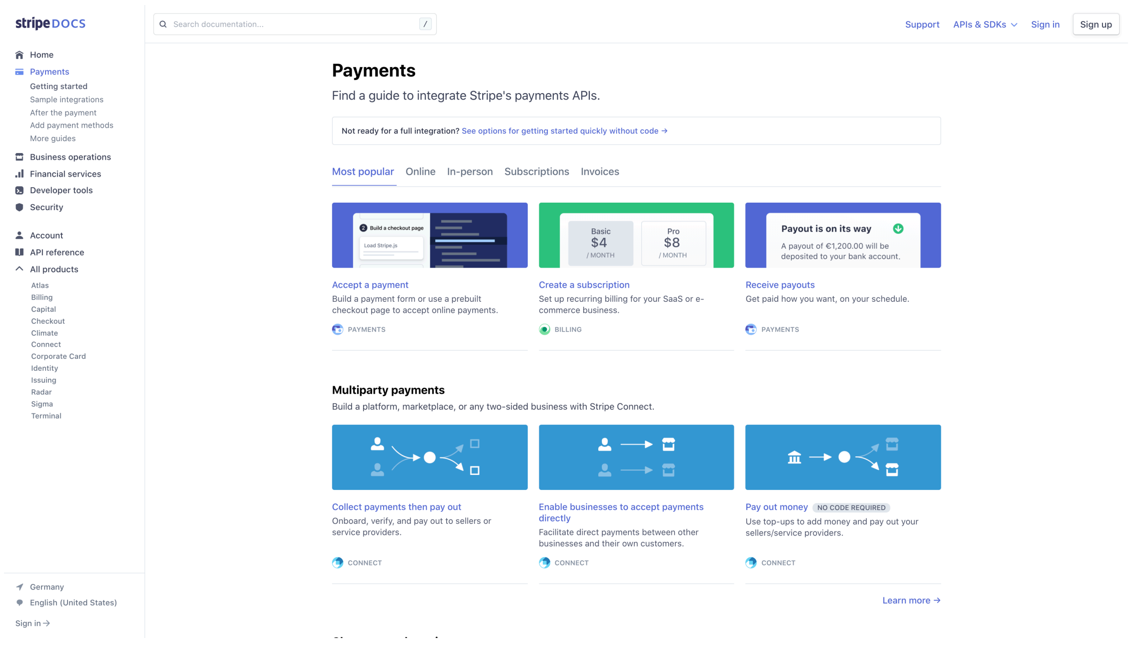
The following applicants are derived from the study data:

- Portal provider

#### *Concerns*

- How to document API products?
- How to support developers with API integration?
- How to communicate with API consumers?

#### *Example*



stripe DOCS

Search documentation... /

Support · APIs & SDKs · Sign in · Sign up

Home
Payments
Getting started
Sample integrations
After the payment
Add payment methods
More guides
Business operations
Financial services
Developer tools
Security
Account
API reference
All products
Atlas
Billing
Capital
Checkout
Climate
Connect
Corporate Card
Identity
Issuing
Radar
Sigma
Terminal

Germany
English (United States)
Sign in →

**Payments**

Find a guide to integrate Stripe's payments APIs.

Not ready for a full integration? See options for getting started quickly without code →

Most popular · Online · In-person · Subscriptions · Invoices

Accept a payment
Build a payment form or use a prebuilt checkout page to accept online payments.
PAYMENTS

Create a subscription
Set up recurring billing for your SaaS or e-commerce business.
BILLING

Receive payouts
Get paid how you want, on your schedule.
PAYMENTS

**Multiparty payments**

Build a platform, marketplace, or any two-sided business with Stripe Connect.

Collect payments then pay out
Onboard, verify, and pay out to sellers or service providers.
CONNECT

Enable businesses to accept payments directly
Facilitate direct payments between other businesses and their own customers.
CONNECT

Pay out money NO CODE REQUIRED
Use top-ups to add money and pay out your sellers/service providers.
CONNECT

Learn more →

Figure 5.7: Stripe's API Integration Guides for its Payments Product

Stripe[24] develops a suite of payment products. The documentation structures products as top level navigation entries[25]. Each product documentation offers

[24]https://stripe.com/en-de
[25]https://stripe.com/docs

its own landing page with an overview of user story-based integration guides. These guides act as second level navigation entries and lead the user to story-based API documentations. Figure 5.7 illustrates how the documentation for the payments product is split into user stories[26].

### *Context*

As described in sections API Management and Roles and Stakeholders, the portal provider has to effectively support the application developers with social and developer boundary resources. In the context of *Pattern 10: Tailoring APIs to products*, the developer portal has to be adjusted to present and document tailored API products.

### *Forces*

Forces that have to be resolved and balanced:

- Domain knowledge transfer
- Ease of integration
- User experience
- User friendly technical documentation

### *Influence Factors*

Table 5.16 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 13: API product documentation*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | **Partner** | **Public** |
| Maturity | Development | Pilot | **Production** | |
| Number of API Consumers | **<20** | **>20** | **>10,000** | na |
| Type of Platform | **Marketplace** | **Developer Portal** | Backend APIs | na |
| API Consumer Heterogeneity | **Homogeneous** | **Heterogeneity** | | |
| Monetization | Free | **In Product** | **Contractual** | **Per API Call** |

Table 5.16: Influence Factors for Pattern 13

Both *Pattern 10: Tailoring APIs to products* and *Pattern 13: API product documentation* are used in conjunction and share the same influence factors. As visible in table 5.16, the documentation of API products is identified for APIs offered to public and partnering third-party consumers. This solution approach is documented in production only. Further, the documentation of API products is observed in organizations with varying numbers of API consumers. Both marketplaces and developer portals are found to document APIs as products. Backend

[26]https://stripe.com/docs/payments

APIs without a developer portal were not discovered to offer API products. All variants of monetization are identified in the context of this solution approach. In the studied cases, product APIs are not offered for free.

### *Solution*

The API documentation should follow a user-centric structure. This improves the user experience and thus, increases the value of the developer portal (Limited & Office, 2019, p. 39, 43). As defined in *Pattern 10: Tailoring APIs to products*, APIs are tailored into products. Each product thereby consists of one or more user stories. Products and user stories provide a clear structure for a consumer-centric API documentation (Medjaoui et al., 2018). Each product is documented separately. This ensures transparency about the offerings of each product. The landing page of the documentation is used to provide an overview of all products and serves the product discovery. A user friendly grid representation of the products improves the visibility. This can be further supported by a search function. Each product entry references a product-specific documentation. These product-specific pages are used to list all user stories of the product and are structured similarly to the main landing page. The user can view product-related information and select between user stories. Each user story provides APIs with documentation based on the requirements of the user story. This ensures focus on the developer's needs and removes unrelated complexity.

### *Consequences*

The following benefits are derived from the study data:

- Focus on value and relevance for the customer
- Reduction and management of complexity
- Promote visibility and discovery of products and user stories
- Improve developer experience
- Improve orientation throughout the API documentation

The following liabilities are derived from the study data:

- Curating efforts for high quality documentation
- Continuous maintenance of documentation required

### *Implementation Details*

First, the list of products has to be created. The search bar has to be placed visibly. A grid-based list can be used, supported by illustrations and icons. If the set of products exceeds a simple list representation, e.g. the user has to scroll to look through all products, filters and sub-menus can be utilized to improve visibility. Each product-specific page should provide a list of user stories. Additionally, a sticky navigation menu shows the current position within the documentation and provides quick links to other products. This supports quick pathways between products and also improves orientation. Each user story documents all required steps from a consumer-perspective.

### *Related Standards*

- Focus on Value (Limited & Office, 2019, p. 39)

### *Related Patterns*

The solution approach *Pattern 10: Tailoring APIs to products* serves as a prerequisite for *Pattern 13: API product documentation*. The documentation of user stories is further described in *Pattern 14: Cookbooks*.

### *Known Uses*

- Stripe, Twilio
- C2, C10, C12, C13

### 5.7.14 Pattern 14: Cookbooks

#### *Stakeholders*

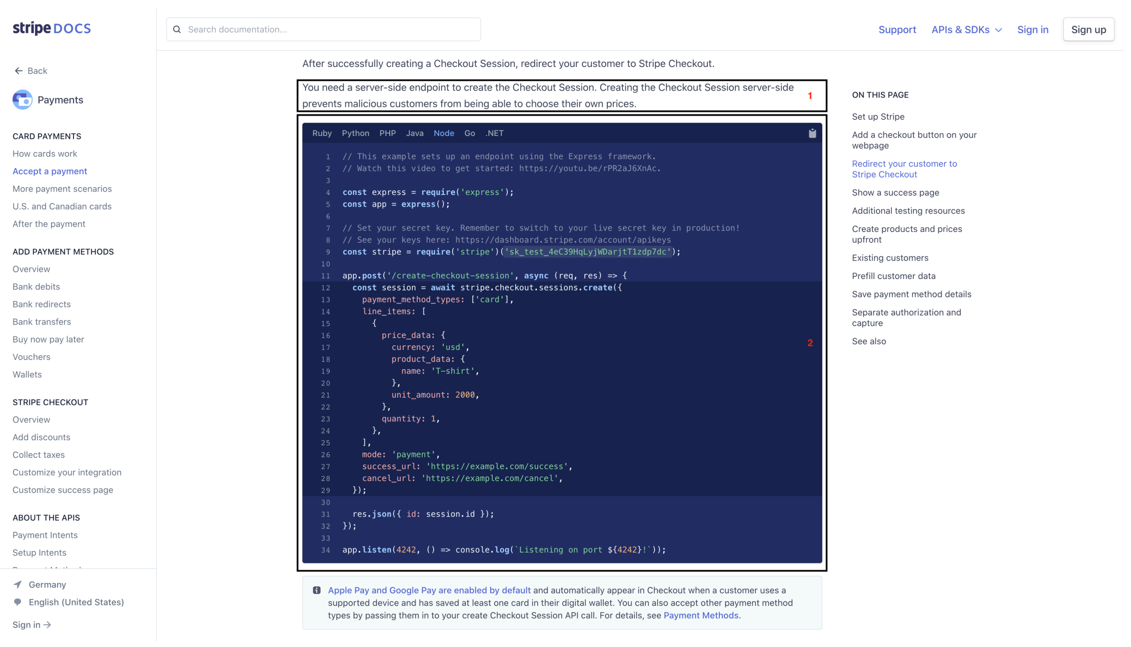
The following applicants are derived from the study data:

- Portal provider

#### *Concerns*

- How to document API products?
- How to support developers with API integration?
- How to communicate with API consumers?

#### *Example*



After successfully creating a Checkout Session, redirect your customer to Stripe Checkout.

You need a server-side endpoint to create the Checkout Session. Creating the Checkout Session server-side prevents malicious customers from being able to choose their own prices.

```
// This example sets up an endpoint using the Express framework.
// Watch this video to get started: https://youtu.be/rPR2aJ6XnAc.

const express = require('express');
const app = express();

// Set your secret key. Remember to switch to your live secret key in production!
// See your keys here: https://dashboard.stripe.com/account/apikeys
const stripe = require('stripe')('sk_test_4eC39HqLyjWDarjtT1zdp7dc');

app.post('/create-checkout-session', async (req, res) => {
  const session = await stripe.checkout.sessions.create({
    payment_method_types: ['card'],
    line_items: [
      {
        price_data: {
          currency: 'usd',
          product_data: {
            name: 'T-shirt',
          },
          unit_amount: 2000,
        },
        quantity: 1,
      },
    ],
    mode: 'payment',
    success_url: 'https://example.com/success',
    cancel_url: 'https://example.com/cancel',
  });

  res.json({ id: session.id });
});

app.listen(4242, () => console.log(`Listening on port ${4242}!`));
```

Apple Pay and Google Pay are enabled by default and automatically appear in Checkout when a customer uses a supported device and has saved at least one card in their digital wallet. You can also accept other payment method types by passing them in to your create Checkout Session API call. For details, see Payment Methods.

Figure 5.8: Stripe's 'Accept Payment' Integration Guide

Following the example described in *Pattern 13: API product documentation*, each product within Stripe's documentation offers a set of integration guides. Figure 5.8 captures the guide for the user story 'accept a payment'. Framed sections of the figure illustrate different parts of the step-by-step documentation of the user story. Section one serves as an instance for additional information provided in each step. Section two presents the user with client-side code snippets. Each code snippet is offered in seven different programming languages. It illustrates how to utilize the offered APIs for a specific use case. A navigation menu on the right-hand side provides the user an overview of the steps of the integration guide. It can be noted that Stripe also offers a raw API specification[27]. It provides a holistic documentation for each endpoint that can be used for further reference.

### *Context*

API documentations support the application developer with the integration of the offered APIs. Raw API specifications offer a raw technical view of the endpoints. Additional resources can provide user-centric approaches.

### *Forces*

Forces that have to be resolved and balanced:

- Domain knowledge transfer
- Ease of integration
- User experience
- User friendly technical documentation

### *Influence Factors*

Table 5.17 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 14: Cookbooks*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | Partner | Public |
| Maturity | Development | Pilot | Production | |
| Number of API Consumers | <20 | >20 | >10,000 | na |
| Type of Platform | Marketplace | Developer Portal | Backend APIs | na |
| API Consumer Heterogeneity | Homogeneous | Heterogeneous | | |
| Monetization | Free | In Product | Contractual | Per API Call |

Table 5.17: Influence Factors for Pattern 14

As visible in table 5.17, the solution approach has been applied with architectural openness of types group, partner, and public. in In all associated cases, the platform has been in production. The number of API consumers varies across documented cases. It is notable that this approach has been followed for consumers of type partner and public only for a number of API consumers greater than 20. The consumer heterogeneity is captured as solely heterogeneous. Product-based,

[27]https://stripe.com/docs/api

contractual, and API call-based monetization are associated with *Pattern 14: Cookbooks*.

### *Solution*

Cookbooks are recipe-like, step-by-step integration guides. They describe the API integration from a consumer perspective. As documented in the solution approach *Pattern 13: API product documentation*, this improves the user experience. Each cookbook should fit into the overall structure of the documentation. For this, a product and user story approach can be used. Furthermore, each user story is documented separately and can be followed in isolation. This supports the developers in the implementation of their targeted use cases. Cookbooks can include references to related user stories, prerequisites, or complementary resources. The overall goal, however, is to guide the user from start to end without the need to follow additional documentation. Each step within the user story provides context, reasoning, and implementation details. Code examples are used to provide consumer-side code snippets that the developer can copy paste (Mulloy, 2012, p. 31). After the completion of all steps, the integration into the application should be accomplished.

### *Consequences*

The following benefits are derived from the study data:

- Focus on value and relevance for the customer
- Improved developer experience
- Reduction and management of complexity

The following liabilities are derived from the study data:

- Continuous maintenance of documentation required
- Curating efforts for high-quality documentation

### *Implementation Details*

Each user story is translated into a cookbook. The recipe-like integration guide splits the user story into distinct activities that the user has to complete. Each step should be designed with the overall cookbook in mind. A sticky navigation menu can be utilized that shows the current position within the cookbook and offers quick links to the other steps. Each recipe step contains all required information and code snippets to complete the specific integration activity. For this, the required domain knowledge has to be communicated in an approachable way. A good balance of complexity reduction and completeness has to be found. Client-side code examples and user-friendly representations of different information improves the experience. For instance, reoccurring icons, colors, and borders can be used to communicate different types of information such as warnings, security information, or pitfalls.

### *Related Standards*

- Focus on Value (Limited & Office, 2019, p. 39)

### *Related Patterns*

The solution approach *Pattern 13: API product documentation* offers an overall structure in which cookbook-based integration guides can be embedded. Both patterns work in complement.

Cookbooks can be further enhanced with library-based code examples. The creation of such libraries is documented in *Pattern 15: Software libraries*.

### *Known Uses*

- Stripe, Twilio
- C2, C3, C10

### 5.7.15 Pattern 15: Software libraries

#### *Stakeholders*

The following applicants are derived from the study data:

- Backend provider
- Portal provider

#### *Concerns*

- How to support a growing number of API consumers?
- How to communicate with API consumers?
- How to document API products?
- How to support developers with API integration?

#### *Example*

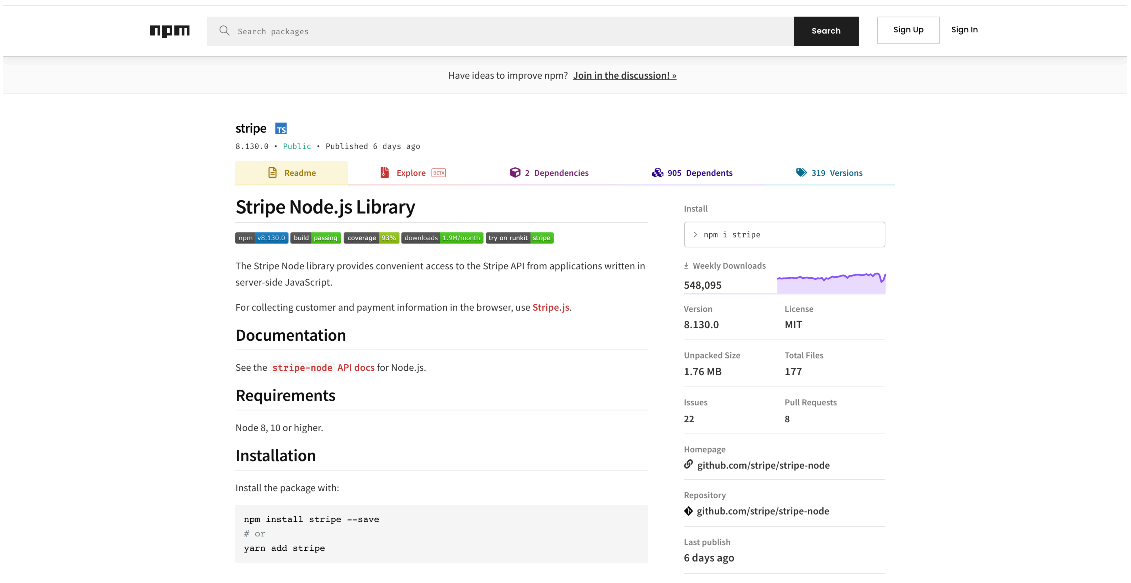
Stripe offers its API consumers software libraries in commonly used programming languages. Each library is open sourced and published on GitHub[28]. The libraries implement REST API calls to Stripe's REST API and can be integrated directly into the API consumers' applications . Thus, the application provider does not interact with the API calls directly but indirectly through the library functions. The libraries are accessible through commonly used package managers. Figure 5.9 shows the Stripe JavaScript library on npm, a commonly used package manager for JavaScript packages. On npm, the library is averaging more than 500,000 weekly downloads[29].

Stripe's product-based documentation, which is described in the examples of *Pattern 13: API product documentation* and *Pattern 14: Cookbooks*, illustrates the library code in its code examples. For instance, the code snippet given in figure 5.8 shows the 'redirect your customer to Stripe Checkout' integration step written with the stripe-node library. Figure 5.8 also illustrates that the same code example is available in all common programming languages.

#### *Context*

API providers aim to reduce the complexity and ease the integration of their offerings (Boudreau, 2011). The API provider can provide API consumers code utilities to support the API integration.

[28]https://github.com/stripe
[29]https://www.npmjs.com/package/stripe



Have ideas to improve npm? Join in the discussion! »

stripe TS

8.130.0 • Public • Published 6 days ago

Stripe Node.js Library

The Stripe Node library provides convenient access to the Stripe API from applications written in server-side JavaScript.

For collecting customer and payment information in the browser, use Stripe.js.

Documentation

See the `stripe-node` API docs for Node.js.

Requirements

Node 8, 10 or higher.

Installation

Install the package with:

```
npm install stripe --save
# or
yarn add stripe
```

Install

```
> npm i stripe
```

| | |
|---|---|
| Weekly Downloads | 548,095 |
| Version | 8.130.0 |
| License | MIT |
| Unpacked Size | 1.76 MB |
| Total Files | 177 |
| Issues | 22 |
| Pull Requests | 8 |
| Homepage | github.com/stripe/stripe-node |
| Repository | github.com/stripe/stripe-node |
| Last publish | 6 days ago |

Figure 5.9: Stripe's JavaScript Library on npm

### *Forces*

Forces that have to be resolved and balanced:

- Domain knowledge transfer
- Ease of integration
- User experience
- User friendly technical documentation

### *Influence Factors*

Table 5.18 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 15: Software libraries*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | Partner | Public |
| Maturity | Development | Pilot | Production | |
| Number of API Consumers | <20 | >20 | >10,000 | na |
| Type of Platform | Marketplace | Developer Portal | Backend APIs | na |

Table 5.18: Influence Factors for Pattern 15

Table 5.18 shows that the solution approach has been applied with architectural openness of types partner and public. The platform in focus is documented to be in production only. The studied cases show a number of API consumers greater than 10,000. Developer portals are used in the context of *Pattern 15: Software libraries*. It can be noted that the pattern is not applied by European API provider organizations.

### *Solution*

Software libraries provide utilities to ease the integration of APIs. As described in section SDKs, software libraries enable re-use, separation of concerns, and allow the integration of shared code within different application architectures. Overall, they improve the software quality of the API consumer application and reduce complexity by simplifying the application source code (Hou & Yao, 2011). API providers can offer software libraries in common programming languages. They are used as wrappers around the APIs and allow the API consumer to invoke them (De, 2017). They can be provided together with the API documentation. A well managed developer portal can tailor the documentation and libraries in a way that they work together and create a user-centric view of the integration process.

### *Consequences*

The following benefits are derived from the study data:

- Focus on value and relevance for the customer
- Improves developer experience
- Improves speed of adoption (Mulloy, 2012, p. 31)
- Promotes API products (Mulloy, 2012, p. 31)
- Reduces integration effort (Mulloy, 2012, p. 31)

The following liabilities are derived from the study data:

- Additional documentation required based on library code examples
- Severely additional effort for the creation and maintenance of libraries

### *Implementation Details*

The development of software libraries requires additional capabilities and adds more complexity to the API provision management. It is a decision that should be evaluated carefully based on a set of influence factors. To avoid additional overhead in the development phase of the API platform, the software library development can be initiated after the platform has been validated in production and accumulated a high number of API consumers. Furthermore, the complexity of the API endpoints should be evaluated. If the APIs is well-documented, follows best-practices, and the volume of support requests is low, software libraries are potentially not required (Mulloy, 2012, p. 31 ). A high number of support requests however is a good indicator that the integration complexity has to be reduced.

If the decision is made to create complementary software libraries for the API offerings, the API provider has to decide if the libraries should be auto-generated or created manually. Section Open API Specification describes how documentation and consumer-side code can be generated automatically based on the OAS. Tools such as Swagger Codegen[30] enable the generation of SDKs based for APIs defined with the OAS. If the API in question already follows the OAS (OpenAPI Initiative, 2020), auto-generating library code might be a good first step. The auto-generated code can be enhanced further. Custom-made libraries enable the API provider to implement domain knowledge and domain dependent utilities into the libraries.

Next, the documentation has be updated to reflect the library code. The installation process of the libraries has to be documented for each supported programming language. Code snippets should be adapted to illustrate library code directly. Additionally, the library code can be published on common code hosting platforms to enable collaboration with the API consumers [IV10]. This can further improve the discoverability of the libraries and promote the API offerings.

### *Related Standards*

- Business Model Canvas (Barquet et al., 2011)
- Design Thinking (Plattner et al., 2010)
- Focus on Value (Limited & Office, 2019, p. 39)
- Value Proposition Design (Osterwalder et al., 2014)

[30]https://swagger.io/tools/swagger-codegen/

### Related Patterns

Libraries can implement tailored API products. The tailoring of API products is described in *Pattern 10: Tailoring APIs to products*. The product-based documentation described in *Pattern 13: API product documentation* and *Pattern 14: Cookbooks* can be used complementary to *Pattern 15: Software libraries*.

As described in the implementation details, the library code can be published on common repository management site. The open sourcing of SDKs is described in *Pattern Candidate 27: Open-source SDK*.

Sample projects can provide API consumers guidance in the implementation of client-side libraries. This is further detailed in *Pattern Candidate 47: Sample projects*.

### Known Uses

- Stripe, Twilio
- C10

### 5.7.16 Pattern 16: Integration partner management

#### *Stakeholders*

The following applicants are derived from the study data:

- Portal provider

The following potential collaborators are derived from the study data:

- API consumer
- Integration partner

#### *Concerns*

- How to support potential API consumers without technical capabilities?
- How to engage business roles of the API consumer?
- How to market API offerings to non-technical roles?
- How to communicate with API consumers?

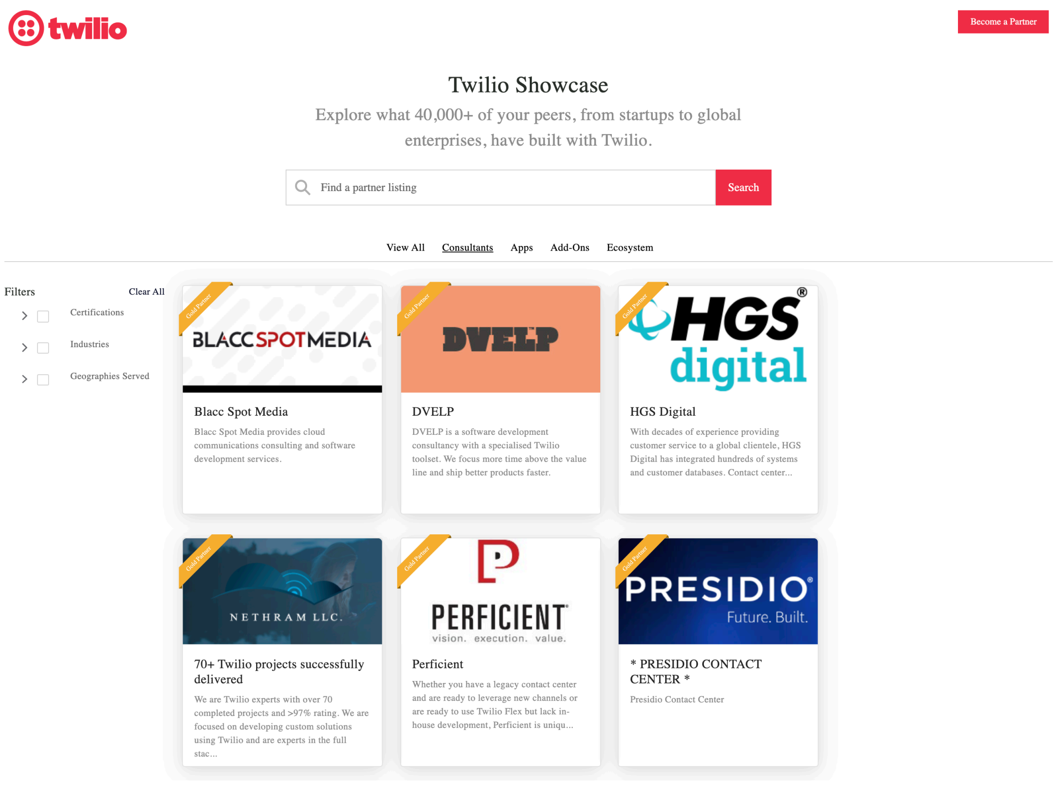
#### *Example*

Twilio maintains a list of partners that are promoted on the website's showcase page[31]. Under the tab 'consultants', API consumers can view a list of curated integration partners. These firms offer the implementation and resale of Twilio services[32]. On the showcase page, the API consumer can filter the list of curated consultants by certification, industry, and geographic area. Thus, API consumers without technical capabilities can select an IT consulting firm or agency that is preselected by Twilio.

#### *Context*

Potential API consumers without technical expertise have to hire external developers or agencies to integrate the APIs into their workflows. The potential customer might be interested in the offered services but does not have the capabilities to integrate the APIs on their own.

[31]https://showcase.twilio.com/s/
[32]https://www.twilio.com/partner-solutions/become-a-partner



Twilio Showcase

Explore what 40,000+ of your peers, from startups to global enterprises, have built with Twilio.

Find a partner listing

View All Consultants Apps Add-Ons Ecosystem

Filters Clear All

Certifications

Industries

Geographies Served

Blacc Spot Media

Blacc Spot Media provides cloud communications consulting and software development services.

DVELP

DVELP is a software development consultancy with a specialised Twilio toolset. We focus more time above the value line and ship better products faster.

HGS Digital

With decades of experience providing customer service to a global clientele, HGS Digital has integrated hundreds of systems and customer databases. Contact center...

70+ Twilio projects successfully delivered

We are Twilio experts with over 70 completed projects and >97% rating. We are focused on developing custom solutions using Twilio and are experts in the full stac...

Perficient

Whether you have a legacy contact center and are ready to leverage new channels or are ready to use Twilio Flex but lack in-house development, Perficient is uniqu...

* PRESIDIO CONTACT CENTER *

Presidio Contact Center

Figure 5.10: Twilio Integration Partners

### *Forces*

Forces that have to be resolved and balanced:

- Support for API consumers
- Quality requirements for integration partners
- Efforts of curating list of high quality integration partners

### *Influence Factors*

Table 5.19 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 16: Integration partner management*.

As visible in table 5.19, the solution approach has been applied with architectural openness of types partner and public. In all cases, the platform has been in production. It can be noted that this approach has been followed only for a number of API consumers greater than 10,000. It is applied solely to developer portals. The consumer heterogeneity is heterogeneous in the identified cases. Product-based, contractual, and API call-based monetization strategies are in use.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | Partner | Public |
| Maturity | Development | Pilot | Production | |
| Number of API Consumers | <20 | >20 | >10,000 | na |
| Type of Platform | Marketplace | Developer Portal | Backend APIs | na |
| API Consumer Heterogeneity | Homogeneous | Heterogeneous | | |
| Monetization | Free | In Product | Contractual | Per API Call |

Table 5.19: Influence Factors for Pattern 16

### *Solution*

API consumers without technical capabilities can be supported with a list of curated integration partners. The creation of a partnership model improves stability within the API ecosystem and enables third-parties to create value (Jansen & Cusumano, 2012). The partner management consists of several continuous activities. Potential integration partners have to be validated and added to the list of offered partners. The API consumer can further provide feedback about the experience with the partners. This adds additional validation. The curated list of partners has to be marketed towards the API consumer.

The developer portal is utilized to support partner management. It is used both to invite potential partners and to market qualified integration partners to the API consumers. A dedicated page for integration partners can be used to explain the concepts to both potential partners and API consumers. It should contain links to a web form for partner applications and to the actual list of curated integration partners. To improve discoverability, the list view of integration partners should be supported by geographical and industry filter options. Each entry within the list view contains a short description that briefly introduces the IT service provider and boasts the firm's experience. Each entry links to the external web page of the integration partner.

### *Variants*

The validation process of potential integration partners can be extended by offering or requiring certifications. Willing integration partners are trained and certified by the API provider. The certification ensures the competencies of the integration partner.

***Consequences***

The following benefits are derived from the study data:

- Customer engagement and buy-ins
- Enhanced customer experience
- Focus on value creation and relevance for the customer
- Increased robustness of ecosystem
- Promote API platform and products
- Stimulate activity in the ecosystem
- Support costumers and strategic partners

The following liabilities are derived from the study data:

- Additional content to be maintained on the developer portal
- Additional partner management tasks

***Implementation Details***

The curated list of integration partners has to be marketed on the developer portal. For improved visibility, it can be mentioned on the landing page. Here, a short chapter about integration partners can inform both potential partners and interested API consumers. The portal provider can cooperate with sales and marketing to incorporate the information into the developer portal.

To streamline the application process for potential integration partners, the developer portal can further offer dedicated contact information or a contact form. An application form can include the following fields:

- Contact information
- Company-related information, e.g. name, size, country, legal entity
- The regions in which services are provided
- Specifications about the offerings, e.g. monetization
- Number of successful integrations
- Target customers, e.g. company size

The API provider can also identify potential integration partners by asking existing API consumers how they integrated the APIs into their systems. If the API consumer used externals to integrate the APIs, the provider should ask about the consumer's experience. If it is positive, the provider can reach out to the integrating entity. This process can be done iteratively to validate offered integration partners and to find new ones. For this, the API provider can utilize surveys (Limited & Office, 2019, p. 154).

### *Related Standards*

- Focus on Value (Limited & Office, 2019, p. 39)

### *Related Patterns*

The marketing material on the developer portal can further be enhanced by the solution approach described in *Pattern 17: Role-based marketing*.

### *Known Uses*

- Twilio
- C3, C8, C10, C13

### 5.7.17 Pattern 17: Role-based marketing

#### *Stakeholders*

The following applicants are derived from the study data:

- Portal provider

The following potential collaborators are derived from the study data:

- Sales and marketing

#### *Concerns*

- How to offer a high-quality user experience for both business and developer roles?
- How to engage business roles of the API consumer?
- How to market API offerings to non-technical roles?
- How to market API offerings to application developers?
- How to communicate with API consumers?

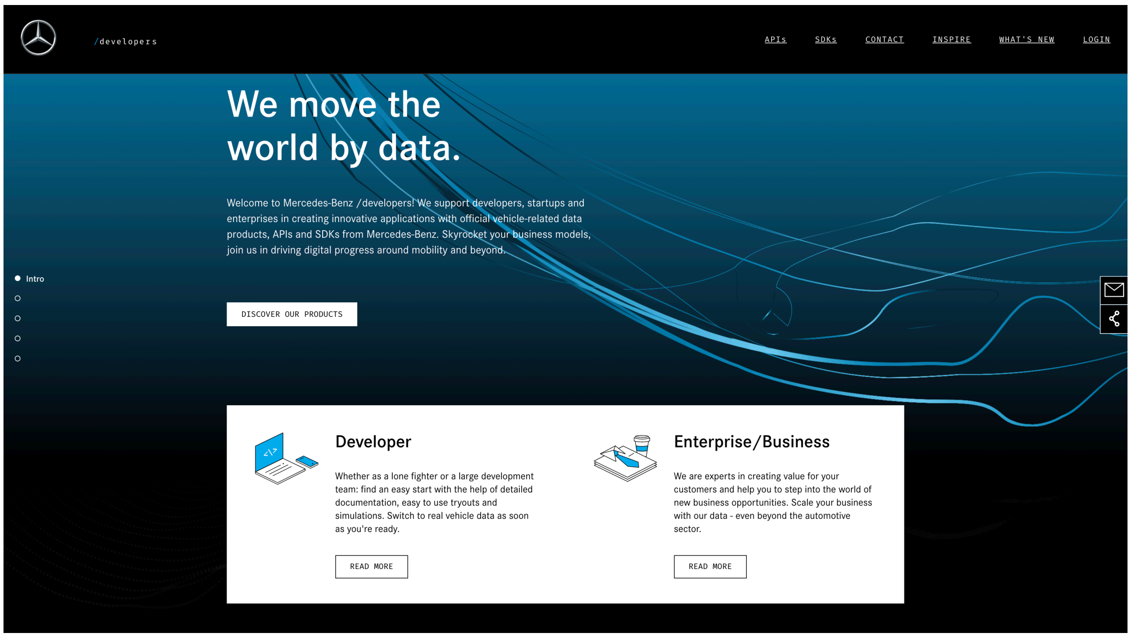
#### *Example*

Mercedes-Benz offers API products and services to partner and third-party developers on its developer portal[33]. Figure 5.11 shows the landing page. The users are immediately asked to select a role, either developer or enterprise/business. Thereby, the portal provider assures that both user types are only one click away from customized marketing material. If the user decides to navigate to 'read more' on the enterprise/business entry, business-related marketing material is presented. The dedicated business page[34] introduces the platform and lists use cases and success stories. Each use case is motivated by a short description and links to a use case details page that further describes the specific product.

#### *Context*

The API consumer organization consists of different teams, roles, and stakeholders. The API documentation targets primary the application developer but other stakeholders are also involved in the buy-decision that has to be made before integrating third-party APIs. Procurement, product owners, and other management and business roles should also be provided with a high-quality user experience.

[33]https://developer.mercedes-benz.com/
[34]https://developer.mercedes-benz.com/home/business



We move the world by data.

Welcome to Mercedes-Benz /developers! We support developers, startups and enterprises in creating innovative applications with official vehicle-related data products, APIs and SDKs from Mercedes-Benz. Skyrocket your business models, join us in driving digital progress around mobility and beyond.

Developer

Whether as a lone fighter or a large development team: find an easy start with the help of detailed documentation, easy to use tryouts and simulations. Switch to real vehicle data as soon as you're ready.

Enterprise/Business

We are experts in creating value for your customers and help you to step into the world of new business opportunities. Scale your business with our data - even beyond the automotive sector.

Figure 5.11: Role Selection on Mercedes Developers

### *Forces*

Forces that have to be resolved and balanced:

- Different roles require additional user stories for their user experience
- API marketing material for non-technical stakeholders

### *Influence Factors*

Table 5.20 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 17: Role-based marketing*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | **Partner** | **Public** |
| Maturity | Development | Pilot | **Production** | |
| Number of API Consumers | **<20** | **>20** | >10,000 | **na** |
| Type of Platform | **Marketplace** | **Developer Portal** | Backend APIs | na |
| Monetization | Free | In Product | **Contractual** | **Per API Call** |

Table 5.20: Influence Factors for Pattern 17

As visible in table 5.20, the pattern is applied for APIs offered to public and partnering third-party consumers and is documented in production only. *Pattern*

*17: Role-based marketing* has not been identified with more than 10,000 API consumers. Both marketplaces and developer portals are found to apply the solution approach. In the context of this pattern, both contractual and API call-based monetization strategies are found. It can be concluded that the additional efforts of curating additional marketing material might only make sense if the number of API consumers is high and a direct monetization strategy is used.

### *Solution*

Role-based marketing divides marketing material in the developer portal to target different roles of users. To improve the user experience of non-technical stakeholders, the portal provider has to describe the APIs as products and define marketing material for non-developers. Furthermore, the portal provider needs to ensure that the additional information does not hinder the user journey of developers that try to visit the documentation or technical specifications of the API. For that, the portal provider has to develop clear user stories for the different types of users of the portal. The provider has to identify the needs of those stakeholders and define user stories that account for their activities. Each user story has to be implemented into the developer portal.

### *Consequences*

The following benefits are derived from the study data:

- Enhance customer experience
- Focus on value creation and relevance for the customer
- Promote API platform and products
- Promote visibility and discovery of products and use cases
- Reduction and management of complexity
- Support costumers and strategic partners

The following liabilities are derived from the study data:

- Increased complexity of developer portal

### *Implementation Details*

The identification of related stakeholders and their pain points can be accomplished by utilizing standard tools such as the Value Proposition Canvas (Osterwalder et al., 2014). Personas can be used to illustrate the different user roles. To design an accessible and informative user experience for both technical and non-technical users, user stories can be utilized. Each user story should be defined towards a specific user role and describe their goals and activities on the developer portal.

The landing page can be used to provide the user a list of possible roles. Each role should carry a short description about the kind of information that it is associated with. A first step can be to distinguish the two roles developer and business.

The portal provider should collaborate with sales and marketing teams to create the marketing material for non-technical stakeholder. These can include use case-based product descriptions, success stories, listings of partners, and listings of API consumer firms that use the products and services.

### *Related Standards*

- Business Model Canvas (Barquet et al., 2011)
- Design Thinking (Plattner et al., 2010)
- Focus on Value (Limited & Office, 2019, p. 39)
- Value Proposition Design (Osterwalder et al., 2014)

### *Related Patterns*

*Pattern 10: Tailoring APIs to products* draws a clear path for the creation of marketing material and can be used as a base line for *Pattern 17: Role-based marketing*.

API consumers without technical capabilities can further be supported by *Pattern 16: Integration partner management*. A common type of marketing material is the success story. Success stories are covered in *Pattern 19: Customer success stories*.

### *Known Uses*

- Mercedes-Benz
- C2, C8, C12

### 5.7.18 Pattern 18: Newsletter

#### *Stakeholders*

The following applicants are derived from the study data:

- Portal provider

The following potential collaborators are derived from the study data:

- Sales and marketing

#### *Concerns*

- How to engage business roles of the API consumer?
- How to market API offerings to non-technical roles?
- How to market API offerings to application developers?
- How to notify API consumers about new API products?
- How to communicate with API consumers?

#### *Example*

As laid out in the example of *Pattern 11: API product validation*, C3 describes an API platform targeting public API consumers while C4 documents API services provided to partner organizations. Both platforms are managed by the same API provider team. In both cases, newsletters are utilized to promote new API offerings to current and potential API consumers.

The public API is tied to a software product. Each user of the software product has free access to the public API. Additionally, as documented in the examples of *Pattern 11: API product validation* and *Pattern 12: Idea backlog*, the API management keeps track of incoming feature requests and their requesters. Hence, the list of all possible customers and potentially interested partners is known to the API provider. The API management includes a marketing role that curates content for both public and partnering API consumers. New API offerings and status updates are promoted through in-product and conventional newsletters.

#### *Context*

API providers have to promote and sell API products and services to current and potential API consumers. Marketing efforts have to target different roles within the API consumer organisation. New product offerings have to be promoted.

### *Forces*

Forces that have to be resolved and balanced:

- API marketing material for non-technical stakeholders
- Different roles require additional user stories for their user experience
- Visibility of API changes

### *Influence Factors*

Table 5.21 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 18: Newsletter*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | **Partner** | **Public** |
| Maturity | Development | Pilot | **Production** | |
| Number of API Consumers | <20 | **>20** | **>10,000** | na |
| Type of Platform | Marketplace | **Developer Portal** | Backend APIs | na |
| Monetization | **Free** | **In Product** | **Contractual** | **Per API Call** |

Table 5.21: Influence Factors for Pattern 18

As visible in table 5.21, the pattern is applied to API platforms that offer APIs to public and partnering third-party consumers. In the studied cases, the solution approach is documented in production only. The number of API consumers is above 20 in the related cases. Additionally, it is only applied within the management of developer portals. All variants of monetization are identified in the context of this solution approach.

### *Solution*

Newsletters can be utilized to promote changes to the API offerings. Different newsletters can be tailored for different groups of API consumers. Continuous collection contact information in combination with feature requests and inquiries enables further individualization of newsletters.

### *Consequences*

The following benefits are derived from the study data:

- Promote visibility and discovery of products and use cases
- Enhance customer experience
- Promote API platform and products
- Focus on value creation and relevance for the customer

The following liabilities are derived from the study data:

- Overhead for the portal provider
- Requirement of a marketing role or close collaboration with sales and marketing

### *Implementation Details*

Common sales and marketing tools can be utilized to manage contact information, costumer lists, and newsletter management. A marketing role can be created within the portal provider teams. Additionally, a close collaboration with sales and marketing teams can support the tailoring of marketing content and newsletter creation.

### *Related Standards*

- Design Thinking (Plattner et al., 2010)
- Focus on Value (Limited & Office, 2019, p. 39)

### *Related Patterns*

API products and user stories provide a clear path to marketing. The creation of API products is documented in *Pattern 10: Tailoring APIs to products*. *Pattern 11: API product validation* and *Pattern 12: Idea backlog* document complimentary solution approaches that support the collection of API consumer contact information and related contextual data.

Changelogs, as documented in *Pattern Candidate 57: Changelogs*, can be used in addition to newsletter. For instance, a newsletter can link to the changelog to provide a more technical view of the promoted changes.

### *Known Uses*

- C2, C3, C4

### 5.7.19 Pattern 19: Customer success stories

#### *Stakeholders*

The following applicants are derived from the study data:

- Portal provider

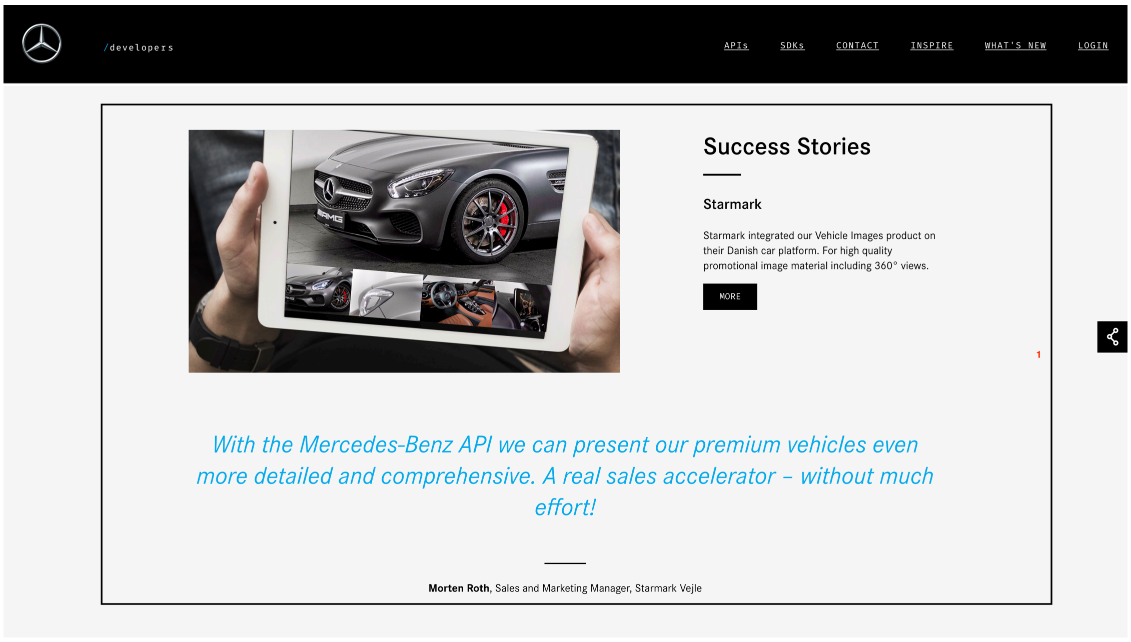
The following potential collaborators are derived from the study data:

- API consumer
- Sales and marketing

#### *Concerns*

- How to engage business roles of the API consumer?
- How to market API offerings to non-technical roles?
- How to communicate with API consumers?

#### *Example*



Success Stories

Starmark

Starmark integrated our Vehicle Images product on their Danish car platform. For high quality promotional image material including 360° views.

*With the Mercedes-Benz API we can present our premium vehicles even more detailed and comprehensive. A real sales accelerator – without much effort!*

**Morten Roth**, Sales and Marketing Manager, Starmark Vejle

Figure 5.12: Daimler Developer Portal - Success Story

As described in the example of *Pattern 17: Role-based marketing*, Mercedes-Benz utilizes role-based marketing on its developer portal. Figure 5.12 illustrates a part of the business and enterprise marketing material. The user is presented a success story[35]. The success story describes one successful API integration. The name of the API consumer is presented with a short description about the integrated API product and the achievements of the integration. Furthermore, the success story quotes a sales and marketing manager from the API consumer in focus. The quotes underlines the ease, success, and outcome of the integration. Interested readers can click on 'more' to get redirected to the full case of the successful integration[36].

### *Context*

The API consumer organization consists of different teams, roles, and stakeholders. The API documentation targets primary the application developer but other stakeholders are also involved in the buy-decision that has to be made before integrating third-party APIs. Procurement, product owners, and other management and business roles should also be provided with a high-quality user experience.

### *Forces*

Forces that have to be resolved and balanced:

- Different roles require additional user stories for their user experience
- API marketing material for non-technical stakeholders

### *Influence Factors*

Table 5.22 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 19: Customer success stories*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | Partner | Public |
| Maturity | Development | Pilot | Production | |
| Number of API Consumers | <20 | >20 | >10,000 | na |
| Type of Platform | Marketplace | Developer Portal | Backend APIs | na |
| Partner Type | B2B | B2C | B2G | none |
| Monetization | Free | In Product | Contractual | Per API Call |

Table 5.22: Influence Factors for Pattern 19

As visible in table 5.22, this pattern is identified for APIs offered to public and partnering third-party consumers. The solution approach is documented in production only. In the studied cases, the number of API consumers is above 20. In

[35]https://developer.mercedes-benz.com/home/business
[36]https://developer.mercedes-benz.com/inspire/starmark

all cases, the platform is identified to be a developer portal. The APIs are found to be offered to businesses, consumers, and government organizations. All variants of monetization are identified in the context of this solution approach. In the studied cases, product APIs are not offered for free.

#### *Solution*

Success stories illustrate the purpose and return of investment of the API integration from an API consumer perspective. They are a commonly used tool to market products and services. Each success story documents a use case of offered API products and services based on a real-world example. The reference to real-world API consumers increases trust in the platform. Success stories can be used to promote API products to business and enterprise roles within the API consumer organizations. The creation of success stories requires the identification of potential success stories and collaboration with the API consumer in focus.

#### *Consequences*

The following benefits are derived from the study data:

- Enhance customer experience
- Engage costumers and strategic partners
- Focus on value creation and relevance for the customer
- Promote API platform and products

The following liabilities are derived from the study data:

- Additional effort of collaboration with API consumers

#### *Implementation Details*

The success story should promote one API product or user story. The story will illustrate the value behind the API platform and products. The portal provider should reach out to sales and marketing teams to collaborate on the creation of the success story. First, the API products and user stories that should be promoted have to be selected. Second, potential API consumers have to be identified. Well-known firms can be used to market the prestige of the platform. As an alternative, strategic partners or partnering API consumers can be chosen to take advantage of existing communication channels. Next, the portal provider should reach out to the potential API consumers. The success story idea is presented to the API consumer. Both parties can iterate and enhance the user story in collaboration. The API consumer can be asked to give some quotable recommendations about the integration. The success story can also be used to promote the business of the API consumer. This can be used as an incentive to convince the consumer to collaborate.

Each success story can be added to a catalog of success stories hosted on the developer portal. The best success stories can further be referenced on the landing page or within the marketing material for business and enterprise stakeholders. Here, the success story is briefly summarized to describe the value of the integration. The brief summary also links to the full success story within the success story catalog.

### *Related Standards*

- Business Model Canvas (Barquet et al., 2011)
- Design Thinking (Plattner et al., 2010)
- Focus on Value (Limited & Office, 2019, p. 39)
- Value Proposition Design (Osterwalder et al., 2014)

### *Related Patterns*

*Pattern 10: Tailoring APIs to products* draws a clear path for the creation of marketing material and can be used as a base line to identify success stories on a product and user story level.

Marketing material for business and enterprise roles can be integrated using a role-based marketing approach as described in *Pattern 17: Role-based marketing*. API consumers without technical capabilities can further be supported by *Pattern 16: Integration partner management*.

### *Known Uses*

- Mercedes-Benz
- C2, C3, C8

### 5.7.20 Pattern 20: First-level support

#### *Stakeholders*

The following applicants are derived from the study data:

- API management

The following potential collaborators are derived from the study data:

- Customer support

#### *Concerns*

- How to support a growing number of API consumers?
- How to communicate with API consumers?
- How to provide efficient support for API consumers?
- How to manage non-complex, routine API consumer requests?

#### *Example*

The API management of C13 collaborates with the in-house first-level customer support to manage API consumer inquiries. The first-level support is installed as the first point of contact for external API consumers. Each customer request arrives at the support team, which then provides assistant for the different kind of inquires, e.g. feature requests, bug reports, or business inquiries. Non-complex and routine requests are answered directly by the first-level support. If needed, the first-level support forwards and escalates issues to second-level support teams for more advanced technical support. The second-level support consists of a team of IT technicians. In case specialized support is required, the request is forwarded to the third-level support which is provided by the backend and portal provider teams.

#### *Context*

API providers have to effectively and efficiently support API consumers. One important aspect of customer support is the management of incoming requests (Limited & Office, 2019, p. 156). The first point of contact plays a key role in the management tasks and dictates the flow of information and support. The portal provider usually acts as a natural first point of contact for the API consumers. A second-level of support is provided by the backend providers and the gateway provider.

### *Forces*

Forces that have to be resolved and balanced:

- Efficient and effective management of customer requests
- Scalability of support activities
- Transparency
- User experience

### *Influence Factors*

Table 5.23 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 20: First-level support*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | **Partner** | **Public** |
| Maturity | Development | Pilot | **Production** | |
| Number of API Consumers | <20 | **>20** | **>10,000** | **na** |
| Type of Platform | **Marketplace** | **Developer Portal** | Backend APIs | na |
| Monetization | **Free** | **In Product** | **Contractual** | **Per API Call** |

Table 5.23: Influence Factors for Pattern 20

As visible in table 5.23, the pattern is applied for APIs offered to public and partnering third-party consumers and is documented in production only. *Pattern 20: First-level support* has not been identified in cases with less than 20 API consumers. Both marketplaces and developer portals are found to apply the solution approach. In the context of this pattern, the monetization strategy varies across the cases.

### *Solution*

The onboarding of a dedicated first-level customer support is a decision that has to be made. Outsourcing support activities to a support team shields the portal provider from non-complex and routine requests. Thus, it should be considered if the number of those requests occupies a large percentage of the overall incoming requests (Walker, 2001, p. 25). A second influence factor is the request volume (Walker, 2001, p. 27-28). If it exceeds a manageable amount for the portal provider, a first-level support can be utilized (Walker, 2001, p. 27-28).

If the API provider organization operates a first-level support for other products and services, the API provider should approach the responsible stakeholders and discuss the onboarding. The onboarding requires close collaboration and thought-out processes to provide an effective customer support. Especially, the exchange of information and requests between the first-level support and portal provider team have to be designed.

When the dedicated first-level support is installed as the new first point of contact, the portal provider becomes the second tier of support. Backend provider and gateway provider teams can further provide a third tier of support for their provided services. This three-tier chain of support follows state-of-the-art approaches (Walker, 2001, p. 28). The first-level support provides routine help desk assistant. The second-level support provides overall technical support. The third-level support is provided by experts of specific services.

### *Consequences*

The following benefits are derived from the study data:

- Reduction and management of complexity
- Reduction of overhead for the portal provider
- Support costumers and strategic partners efficiently

The following liabilities are derived from the study data:

- Close collaboration between first-level support, portal provider, and backend provider required
- Technical and complex requests will not be answered immediately

### *Implementation Details*

The goal should be to enable the customer support to solve as many requests independently as possible. This saves time and frustration for the customer and saves capabilities of the API provider teams.

First, the first-level support has to be onboarded. The onboarding includes knowledge sharing about the API offerings. The portal documentation can be a good starting point. Additional resources that can be provided to the customer support team are FAQs, internal wikis, and others. This ensures that the costumer support can handle routine and non-complex inquires directly. The customer support should maintain and expand given documents whenever a non-technical or non-complex inquiry could not been answered directly. Overtime, given the right feedback, the costumer support can expand its area of assistance.

In case of bug reports, feature requests, or specific technical support requests, the customer support has to forward the request to the API provider. Inter-team communication has to be managed in an efficient manner to avoid loss of contextual information and reduce the overhead of managing those requests.

Whenever a new API service, user story, or product is added to the API platform, the portal provider has to notify the customer support about the changes. The same goes for other life-cycle changes within the offerings.

In case the customer support is managed by a third-party organization, the API provider can aim for a collaboration based on SLAs (Walker, 2001, p. 28).

### *Related Standards*

- Focus on Value (Limited & Office, 2019, p. 39)

### *Related Patterns*

Documentation and related artifacts are used to guide the user through the API integration. *Pattern 13: API product documentation* and *Pattern 14: Cookbooks* can be utilized to improve the quality of the API documentation which can lead to fewer support requests.

*Pattern 10: Tailoring APIs to products* improves pricing transparency and service discoverability by bundling API offerings into API products based on API consumer needs.

To manage incoming feature requests efficiently and effectively, *Pattern 21: Service desk software* can be utilized. The first level support can be supported by maintaining a growing FAQ as described in *Pattern Candidate 44: Growing FAQ*.

*Pattern Candidate 24: Contact form automation* and *Pattern Candidate 25: Smart contact form* can be used to improve the capabilities of the contact form of the developer portal.

### *Known Uses*

- C3, C4, C8, C10, C13

### 5.7.21 Pattern 21: Service desk software

*Stakeholders*

The following applicants are derived from the study data:

- Backend provider
- Portal provider

The following potential collaborators are derived from the study data:

- CIO
- Customer support

*Concerns*

- How to support a growing number of API consumers?
- How to communicate with API consumers?
- How to provide efficient support for API consumers?
- How to manage non-complex, routine API consumer requests?
- How to resolve bug reports effectively and transparently?
- How to effectively and efficiently collaborate with first-level support?
- How to effectively and efficiently collaborate with other API provision teams?

*Example*

The portal provider of C2 utilizes an automated contact form on its developer portal. The contact form follows the best practices described in *Pattern Candidate 24: Contact form automation* and *Pattern Candidate 25: Smart contact form*. In a next step, the portal provider aims to integrate service desk software into the current workflow. This enables the automatic generation of support tickets based on inquiries formulated through the smart contact form. For each inquiry, the API consumer receives an email with a link to a created support ticket within the service desk software. The ticket is used internally to manage the request and to communicate progress and support to the API consumer. Both internal communication hidden from the consumer and direct responses are possible. The goal is to increase transparency and improve collaboration between the supporting parties.

*Context*

API providers have to effectively and efficiently support API consumers. This includes the management of incoming requests (Limited & Office, 2019, p. 156).

Two important aspects that have to be considered are transparency for the API consumer and efficient collaboration between the supporting parties.

### *Forces*

Forces that have to be resolved and balanced:

- Efficient and effective management of customer requests
- Scalability of support activities
- Transparency
- User experience

### *Influence Factors*

Table 5.24 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 21: Service desk software*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | **Partner** | **Public** |
| Maturity | Development | Pilot | **Production** | |
| Number of API Consumers | **<20** | **>20** | **>10,000** | **na** |
| Type of Platform | **Marketplace** | **Developer Portal** | Backend APIs | na |
| Partner Type | **B2B** | **B2C** | B2G | none |
| Monetization | **Free** | **In Product** | **Contractual** | **Per API Call** |

Table 5.24: Influence Factors for Pattern 21

Table 5.24 shows that the solution approach has been applied with architectural openness of types partner and public. The platform in focus is documented to be in production only. The number of API consumers varies across the studied cases. This is a notable difference to the influence factors detected for *Pattern 8: Data clearance*, where the number of API consumers is above 20. Both marketplaces and developer portals are used in the context of *Pattern 21: Service desk software*. The type of partner is found to be both B2B and B2C. The monetization strategy varies across the documented cases.

### *Solution*

Service desk software supports the service provider through automation (Limited & Office, 2019, p. 150). It follows the ITIL (2019) principle 'optimize and automate' which states: "Human intervention should only happen where it really contributes value" (Limited & Office, 2019, p. 39). A service desks is designed to be a single point of contact between the service provider and consumer (Limited & Office, 2019, p. 149-150). Each support ticket that is created through the service desk software captures all relevant information for the individual support case. The integration of service desk software should be considered a project that requires planning, integration, and management capabilities.

### *Consequences*

The following benefits are derived from the study data:

- Focus on value creation and relevance for the customer
- Increased transparency for the customer
- Reduction and management of complexity through automation
- Reduction of overhead for the portal provider
- Support costumers and strategic partners efficiently

The following liabilities are derived from the study data:

- Close collaboration between first-level support, portal provider, and back-end provider required
- Service desk software integration project required
- Training required to utilize the service desk software

### *Implementation Details*

First, the API provider should investigate if service desk software is utilized by other teams within the organization. If the organization includes customer support, IT support, or similar teams, they likely utilize a dedicated service desk software. If the API provider already collaborates with a first-level support, the integration of service desk software should come hand in hand with the onboarding of the first-level support. Otherwise, the API provider can request the onboarding onto the utilized service desk software from the support team. In case the organization does not utilize a service desk software yet, a collaboration with top management and procurement can be initialized to request the purchase of such software tools for the API service management.

Second, the API provider has to integrate the service desk software into current workflows. For instance, if a contact form is utilized, it can be used to automatically create support tickets for each submitted inquiry. Service desk software might also offer dedicated portals that can be integrated into the developer portal. If the developer portal is publicly accessible, the portal provider should collaborate with legal to investigate any legal requirements for support channels and contact information.

Third, the service desk software has to be adapted to the possible intents of the API consumers. This ensures that no contextual information is lost. Each contact inquiry contains information that has to be captured to effectively and efficiently manage and fulfill the request. This is especially important if the first point of contact cannot fulfill the inquiry directly. Furthermore, contact inquiries can have very different intents that require different data to be captured. To conclude, the support tickets should contain all required fields and be adapted to fit both technical and non-technical support use cases.

All supporting parties have to be convinced and onboarded onto the service desk software. This includes the backend providers. Backend providers have to support customers in case of dedicated requests regarding their offered services. The service desk software enables efficient forwarding of service requests between the different support tiers. Each supporting party has to be onboarded and taught how to use the software to support the API consumer. The individual responsibilities and utilization of the software has to be agreed on between all supporting parties.

#### *Related Standards*

- Collaborate and Promote Visibility (Limited & Office, 2019, p. 39)
- Focus on Value (Limited & Office, 2019, p. 39)
- Optimize and Automate (Limited & Office, 2019, p. 39)
- Service Request Management (Limited & Office, 2019, p. 156)

#### *Related Patterns*

A service desk software can also be managed by a dedicated first-level support. The onboarding of a first-level support team is described in *Pattern 20: First-level support*.

Additionally, *Pattern Candidate 24: Contact form automation* and *Pattern Candidate 25: Smart contact form* can be utilized to connect the developer portal's contact form to the service desk software.

*Pattern 2: Company-wide ticketing system* can be utilized to manage the collaboration between API provider teams.

#### *Known Uses*

- C3, C4, C8, C10, C12, C13

### 5.7.22 Pattern 22: Self-service

#### *Stakeholders*

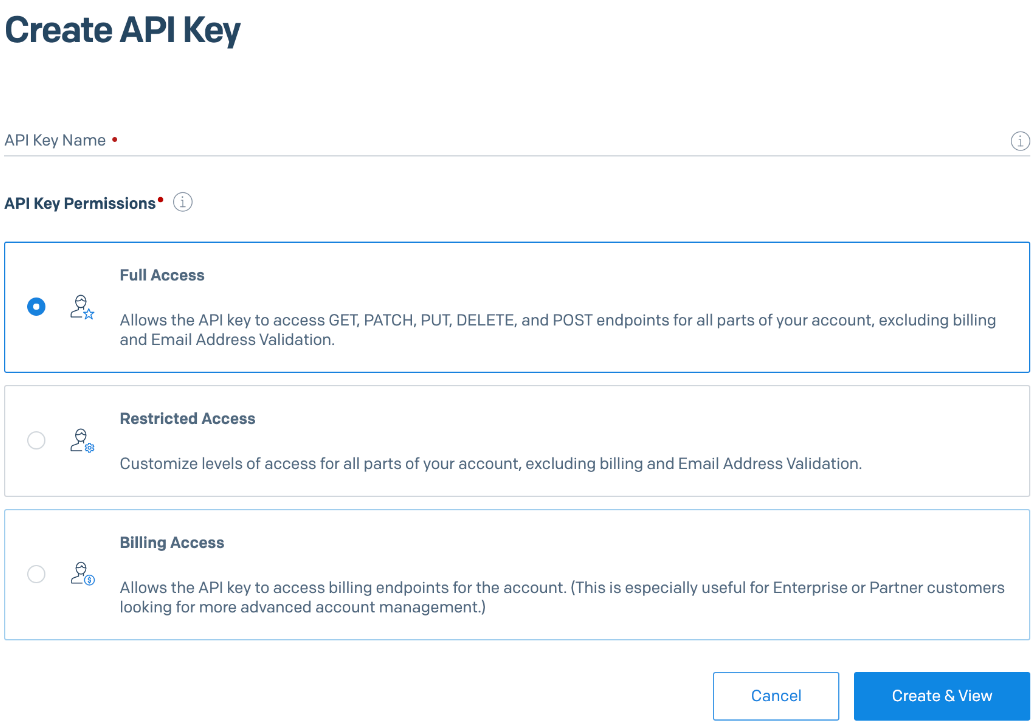
The following applicants are derived from the study data:

- API management

#### *Concerns*

- How to support a growing number of API consumers?
- How to manage non-complex, routine API consumer requests?
- How to onboard API consumers efficiently?

#### *Example*


Create API Key

API Key Name •

API Key Permissions•

(●) Full Access
Allows the API key to access GET, PATCH, PUT, DELETE, and POST endpoints for all parts of your account, excluding billing and Email Address Validation.

( ) Restricted Access
Customize levels of access for all parts of your account, excluding billing and Email Address Validation.

( ) Billing Access
Allows the API key to access billing endpoints for the account. (This is especially useful for Enterprise or Partner customers looking for more advanced account management.)

Figure 5.13: SendGrid Self-Service

SendGrid[37] offers web APIs for application developers to integrate e-mail and marketing solutions. Figure 5.13 shows the API key creation modal within the

[37]https://sendgrid.com/

SendGrid API consumer dashboard[38]. The user can give its new API key a name and select the permissions that should be granted. Giving the API key full access allows the associated application to utilize all API capabilities. Billing and e-mail address validation is excluded even with full access permissions. For this, the user can select billing access for advanced account management. This is a security mechanism that ensures that even with full access, regular API keys cannot access sensible account information. Additionally, users can click on 'restricted access' to specify fine-grained access levels for each API product. The 'create and view' button unveils the new API key. The API consumer has to copy the key into its application. After that, the key will never be shown again. This is another security mechanism. The API consumer can now proceed and integrate the API calls using the new key.

### *Context*

API providers have to effectively and efficiently support API consumers. This includes the onboarding of new application developers (De, 2017, p. 25-26). API consumers have to register each application that integrates with the API platform (De, 2017, p. 25-26).

### *Forces*

Forces that have to be resolved and balanced:

- Efficient and effective management of API consumer onboarding
- Scalability of support activities
- User experience

### *Influence Factors*

Table 5.25 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 22: Self-service*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | Group | **Partner** | **Public** |
| Maturity | Development | Pilot | **Production** | |
| Number of API Consumers | **<20** | **>20** | **>10,000** | **na** |
| Type of Platform | **Marketplace** | **Developer Portal** | Backend APIs | na |
| Monetization | Free | **In Product** | **Contractual** | **Per API Call** |

Table 5.25: Influence Factors for Pattern 22

Table 5.25 illustrates that the solution approach has been applied with architectural openness of types partner and public. In all cases, the platform has been in

[38]https://app.sendgrid.com/

production. The studied cases show a varying number of API consumers. Marketplaces and developer portals are used in the context of *Pattern 22: Self-service*. Product-based, contractual, and API call-based monetization strategies are in use.

### *Solution*

Self-service automates parts of the API integration process. Full self-service allows the API consumers to register themselves and their applications to the API platform without the need to interact with the API management altogether. The developer portal can provide all the required automation to make a full self-service work. This includes configuration control over billing and the developer onboarding process (De, 2017, p. 25-26).

To enable self-service, the developer portal needs to offer the API consumers capabilities to add billing information, to accept terms of use, and other contract related components. Furthermore, the API consumers require a mean to create API keys. The API gateway uses API keys to identify, authorize, and authenticate applications. Full self-service cannot be enabled if custom contracts or SLAs are necessary. Nevertheless, at least parts of the process can be automated even in case partnering API consumers require prerequisites before they can get started.

### *Consequences*

The following benefits are derived from the study data:

- Focus on value creation and relevance for the customer
- Increased transparency for the customer
- Reduction and management of complexity through automation
- Reduction of overhead for the portal provider
- Support costumers and strategic partners efficiently

The following liabilities are derived from the study data:

- Automation project required to implement the self-service

### *Implementation Details*

First, the monetization strategy has to be known. In case custom contracts and SLAs have to be negotiated with new API consumers, the self-service has to begin after the contractual part has been agreed upon by the API consumer and provider. In case, the monetization strategy does not require custom contracts, the terms of use and transparent pricing can be documented within the developer portal. This allows the API consumer to agree to named terms without the need to interact with the API provider. The API management can collaborate with legal to create legal material and design the required API consumer actions.

After the API consumers have agreed to the terms of use, they can register themselves on the developer portal. Two-factor authorization is commonly used by public developer portals to secure the API consumer information. For instance, both Twilio[39] and SendGrid[40] made two-factor authentication mandatory for all API consumers.

The registration on the developer portal provides the API consumers access to a dashboard. The consumer dashboard should provide further self-service functionality. Here, the API consumer can select or upgrade plans, insert billing information, and generate API keys for its applications. To improve security, generated API keys should be shown to the user only once. The API consumer has to ensure that the credentials are stored safe. The dashboard provides utilities to deactivate API keys and create new ones.

### *Related Standards*

- Focus on Value (Limited & Office, 2019, p. 39)
- Optimize and Automate (Limited & Office, 2019, p. 39)
- Service Request Management (Limited & Office, 2019, p. 156)

### *Related Patterns*

*Pattern Candidate 32: Role system in developer portal* can reduce the complexity within the developer portal. The role system within the API consumer dashboard can be used to break down self-service functionalities into a set of user stories for each role. This can improve the user experience of different stakeholders of the API consumer.

### *Known Uses*

- SendGrid, Stripe, Twilio
- C2, C3, C8, C10, C12, C13

[39]https://www.twilio.com/blog/mandatory-2fa-account-login
[40]https://sendgrid.com/docs/ui/account-and-settings/two-factor-authentication/

### 5.7.23 Pattern 23: Multi-tenant management

#### *Stakeholders*

The following applicants are derived from the study data:

- API management

The following potential collaborators are derived from the study data:

- API consumer
- Backend provider

#### *Concerns*

- How to manage APIs within a group of subsidiary or partnering firms?
- How to centralize but allow distributed control of API management?

#### *Example*

The API management documented in C5 provides a multi-tenant API management platform. The API management operates as an IT service supplier for a set of subsidiary companies. Each firm within the group acts both as an API consumer and an API provider. Thus, each subsidiary can provide API services that are published through a central portal and consume other services. Some subsidiary firms have special requirements and are treated as separate tenants. This is enabled through the multi-tenant management capabilities of the API gateway software. Each tenant is granted an own instance of the portal on which the tenant can publish private APIs and configure its portal API management in isolation. Since all portal instances are operated through a centralized API gateway, the interoperability between APIs is still given.

#### *Context*

Within the context of a corporation with several subsidiaries and partnering firms, a central API platform can be utilized to share API products and services across the ecosystem. Subsidiary firms or partners might want to publish their own private APIs and request more control over their platforms.

#### *Forces*

Forces that have to be resolved and balanced:

- Centralization efforts
- Distributed control requested

### *Influence Factors*

Table 5.26 marks selected influence factors from the context of the detected cases that utilize the solution approach described in *Pattern 23: Multi-tenant management*.

| Attribute | Attribute Values | | | |
|---|---|---|---|---|
| Architectural Openness | Private | **Group** | **Partner** | **Public** |
| Maturity | Development | Pilot | **Production** | |
| Number of API Consumers | **<20** | >20 | >10,000 | **na** |
| Type of Platform | **Marketplace** | **Developer Portal** | Backend APIs | na |
| Partner Type | **B2B** | B2C | B2G | none |
| Monetization | Free | In Product | **Contractual** | **Per API Call** |

Table 5.26: Influence Factors for Pattern 23

As visible in table 5.26, the solution approach has been applied with architectural openness of types: group, partner, and public. Thus, in the documented cases, it is found suitable to interact with external consumers. In all cases, the platform has been in production. The number of API consumers is identified not to exceed 20. Both marketplace and developer portal based API management are documented in the context of *Pattern 23: Multi-tenant management*. In all cases, the API offerings are directed towards other businesses and a monetization strategy based on volume and contracts is utilized.

### *Solution*

Multi-tenant management enables both centralized infrastructure and distributed control. The API gateway can be configured to treat each partnering or subsidiary company as a different tenant. The tenants are granted different views of the same API ecosystem. The infrastructure of an internal API platform can be maintained by one central gateway team and utilized within several subsidiary or partnering companies. Thus, the API management acts as an IT service and infrastructure provider for each tenant. A tenant utilizes the services of the central API management but can also utilize own configurations and hence, API management tasks on its own portal. This enables the centralization of capabilities and control for each tenant. Since all APIs share the same API gateway, API interoperability between tenants is enabled. Additionally, a central developer portal can be provided for subsidiaries that do not see the need to be treated as a separate tenant. Each tenant can select which offered APIs should be visible. Furthermore, tenants can publish their own private APIs that are only visible to them.

### *Consequences*

The following benefits are derived from the study data:

- Centralized API management for all tenants
- Custom configuration options for each tenant

The following liabilities are derived from the study data:

- Additional requirement of multi-tenant management for gateway software
- Increased complexity for API management

### *Implementation Details*

Multi-tenant management should be avoided to reduce complexity and overhead for the central API management [IV6]. In case the API management initiative aims to provide API management services for a set of subsidiary or partnering firms, tenant separation might be a requirement from the API consumers. The switch between different API gateway software systems is a complex process [IV3, IV4, IV6]. Since only a few API gateway software systems support multi-tenant management, the requirement for multi-tenant support should be analyzed as soon as possible to avoid the need to migrate API gateways later on. Multi-tenant provision gives each tenant control over their API provision management. Each tenant can decide how to manage its APIs for its API consumers separately. This includes monetization strategies. The goal should be to centralize API management as much as possible. If subsidiaries or partner firms demand more independence or ownership, a multi-tenant system can provide a good compromise.

### *Related Standards*

- Collaborate and Promote Visibility (Limited & Office, 2019, p. 39)
- Focus on Value (Limited & Office, 2019, p. 39)
- Optimize and Automate (Limited & Office, 2019, p. 39)
- Service Request Management (Limited & Office, 2019, p. 156)

### *Related Patterns*

Since each tenant is both API consumer and API provider, both *Pattern 6: SLAs with backend providers* and *Pattern 7: SLAs with API consumers* can be utilized.

### *Known Uses*

- C5, C8, C12

### 5.7.24 Pattern Candidate 24: Contact form automation

Customer inquires can have different intents (Limited & Office, 2019, p. 156). Contact inquiry forwarding should be automated. A custom contact form should allow the API consumer the selection of the inquiry type. Each inquiry should then be processed automatically based on its type and forwarded to either a first-level support, sales, or directly to the portal provider team. Contact form automation can be integrated with *Pattern 21: Service desk software* and *Pattern Candidate 25: Smart contact form*.

**Known Uses:** C2

### 5.7.25 Pattern Candidate 25: Smart contact form

A smart contact form can be utilized to gather specific information based on the type of inquiry. Based on the inquiry type, different form content should be offered to the user. For instance, an inquiry of type technical or bug report changes the contact form to include input fields for log files and a selection menu of the endpoint connected to the defect. Furthermore, it could require the user to log in to link the inquiry to the user's API keys and other related information.

**Known Uses:** C2

### 5.7.26 Pattern Candidate 26: Video series

The documentation can be enhanced with a video series of integration examples. Video series document the API implementation form an API consumer perspective and can be used to illustrate common use cases or explain domain knowledge.

**Known Uses:** C10

### 5.7.27 Pattern Candidate 27: Open-source SDK

Software libraries and the documentation material can be open sourced. This invites API consumers to collaborate on the material. Public repositories can be used to manage API consumer requests and collaboration.

**Known Uses:** C10

### 5.7.28 Pattern Candidate 28: Service validation workshops

Workshops can be used to validate service ideas across silo boundaries. Formats like event storming[41] are utilized to enable close collaboration between API man-

[41]http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html

agement and backend provider teams. In collaboration, the parties can evaluate the feasibility of new services. Workshops can increase trust and improve personal relationships between the participants.

**Known Uses:** C5

### 5.7.29 Pattern Candidate 29: Account management

API providers should offer dedicated support for strategic partners. This can include technical and non-technical support staff and integration developers that are assigned to different API consumer accounts. The account management can be part of the SLA between the API provider and API consumer.

**Known Uses:** C6, C9

### 5.7.30 Pattern Candidate 30: Plug-in development

The API provider can implement own API offerings into consumer-side plug-ins for popular software ecosystems. API consumers of the ecosystem can install the plug-in that will manage the communication with the APIs. Open-source integrations promote own products within the target software ecosystem and ease the onboarding of new API consumers. For instance, WordPress plug-ins[42] enable quick integrations of third-party services into WordPress sites. API providers can provide custom plug-ins that ease the integration of their services.

**Known Uses:** C3

### 5.7.31 Pattern Candidate 31: Data clearing office

The exposure of new API endpoints to partner and public API consumers requires collaboration with legal and other functional teams. Data can be confidential, sensible, strategic, or have other properties that require clearance. Each data point exposed to external API consumers has to be approved by all stakeholders. This effort can be centralized by introducing a data clearing office. A data clearing office is an interdisciplinary committee which has to be contacted whenever data is offered to external API consumers.

**Known Uses:** C2

### 5.7.32 Pattern Candidate 32: Role system in developer portal

The developer portal is a single point of information for the API consumer (De, 2017, p. 172). The API consumer utilizes the developer portal to manage its integrations. Since the API consumer itself consists of different roles, teams,

[42]https://wordpress.org/plugins/

and stakeholders that collaborate to integrate APIs, the developer portal should provide functionality tailored for each identified persona. A role-based system enables role-based authorization strategies, and offers a better user experience through reduced complexity.

**Known Uses:** C12

### 5.7.33 Pattern Candidate 33: Procurement integration

The API consumer organization includes different roles, teams, and stakeholders that collaborate to integrate APIs. The API provider might need to convince non-technical stakeholders of the API consumer about the value of their offerings. The developer portal should provide functionality tailored for each identified persona. The API consumer's procurement could potentially influence the buy-decision of the API [IV12]. Traditionally, procurement teams handle the purchases and require strict processes to be able to fulfill a purchase. Procurement in traditional companies and especially in big enterprises follow strict compliance guidelines. The API provider has to enable transparent pricing models and contracts to ensure that the procurement is able to process the purchase. Furthermore, billing per API call might be difficult to integrate into traditional procurement tools. As a solution, the API provider can proactively integrate procurement functionality, e.g. SAP tooling, within the developer portal to allow automated billing and other advantages.

**Known Uses:** C12

### 5.7.34 Pattern Candidate 34: Keyword marketing

Google Ads[43] and other keyword-based marketing approaches can be utilized to promote and sell API products. Advertisement improves visibility and discoverability of the offered API products and services.

**Known Uses:** C3

### 5.7.35 Pattern Candidate 35: Hackathons

Hackathons can be used to explore different solution approaches or try out innovative ideas. In the context of API management, they can be used to trigger new initiatives.

**Known Uses:** C7

[43]https://ads.google.com/intl/en_en/getstarted/

### 5.7.36 Pattern Candidate 36: Pilot workshops

Workshops can be utilized to enable close collaboration between API consumer and API provider teams. For instance, workshops can be used to kick-off pilot projects or to collect feedback from stakeholders. They increase trust and improve personal relationships between the participants.

**Known Uses:** C5, C11

### 5.7.37 Pattern Candidate 37: Conferences

Conferences can be utilized to promote public or partner API products and services. Advertisement improves visibility and discoverability of the offered API products and services.

**Known Uses:** C3

### 5.7.38 Pattern Candidate 38: Bar camps

A bar camp creates a similar atmosphere as a conferences but does not organize talks. Instead it is used to bring the API provider and consumer together in one room. An organization can organize a bar camp and invite all interested parties. It enables direct feedback and networking.

**Known Uses:** C11

### 5.7.39 Pattern Candidate 39: Tech talks

Self-marketing can be utilized to promote the API platform internally and convince other teams and top management of the importance and validity of the business case. Tech talks, road-map presentations, or similar internal conferences can provide a stage to promote the API platforms and products.

**Known Uses:** C3, C4

### 5.7.40 Pattern Candidate 40: Intranet and social media

Similar to the solution approach described in *Pattern Candidate 39: Tech talks*, the intranet and additional internal social media platforms can be used to promote the API platform and its products. For private and group-based API platforms, this can increase discoverability for potential API consumers.

**Known Uses:** C5, C7

### 5.7.41 Pattern Candidate 41: Inner source-based platforms

An API initiative can decide to inner source API platform projects. This increases visibility, transparency, and discoverability. The internal version control code hosting platform can be utilized to manage issues and requests of collaborators and API consumers.

**Known Uses:** C7

### 5.7.42 Pattern Candidate 42: Declarative API platform

If API platform projects are inner-sourced, a declarative approach can be used to add new APIs to the offering. In a declarative contribution approach, backend providers have to commit potential configuration and documentation changes for their API offerings directly within the code repositories of the API platform. This code-first approach can increase the transparency of changes within the API platform. Inner-sourcing is documented in *Pattern Candidate 41: Inner source-based platforms*.

**Known Uses:** C7

### 5.7.43 Pattern Candidate 43: Support community

A forum-based support community can be integrated into the developer portal to allow exchange between API consumers (De, 2017, p. 26). This creates a new support channel where API consumers can collaborate and support each other (De, 2017, p. 26).

**Known Uses:** C2

### 5.7.44 Pattern Candidate 44: Growing FAQ

An FAQ page can help to answer common questions of API consumers. It can further be used to onboard a first level support as described in *Pattern 20: First-level support*. A growing FAQ is maintained over time and updated whenever a new common question is identified. It can consist of a public part and a private part. The private part can be used to quickly reuse support responses while the public part can be integrated into the developer portal directly.

**Known Uses:** C3

### 5.7.45 Pattern Candidate 45: API status

An API status page can be used to automatically report issues and defects of back-end services and API platforms. It can be integrated into the developer portal to automatically inform API consumers of current downtimes and issues. This can reduce the volume of incoming redundant bug reports.

**Known Uses:** C2, C12

### 5.7.46 Pattern Candidate 46: Support hero

Incoming customer support requests can be disruptive to the current work. One way to handle support requests in an agile way is to create a support hero role. The role assignment rotates every sprint, every week, or bi-weekly between the team members. The support hero has the responsibility to work on all incoming requests. In a Scrum-based environment, the estimated support effort should be considered during sprint planning meetings. Each team of the API provision management should have its own support hero, e.g. each backend provider, portal provider, and gateway provider team. This ensures that every team within the support chain stays responsive and works on forwarded tickets. Service desk software can ease the communication between the support heroes. *Pattern 21: Service desk software* documents the implementation of such software.

**Known Uses:** C3, C4

### 5.7.47 Pattern Candidate 47: Sample projects

Sample projects are open source integration examples that utilize API products and illustrate specific user stories. They should be published on commonly used repository management sites such as GitHub and can be referenced by the documentation. Sample projects should be simple but working API consumer applications. They can be used as starters or references to ease the API integration. A short documentation should be provided for how to setup the project and get it running locally on the application developer's machine. They can also be used to illustrate library-based integrations as described in *Pattern 15: Software libraries*.

**Known Uses:** C10

### 5.7.48 Pattern Candidate 48: Internet and social media

Social media platforms allow the promotion of a public and partner-based API platform and its products. Additionally, social media platforms can act as a place for the community to exchange knowledge. For private and group-based API platforms, intranet-based social media platforms can be used as described in *Pattern Candidate 40: Intranet and social media*. To offer a dedicated forum for community support and collaboration, *Pattern Candidate 43: Support community* can be utilized.

**Known Uses:** C2

### 5.7.49 Pattern Candidate 49: Quarterly alignment meetings

Quarterly alignment meetings between all backend providers, the gateway provider, and the portal provider teams can strengthen the commitment and align goals. The portal provider team can use those alignment meetings to report new API consumer requests. Thus, alignment meetings can be used as a platform to convince backend providers to implement required services for new API products.

**Known Uses:** C3, C4

### 5.7.50 Pattern Candidate 50: Scrum master resolution

In case of outage or defects of backend services, an efficient and effective collaboration between all provider teams is required. In most scenarios, the defect is detected by an API consumer or through monitoring on the API gateway. The defect has to be reported to the backend provider. In case, agile methods such as Scrum are utilized, the prioritization and resolution of the issue can be managed by the Scrum masters of each team. This improves personal relationships between the teams and has a positive impact on the defect resolution time.

**Known Uses:** C11

### 5.7.51 Pattern Candidate 51: Supplier onboarding

Marketplaces are characterized by their architectural openness on both the demand and supply side of the platform. Thus, the supply-side of the marketplace is opened for third-party API providers. A marketplace provider has to ensure that the API offerings are aligned and adhere to the quality standards of the platform. For this, the marketplace provider should curate possible API providers based on a defined application and onboarding process. The quality standards should be integrated into SLAs. The utilization of SLAs is further documented in *Pattern 6: SLAs with backend providers*.

**Known Uses:** C12

### 5.7.52 Pattern Candidate 52: Supplier monitoring

A marketplace provider has to ensure that the API offerings follow the quality standards of the marketplace. For this, continuous analytics and monitoring should be integrated into the API gateway. If an API provider does not meet certain KPIs, automated notifications can be sent by the API gateway. This is further documented in *Pattern Candidate 58: Notification system*.

**Known Uses:** C12

### 5.7.53 Pattern Candidate 53: API test values

API providers can offer a set of defined test values and corresponding test responses. They support the API consumer to integrate error cases and test the API implementation in a sandbox environment. The test responses can be triggered by defined test values within the API request. For instance, a payment provider can provide a set of fake credit cards that each will lead to a defined error or success API response. API consumers can utilize the fake credit cards to test different scenarios within their applications. The API test values have to be documented thoroughly. *Pattern 14: Cookbooks* provides a good starting point to abstract use stories into documentation. In this context, the API test values can be added to the cookbooks.

**Known Uses:** C9, C10

### 5.7.54 Pattern Candidate 54: Penetration tests

API providers have to ensure the security of their API platforms. The gateway provider can utilize penetration tests to proof the ability of the gateway to endure potential attack vectors. Penetration tests are usually provided by third-party security firms that offer audits of the security of IT systems.

**Known Uses:** C5

### 5.7.55 Pattern Candidate 55: Integration levels

Internal API platforms are used to improve API discoverability and concentrate API management tasks. An API platform initiative has to onboard API providers and backend providers onto its API platform. A three-level scale can be utilized to manage the integration of APIs. First, other API platforms and offerings can be linked within an API registry. This is documented in *Pattern 1: Internal API registry*. Second, the API documentation can be moved to the developer portal. Third, the API is integrated with the API gateway of the API platform. The three-level integration approach provides a clear integration path for internal APIs onto an API platform and enables value-creation for the API consumer from the beginning.

**Known Uses:** C7, C14

### 5.7.56 Pattern Candidate 56: Blogs

API providers can offer blogs that promote, illustrate, and document different aspects of the integration experience, technical deep-dives, customer success stories, and others. Each blog post focuses on one topic. Thereby, blog posts do not need to have a strong coupling with each other or to other materials. They can be utilized to offer additional information that would not fit into other categories.

Newest blog posts can be linked in a weekly, monthly, or quarterly newsletter. The utilization of newsletter is further documented in *Pattern 18: Newsletter*. The creation of customer success stories is explained in *Pattern 19: Customer success stories*.

**Known Uses:** C2, C10

### 5.7.57 Pattern Candidate 57: Changelogs

Changelogs can be utilized to document continuous improvements and additions to the API offerings. Each change to an API or its endpoints is summarized within the changelog. It provides a temporal sorted overview of the progress of the API platform. It is generally tailored towards application developers and other technical stakeholders.

**Known Uses:** C3

### 5.7.58 Pattern Candidate 58: Notification system

The API gateway should implement a notification system that alerts stakeholders when backend services time out or do not meet quality parameters. Automated e-mails and other types of notifications can be sent directly to the contact persons of each backend service. The notification can include contextual information such as logs, missed KPIs, and identifiers of affected clients and endpoints. Since the API gateway is the single source of truth within the API management, it can be utilized to automate those status messages. A notification system can increase the reaction time. Further, the automation of those messages can ease the personal relationship between API management and backend providers since the call to work comes from the API gateway directly. Notification systems can support the quality management defined in *Pattern 6: SLAs with backend providers* and *Pattern 7: SLAs with API consumers*.

**Known Uses:** C2, C12

# 6 Discussion

In this chapter, the three RQs are answered. The results of the study are explained and interpreted. Further, the results are evaluated and the research approach justified.

As specified in section Objectives, this thesis aims to identify API management concerns and document solution patterns from an API provider perspective. To accomplish this, we conducted 16 semi-structured interviews with API provider stakeholders. The encoding and evaluation of the interviews lead to the creation of 14 cases of different API platforms. The case data was used to create three research artifacts.

- **A stakeholder-relationship map** details relationships between different roles, teams, stakeholders, and software artifacts of API management. The outcome is further compared to the literature and lays out naming conventions for the pattern language.
- **A context distribution matrix** illustrates the distribution of the studied cases across derived context attributes and values. The matrix is used to put the studied cases into perspective.
- **A pattern catalog** links stakeholders and identified concerns to documented patterns. It provides insights about API management challenges and solution approaches.

In the following, the results of this study will be discussed in more detail.

The stakeholder-relationship map illustrates that most communication between the API provider and API consumer happens through software artifacts such as the developer portal. The communication between the two parties is thereby dependent on social boundary resources managed by the portal provider. This emphasizes the importance of effective and efficient portal provision management but it also underlines the provider-consumer relationship where the API consumer accesses resources supplied by the API provider. Thus, the API management has to lead the initiative. The relationships between the different API provider roles, teams, and stakeholders draw a different picture. It can be noted that the collaboration within the API provision management lacks standard software artifacts that support the communication. In the studied cases, most communication between API provider roles, teams, and stakeholders is based on ad hoc channels such as emails. *Pattern 2: Company-wide ticketing system* , *Pattern 6: SLAs with backend providers* , and *Pattern 21: Service desk software* illustrate that solution approaches exist to standardize the collaboration but feedback received during the interviews stresses challenges of collaboration between API provider entities. Examples include collaboration for quality, defect, and incident management across team, business unit, or company boundaries. Further, the stakeholder-relationship map emphasizes the special role of the portal provider and API governance authority. Both have to collaborate with almost all other API provider entities.

Next, the cases derived from the interviews where analyzed and compared based on 20 context attributes. The attributes are derived from the literature or identified through the interview encoding. Section Influence Factors describes the origin of every context attribute and documents occurrences and percentages of cases for each attribute value. The attributes provide context and show the distribution of the cases. 64% of the studied cases offer APIs to partnering organizations while only 43% offer API products to the public. 71% of all cases maintain API platforms in production. 86% of platforms offer API products among others to businesses. 64% of the identified platforms are categorized as developer portals while 14% are either characterized as marketplaces or pure backend APIs. The monetization strategy varied between the studied cases. Most (57%) API provider-consumer relationships are based on a contract while 21% of the platforms offer free API access. 43% of studied cases use a volume-based monetization strategy. Described attributes directly influence the decision making of the API management in focus and thus, lead to the identified solution patterns. The context distribution matrix puts the results of this work in perspective and allows a critical evaluation of the results.

To support the evaluation of the context in which each pattern has been derived, the most important context variables of the known uses are denoted in each pattern individually. We call those context attributes the influence factors that lead to the development of the solution approach. The influence factors aim to put the pattern into perspective and draw potential limitations to the generality of the solution approach. For instance *Pattern 15: Software libraries* potentially requires the most expenditure to implement. The solution approach is put into perspective since its influence factors illustrate that it is only applied when the number of API consumers is above 10,000. *Pattern 13: API product documentation* and *Pattern 14: Cookbooks* detail solution approaches to improve the documentation on the developer portal. Both solution approaches are linked to cases with API platforms in production only. The implementation of those solution approaches could still make sense in development or pilot phases of the platform. For instance, the portal provider could decide to create the first iteration of the API documentation following the patterns. Still, the patterns are put into perspective as this has not been the case in the studied cases.

The main outcome of this study is the pattern catalog. It follows the pattern language developed in section Pattern Language and includes 35 pattern candidates and 23 patterns. As described in section Pattern Language, the rule of three known uses is utilized to validate pattern candidates. Each validated pattern appeared to have more than three known uses within the studied cases. The documentation of the patterns follows best practices identified in the pattern literature. Each pattern includes linked stakeholders that act as applicants, linked concerns, an example, context, identified influence factors, the solution description, consequences, implementation details, related standards, related patterns, and a list of known uses. The interviews are predominately based on an API management and portal provider perspective. The provision of the API platform has been the focus of most interviews. Thereafter, the pattern catalog focuses on the effective and efficient provision of social and technical boundary resources to the API consumer. In this context, the communication between API provider entities and with the API consumer is identified as the main effort of the API management.

The three research questions of this study can be answered based on the three developed research artifacts.

**RQ1: What concerns do API providers face in their daily work?**

The pattern catalog taxonomy, which can be found in figure 1 of the appendix, links identified stakeholders to raised concerns and detected concerns to documented solution patterns. In total, 32 concerns have been derived from the interview data. The concerns are categorized using seven common concepts. 15 concerns are associated with the creation of API offerings. This emphasizes the focus on the requirements and needs of the API consumer. Four concerns target the collaboration within the API provision management. Seven concerns are linked to support management. One concern is associated to incident management and quality management respectively. Two concerns are raised in the context of internal API platform initiatives and one concern is raised in the context of a venture opportunity. The categories are derived based on common concepts of the literature such as API management concepts from De (2017) and ITIL (2019) (De, 2017; Limited & Office, 2019). They illustrate general directions of the raised concerns.

The stakeholder-relationship map, visualized in figure 5.2, provides the naming conventions for the API provider roles, teams, and stakeholders that are linked to the concerns. Four API provider entities have been linked to concerns: API management, portal provider, gateway provider, and backend provider. The API management is used to capture all roles and teams that build the core of the API provision management. The API management includes the portal provider and gateway provider. One concern can be raised by several stakeholders. Concerns raised by the API management are not linked to the portal provider and gateway provider again. Since interviews are conducted from a portal provider or API management perspective, no concerns are identified that link solely to the gateway provider. Overall, API management is associated to 16 concerns. The portal provider is linked to 20 concerns independently. 17 concerns are raised by the backend provider and three concerns by the API governance authority.

It can be noted that the emphasis on the daily work sets the focus on the API provision management and less on the overall API strategy. Thus, key strategy decisions are defined as the context for the daily work and mostly unquestioned within the pattern catalog. For instance, the monetization strategy of the platform is used as an influence factor and not as a solution approach to strategy concerns. On the other hand, *Pattern 3: API testing strategy* documents the creation of a testing strategy. In this case, the overall strategy and the daily work are deeply interconnected and the strategy is a necessary solution approach. The distinction between management and strategy and context and solution approach is made for every context attribute and pattern individually.

**RQ2: What influence factors impact the API management?**

The context distribution within the studied cases is laid out in section Influence Factors. Each pattern is linked to a set of context attributes that are identified as key influence factors. The most important influence factors are strategic decisions such as monetization strategies and the architectural openness of the platform. Additionally, the maturity of the platform and the number of current API consumers play an important role in the daily provision management. Other context attributes that provide interesting insights are the initial trigger or driver and the type of the utilized gateway. In the studied cases, 50% of API initiatives follow a top down driver while 50% are initiated by bottom up initiatives. Multiple cases follow both a bottom up and top down trigger, thus, the multiple counting of cases. The gateway is found to be a commercial product in 57% of the cases while 14% of the cases utilized an open source or no gateway.

The context attributes utilized in this thesis are drawn from the SOA literature. This emphasizes the similarity between SOAs and API platforms. Both describe software ecosystems that may include several subsidiaries, partnering, or third-party organizations (De, 2017, p. 12). Both are influenced by the partner type, network topology, service granularity, the heterogeneity of the partners, value chain integration, network governance, networking target, process output, the trigger motivation, and more. One identified difference is the governance type. In the studied cases, the API platform is always governance focally (100%) while Löhe and Legner (2010) also discover polycentric approaches (21%) in studied SOA cases (Löhe & Legner, 2010a, 2010b).

**RQ3: How do API providers manage concerns and what is the rationale behind the solutions?**

The pattern catalog documents the identified solution approaches. Each pattern is linked to targeted concerns and explains the rationale behind the solution. The fields Forces, Context, Consequences, Solution, and Implementation Details explain different aspects of the rationale and offer a blueprint for the decision making of the API management stakeholders.

Overall, 23 patterns are documented. The API management is linked as an applicant to 10 patterns. The portal provider is identified as the applicant in 13 additional patterns. This underlines the central role of the portal management within the provision management but can also be explained by the portal provider perspective of the interview data. Five patterns link the backend provider as an applicant. Two patterns are linked to the API governance. Additionally, the backend provider is mentioned six times as a potential collaboration partner, more than any other potential collaborator. This emphasizes the importance of the collaboration between the individual backend providers and the portal provider and overall API management. Quality management, incident management, and support management activities are shared across the provider entities and require effective and efficient collaboration and communication. Sales and marketing are linked as potential collaborators in four patterns while legal stakeholders are associated to three patterns. Customer support is mentioned in two patterns. The three patterns *Pattern 4: Pilot project*, *Pattern 7: SLAs with API consumers*, and *Pattern 19: Customer success stories* require active collaboration with API consumers. Overall, eight different potential collaborators are listed in different solution approaches. Hence, API management has to collaborate and communicate with several different roles, teams, and stakeholder across team, business unit, and organizational boundaries.

Identified patterns such as *Pattern 20: First-level support* and *Pattern 21: Service desk software*, and *Pattern 22: Self-service* stress the importance of scale within the API management. The standardization of communication channels drives solution approaches such as *Pattern 2: Company-wide ticketing system* and *Pattern 21: Service desk software*. The commodification of knowledge transfer can be interpreted as the underlying goal of *Pattern 13: API product documentation*, *Pattern 14: Cookbooks*, and *Pattern 15: Software libraries*. It can be noted that the role of commoditized knowledge embedded in software artifacts increases with the size of the platform. This matches the conclusion drawn by Islind et al. (2016) (Islind et al., 2016).

The pattern catalog is balancing rigor and relevance by embedding sound research methodologies, addressing research gaps, and following challenges from the industry. It is meant to offer patterns and supporting information for the decision making of API management. Related concerns, forces, consequences, and influence factors are meant to support API management stakeholders in the evaluation of documented solution approaches. However, patterns are only meant to sketch the solution approaches (Zimmermann et al., 2020). They provide the overall blueprint based on the findings of expert interviews and are iteratively improved based on peer feedback. They should not be followed blindly but utilized as starting points to solve common impediments (Zimmermann et al., 2020).

The catalog offers insights and value for both research and industry. For future research, the pattern catalog forms a domain vocabulary (Evans, 2003; Zimmermann et al., 2020). Zimmerman et al. (2020) argue that a 'lingua franca' of API design is missing to date (Zimmermann et al., 2020). Similarly, to the best knowledge, the context, concerns, and solution approaches of API management from an API provider perspective have not been documented within the scientific literature. However, it should be noted that a wide set of handbooks and guidelines for API management from an API provider perspective exists. Most notable for this study, De (2017), which covers a wide set of best practices and documents state-of-the-art solution approaches for API management and governance (De, 2017). This thesis draws best practices from different sources also outside the scientific literature such as standards, management books, websites, and documentations but aims to embed the findings in a rigor research approach.

The results will be concluded in the next chapter. Additionally, an outlook for future research is given.

# 7 Summary

In the following chapter, the status of this thesis is summarized. First, this thesis is concluded. Both achieved and open goals will be presented. Next, identified limitations of this thesis are discussed. Finally, an outlook for future work is given.

## 7.1 Conclusion

This study offers applications for future research and API management stakeholders from the industry. 32 API management concerns are identified and linked to four API provider entities. 23 patterns and 35 pattern candidates are formulated to document detected solution approaches. A stakeholder-relationship map, a context matrix, and a pattern catalog answer the three RQs and provide insights about concerns, influence factors, solution approaches, and their reasoning. The three research artifacts are based on 14 studied cases derived from 16 semi-structured interviews. In the following, the realized and open goals of this thesis are presented. Next, identified limitations are discussed.

### Realized Goals

The main objective of this thesis is to identify API management concerns and document solution patterns from an API provider perspective. For this, three RQs have been developed. The RQs are answered through the creation of three research artifacts. First, real-world concerns of API providers are identified. Second, influence factors for the API management are derived. Third, recurring solution approaches and their reasoning are documented.

The knowledge base of this study is based on extensive literature reviews. It builds upon research agendas from Yoo et al. (2010) and de Reuver et al. (2017) (de Reuver et al., 2018; Yoo et al., 2010). Related areas of research have been identified iteratively by going forward and backward through citations. The literature reviews enable the identification of further research gaps and challenges, applicable research methodologies and frameworks, scientific foundations, and related studies. Several research gaps and challenges have been found that motivate this study (Eaton et al., 2015; Henfridsson & Bygstad, 2013; Jansen et al., 2009; Koci et al., 2019; Mathijssen et al., 2020; Sohan et al., 2015). The development of the research approach and sound methodologies is based on literature reviews of related work and the scientific literature of IS research. The scientific foundations are used to build a common vocabulary and understanding of the socio-technical environment in which this study is embedded. The pattern literature has been reviewed to collect best practices for the development of a pattern language.

The research is following an iterative design science framework. Iteratively and through continuous improvement, we developed three research artifacts. To collect data, we conducted semi-structured interviews with API management stakeholders. Each interview has been transcribed, encoded, and analyzed to evaluate the research artifacts. New insights from the interview data triggered changes to the research artifacts and led to new directions for additional and follow-up interviews. Overall, 16 interviews with API provider stakeholders from different backgrounds and industries have been conducted. More than 12 hours of interview material have been transcribed and encoded to collect data for this study. The encodings are validated through intercoder reliability. All derived pattern candidates utilize the rule of three known uses as established by Coplien (1994) (Buckl et al., 2008; Coplien, 1994).

To create an understanding of the relationships between different roles, teams, and stakeholders of API management, a stakeholder-relationship map is created. For this, the API provider entity is split into manageable entities and compared with one another and with API consumer entities. The developed stakeholder-relationship map provides naming conventions and insights about commonly used collaboration that was observed within the studied cases. A context-matrix is built to gather context attributes and values from the literature and studied cases to identify potential influence factors and put the solution approaches into perspective. In order to document stakeholders, concerns, and solutions in a standardized manner, a pattern catalog is utilized (Buckl et al., 2013).

## Open Goals

The following open goals have been identified. First, more follow-up interviews would allow for more status updates about the evolution and outcome of solution approaches over time (Buckl et al., 2013). Multiple interviews with the same interview partners enable the collection of longitudinal data. De Reuver et al. (2017) argue that longitudinal data about API management evolution is lacking (de Reuver et al., 2018). Furthermore, follow-up questions enable iterative assessment and refinement. Additional follow-up interviews with the interview partners were not conducted due to time constraints of this thesis.

Second, documented patterns are validated using the rule of three known uses. There are additional opportunities for further evaluation of the final pattern catalog. For instance, API providers could be guided through the application of the pattern catalog based on pattern workshops (Buckl et al., 2013). This could trigger further iterations of improvement. Successful implementation of the solution approaches would further justify the research artifacts.

Third, the identified concerns have not been reviewed in the context of the related literature. Similar to the derivation of influence factors from the literature and the comparison of identified stakeholders with the literature, detected concerns could be investigated based on the literature. This study lacks insights about the novelty of the raised concerns and the similarity between concerns within similar fields of research such as service-orientation.

### Limitations

The open goals lead to limitations of this study. Identified limitations include the lack of evaluation steps of the pattern catalog. The pattern-based design research methodology developed by Buckl et al. (20013) describes further evaluation and learning steps that are not applied in this thesis (Buckl et al., 2013). Instead, this study uses the pattern-based methodology as a justification to combine behavioral science and design science methodologies together but does not implement all proposed steps of the research approach (Buckl et al., 2013).

Further limitations are in regard to the potentially lacking generality of the identified pattern catalog. All but two studied cases are based on European organizations. The offered pattern catalog is based on API management platforms from several industries and different context and influence factors. The generality is limited however by the lack of interviews with international API provider stakeholders.

Additionally, only 43% of the studied platforms offered API platforms to public third-party developers and only 21% of the studied cases interacted with more than 10,000 API consumers. Islind et al. (2016) stresses the importance of research about smaller API platforms but the lack of scale does affect the generality of the pattern catalog nonetheless (Islind et al., 2016).

## 7.2 Future Work

In the following, the open goals and limitations are connected to an outlook for future research. In the previous section, the lack of generality of the pattern catalog is addressed. However, the focus on mostly established firms from Europe can be used in the future as a foundation to identify differences to solution approaches utilized by incumbent technology firms. The detection of differences could lead to identification of further legal, economic, social, technological, and organizational barriers as described by Bondel et al. (2020) (Bondel et al., 2020). Islind et al. (2016) calls for further research about the fine-tuning of platform boundary resources in small-scale contexts. The lack of scale within the studied API platforms can be utilized to better understand the first steps within the creation of API platforms. For this, further iterations of the pattern catalog could focus on API platforms in early stages of development and pilot phases.

Future work should further emphasize longitudinal data (de Reuver et al., 2018; Eaton et al., 2015). The collection of longitudinal data requires long-term research but is identified as a critical research gap within the platform and boundary resources literature (de Reuver et al., 2018; Eaton et al., 2015).

Finally, this thesis identified the similarities between API management and service-orientation. Similar to Zimmermann's (2017) discussion of the similarities between microservices and service-orientation, the interconnection of API management and service-orientation should be investigated (Zimmermann, 2017). Current literature understands API management and service-orientation as related but distinct fields of research (De, 2017, p. 12). Future research should investigate if the instigation of the API Economy and advancements of cloud computing and remote service communication lead to an advancement of the understanding of SOA. With the focus on service-orientation within API management and the utilization of new standards and technologies within SOA, the merging of the two fields of study should be researched.

# List of Figures

# List of Tables

# Bibliography

Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). *Web Services: Concepts, Architectures and Applications* (2004. Edition). Berlin ; New York, Springer.

Amaravadi, C. S. (2014). Office Information Systems: A Retrospective and a Call to Arms. *Journal of Software Engineering and Applications*, *07*(08), 700–712. https://doi.org/10.4236/jsea.2014.78065

Armstrong, M. (2006). Competition in Two-Sided Markets. *The RAND Journal of Economics*, *37*(3), 668–691.

Barquet, A., Cunha, V., Oliveira, M., & Rozenfeld, H. (2011). Business Model Elements for Product-Service System. In *Functional Thinking for Value Creation* (pp. 332–337). https://doi.org/10.1007/978-3-642-19689-8_58

Basole, R. C. (2016). Accelerating Digital Transformation: Visual Insights from the API Ecosystem. *IT Professional*, *18*(6), 20–25. https://doi.org/10.1109/MITP.2016.105

Beck, K. (2002). *Test Driven Development: By Example* (1. Edition). Boston, Addison-Wesley Professional.

Beck, K., & Andres, C. (2004). *Extreme Programming Explained: Embrace Change* (2nd edition). Boston, MA, Addison-Wesley Professional.

Bianco, V. D., Myllarniemi, V., Komssi, M., & Raatikainen, M. (2014). The Role of Platform Boundary Resources in Software Ecosystems: A Case Study, In *2014 IEEE/IFIP Conference on Software Architecture*, Sydney, Australia, IEEE. https://doi.org/10.1109/WICSA.2014.41

Billé, R. (2010). Action without change? On the use and usefulness of pilot experiments in environmental management. *S.A.P.I.EN.S. Surveys and Perspectives Integrating Environment and Society*, (3.1).

Boisot, M. H. (1986). Markets and Hierarchies in a Cultural Perspective. *Organization Studies*, *7*(2), 135–158. https://doi.org/10.1177/017084068600700204

Boland, R., Tenkasi, R., & Te'eni, D. (1994). Designing Information Technology to Support Distributed Cognition. *Organization Science*, *5*, 456–475. https://doi.org/10.1287/orsc.5.3.456

Bonardi, M., Brioschi, M., Fuggetta, A., Verga, E. S., & Zuccalà, M. (2016). Fostering collaboration through API economy: The E015 digital ecosystem, In *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice - SER&IP '16*, Austin, Texas, ACM Press. https://doi.org/10.1145/2897022.2897026

Bondel, G., Nägele, S., Koch, F., & Matthes, F. (2020). Barriers for the Advancement of an API Economy in the German Automotive Industry and Potential Measures to Overcome these Barriers: In *Proceedings of the 22nd International Conference on Enterprise Information Systems*, Prague, Czech Republic, SCITEPRESS - Science and Technology Publications. https://doi.org/10.5220/0009353407270734

Boudreau, K. (2011). Let a Thousand Flowers Bloom? An Early Look at Large Numbers of Software App Developers and Patterns of Innovation. *Organization Science*, *23*. https://doi.org/10.2139/ssrn.1826702

Brown, W. J., Malveau, R. C., Iii, H. W. M., & Mowbray, T. J. (1998). *Refactoring Software, Architectures, and Projects in Crisis*. Canada, John Wiley & Sons, Inc.

Buckl, S., Ernst, A. M., Lankes, J., & Matthes, F. (2008). Enterprise Architecture Management Pattern Catalog, 322.

Buckl, S., Matthes, F., Schneider, A. W., & Schweda, C. M. (2013). Pattern-Based Design Research – An Iterative Research Method Balancing Rigor and Relevance. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. vom Brocke, R. Hekkala, S. Ram, & M. Rossi (Eds.), *Design Science at the Intersection of Physical and Virtual Design* (pp. 73–87). Berlin, Heidelberg, Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-38827-9_6

Buschmann, F., Henney, K., & Schmidt, D. C. (2007a). *Pattern Oriented Software Architecture: On Patterns and Pattern Languages*. Chichester, John Wiley & Sons.

Buschmann, F., Henney, K., & Schmidt, D. C. (2007b). *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Volume 4* (1. Edition). Chichester, Wiley.

Calhoun, C. (2011). Communication as Social Science (and More). *International Journal of Communication*, *5*, 1479–1496. https://doi.org/10.1590/S1809-58442012000100014

Chametzky, B. (2016). Coding in Classic Grounded Theory: I've Done an Interview; Now What? *Sociology Mind*, *06*(04), 163–172. https://doi.org/10.4236/sm.2016.64014

Charmaz, K. (2014). *Constructing Grounded Theory* (2. Edition). London ; Thousand Oaks, Calif, SAGE Publications Ltd.

Cook, S. D. N., & Brown, J. (1999). Bridging Epistemologies: The Generative Dance Between Organizational Knowledge and Organizational Knowing. *Organization Science*, *10*, 381–400. https://doi.org/10.1287/orsc.10.4.381

Coplien, J. O. (1994). A Development Process Generative Pattern Language, 34.

Cotton, I. W., & Greatorex, F. S. (1968). Data structures and techniques for remote computer graphics, In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I)*, San Francisco, California, ACM Press. https://doi.org/10.1145/1476589.1476661

Daigneau, R. (2011). *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services* (Illustrated Edition). Upper Saddle River, NJ, Addison Wesley.

De, B. (2017). API Management. In B. De (Ed.), *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization* (pp. 15–28). Berkeley, CA, Apress. https://doi.org/10.1007/978-1-4842-1305-6_2 de Reuver, M., Sørensen, C., & Basole, R. C. (2018). The Digital Platform: A Research Agenda. *Journal of Information Technology*, *33*(2), 124–135. https://doi.org/10.1057/s41265-016-0033-3

Demirkan, H., Kauffman, R. J., Vayghan, J. A., Fill, H.-G., Karagiannis, D., & Maglio, P. P. (2008). Service-oriented technology and management: Perspectives on research and practice for the coming decade. *Electronic Commerce Research and Applications*, *7*(4), 356–376. https://doi.org/10.1016/j.elerap.2008.07.002

Dube, L., & Pare, G. (2003). Rigor In Information Systems Positivist Case Research: Current Practices, Trends, and Recommendations. *MIS Quarterly*, *27*, 597–635. https://doi.org/10.2307/30036550

Eaton, B., Elaluf-Calderwood, S., Sørensen, C., & Yoo, Y. (2015). Distributed Tuning of Boundary Resources: The Case of Apple's iOS Service System. *MIS Quarterly*, *39*(1), 217–243. https://doi.org/10.25300/MISQ/2015/39.1.10

Espinha, T., Zaidman, A., & Gross, H. (2014). Web API growing pains: Stories from client developers and their code, In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. https://doi.org/10.1109/CSMR-WCRE.2014.6747228

European Commission. JRC. (2019). *Web Application Programming Interfaces (APIs): General purpose standards, terms and European Commission initiatives.* (tech. rep.). Publications Office. LU.

Evans. (2003). *Domain-Driven Design: Tacking Complexity In the Heart of Software.* USA, Addison-Wesley Longman Publishing Co., Inc.

Fehling, C., Leymann, F., Retter, R., Schupeck, W., & Arbitter, P. (2014). *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications* (2014. Edition). Wien, Springer.

Fichter, D. (2006). Doing the monster mashup. *Online*, *30*, 48–50.

Fichter, D., & Wisniewski, J. (2009). They Grow Up So Fast: Mashups in the Enterprise. *Online*, *33*, 54–57.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* (Doctoral dissertation).

Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E., & Lau, A. (2011). An Empirical Study on Web Service Evolution, In *2011 IEEE International Conference on Web Services*. https://doi.org/10.1109/ICWS.2011.114

Fowler, M. (2002). *Patterns of Enterprise Application Architecture* (1. Edition). Boston, Addison Wesley.

Fowler, M. (2006). *Writing Software Patterns*. https://martinfowler.com/articles/writingPatterns.html

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software.* Reading, Mass, Prentice Hall.

Gawer, A. (2009). Platforms, Markets and Innovation. *Platforms, Markets and Innovation*. https://doi.org/10.4337/9781849803311

Gawer, A. (2014). Bridging differing perspectives on technological platforms: Toward an integrative framework. *Research Policy*, *43*(7), 1239–1249. https://doi.org/10.1016/j.respol.2014.03.006

Gawer, A., & Cusumano, M. (2014). Industry Platforms and Ecosystem Innovation. *Journal of Product Innovation Management*, *31*. https://doi.org/10.1111/jpim.12105

Germonprez, M., & Hovorka, D. (2013). Member engagement within digitally enabled social network communities: New methodological considerations. *Information Systems Journal*, *23*. https://doi.org/10.1111/isj.12021

Ghazawneh, A., & Henfridsson, O. (2010). GOVERNING THIRD-PARTY DEVELOPMENT THROUGH PLATFORM BOUNDARY RESOURCES, 18.

Ghazawneh, A., & Henfridsson, O. (2013). Balancing platform control and external contribution in third-party development: The boundary resources model. *Information Systems Journal*, *23*. https://doi.org/10.1111/j.1365-2575.2012.00406.x

Glaser, B. G., & Strauss, A. L. (1967). *The discovery of grounded theory: Strategies for qualitative research* (4. paperback printing). New Brunswick, Aldine Publishing Company
OCLC: 553535517.

Haupt, F., Leymann, F., & Vukojevic-Haupt, K. (2018). API governance support through the structural analysis of REST APIs. *Computer Science - Research and Development*, *33*(3-4), 291–303. https://doi.org/10.1007/s00450-017-0384-1

Henfridsson, O., & Bygstad, B. (2013). The Generative Mechanisms of Digital Infrastructure Evolution. *Management Information Systems Quarterly*, *37*(3), 896–931.

Hentrich, C., & Zdun, U. (2011). *Process-Driven Soa: Patterns for Aligning Business and It* (New Edition). Boca Raton, FL, AUERBACH PUBN.

Hevner, A., & Chatterjee, S. (2010). *Design Research in Information Systems* (Vol. 22). Boston, MA, Springer US. https://doi.org/10.1007/978-1-4419-5653-8

Hevner, A., R, A., March, S., T, S., Park, Park, J., Ram, & Sudha. (2004). Design Science in Information Systems Research. *Management Information Systems Quarterly*, *28*, 75.

Hillpot, J. (2020). *Microservices vs Web Services*. https://blog.dreamfactory.com/microservices-vs-web-services/

Hislop, D. (2002). Mission Impossible? Communicating and Sharing Knowledge via Information Technology. *Journal of Information Technology*, *17*(3), 165–177. https://doi.org/10.1080/02683960210161230

Hora, A., Robbes, R., Valente, M. T., Anquetil, N., Etien, A., & Ducasse, S. (2018). How do developers react to API evolution? A large-scale empirical study. *Software Quality Journal*, *26*(1), 161–191. https://doi.org/10.1007/s11219-016-9344-4

Hou, D., & Yao, X. (2011). Exploring the Intent behind API Evolution: A Case Study, In *2011 18th Working Conference on Reverse Engineering*, Limerick, Ireland, IEEE. https://doi.org/10.1109/WCRE.2011.24

Hussain, F., Hussain, R., Noye, B., & Sharieh, S. (2020). Enterprise API Security and GDPR Compliance: Design and Implementation Perspective. *IT Professional*, *22*(5), 81–89. https://doi.org/10.1109/MITP.2020.2973852

IBM Developer Staff. (2018). *APIs versus Services*. https://developer.ibm.com/devpractices/api/articles/api-vs-services-whats-the-difference/

Islind, A. S., Lindroth, T., Snis, U. L., & Sørensen, C. (2016). Co-creation and Fine-Tuning of Boundary Resources in Small-Scale Platformization. In U. Lundh Snis (Ed.), *Nordic Contributions in IS Research* (pp. 149–162). Cham, Springer International Publishing. https://doi.org/10.1007/978-3-319-43597-8_11

Jansen, S., & Cusumano, M. (2012). Defining software ecosystems: A survey of software platforms and business network governance, In *Software Ecosystems*, Edward Elgar Publishing. https://doi.org/10.4337/9781781955635.00008

Jansen, S., Finkelstein, A., & Brinkkemper, S. (2009). A Sense of Community: A Research Agenda for Software Ecosystems, In *2009 31st International Conference on Software Engineering - Companion Volume, ICSE 2009*. https://doi.org/10.1109/ICSE-COMPANION.2009.5070978

Jezek, K., & Dietrich, J. (2017). API Evolution and Compatibility: A Data Corpus and Tool Evaluation. *The Journal of Object Technology*, *16*(4), 2:1. https://doi.org/10.5381/jot.2017.16.4.a2

Josuttis, N. M. (2007). *SOA in Practice: The Art of Distributed System Design* (1. Edition). Beijing ; Sebastopol, O'Reilly and Associates.

Kambil, A. (2008). Purposeful abstractions: Thoughts on creating business network models. *Journal of Business Strategy*, *29*(1), 52–54. https://doi.org/10.1108/02756660810845723

Karhu, K., Gustafsson, R., & Lyytinen, K. (2018). Exploiting and Defending Open Digital Platforms with Boundary Resources: Android's Five Platform Forks. *Information Systems Research*, *29*(2), 479–497. https://doi.org/10.1287/isre.2018.0786

Karmel, A., Chandramouli, R., & Iorga, M. (2016). *NIST Definition of Microservices, Application Containers and System Virtual Machines* (tech. rep. NIST Special Publication (SP) 800-180 (Draft)). National Institute of Standards and Technology.

Kelly, K. (2016). *The Inevitable: Understanding the 12 Technological Forces that Will Shape Our Future*. Viking.

Kendrick, T. (2015). *Identifying and Managing Project Risk: Essential Tools for Failure-Proofing Your Project* (Third Edition). New York, AMACOM.

Kerr, J., & Hunter, R. (1994). *Inside RAD: How to build fully functional computer systems in 90 days or less*. USA, McGraw-Hill, Inc.

Khosroshahi, P. A., Hauder, M., Schneider, A. W., & Florian, D. (2015). Enterprise Architecture Management Pattern Catalog, 140.

Koci, R., Franch, X., Jovanovic, P., & Abello, A. (2019). Classification of Changes in API Evolution, In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, Paris, France, IEEE. https://doi.org/10.1109/EDOC.2019.00037

Krintz, C., & Wolski, R. (2013). Unified API Governance in the New API Economy, 5.

Leach, P. J., Berners-Lee, T., Mogul, J. C., Masinter, L., Fielding, R. T., & Gettys, J. (1999). *Hypertext Transfer Protocol – HTTP/1.1*. https://tools.ietf.org/html/rfc2616

Lemley, M. A., & Cohen, J. E. (2000). Patent Scope and Innovation in the Software Industry. *California Law Review*, *89*(1). https://doi.org/10.2139/ssrn.209668

Limited, A., & Office, T. S. (2019). *ITIL Foundation: ITIL 4 Edition* (4th Edition). TSO.

Löhe, J., & Legner, C. (2010a). SOA adoption in business networks: Do service-oriented architectures really advance inter-organizational integration?, 16.

Löhe, J., & Legner, C. (2010b). SOA Adoption in Business Networks: Does SOA live up to High Expectations?, 15.

Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U., & Stocker, M. (2019). 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*, Irsee, Germany, ACM Press. https://doi.org/10.1145/3361149.3361164

Luthria, H., & Rabhi, F. (2009). Using Service Oriented Computing for Competitive Advantage, 10.

Manikas, K., & Hansen, K. M. (2013). Software ecosystems – A systematic literature review. *Journal of Systems and Software*, *86*(5), 1294–1306. https://doi.org/10.1016/j.jss.2012.12.026

Mathijssen, M., Overeem, M., & Jansen, S. (2020). Identification of Practices and Capabilities in API Management: A Systematic Literature Review. *arXiv:2006.10481 [cs]*arxiv 2006.10481.

Maximilien, E. M., Ranabahu, A., & Gomadam, K. (2008). An Online Platform for Web APIs and Service Mashups. *IEEE Internet Computing*, *12*(5), 32–43. https://doi.org/10.1109/MIC.2008.92

Medjaoui, M., Wilde, E., Mitra, R., & Amundsen, M. (2018). *Continuous API Management: Making the Right Decisions in an Evolving Landscape*. Sebastopol, CA, O'Reilly UK Ltd.

Mulloy, B. (2012). *Web API Design - Crafting Interfaces that Developers Love*.

Newman, R., & Newman, J. (1985). Information Work: The New Divorce? *The British Journal of Sociology*, *36*(4), 497–515. https://doi.org/10.2307/590328

Nicholls-Nixon, C. L., & Woo, C. Y. (2003). Technology Sourcing and Output of Established Firms in a Regime of Encompassing Technological Change. *Strategic Management Journal*, *24*(7), 651–666.

Nonaka, I., & Takeuchi, H. (1995). *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation* (Illustrated Edition). New York, Oxford University Press.

OpenAPI Initiative. (2020). *OpenAPI Specification Repository*. https://github.com/OAI/OpenAPI-Specification

Osterwalder, A., Pigneur, Y., Bernarda, G., & Smith, A. (2014). *Value Proposition Design: How to Create Products and Services Customers Want* (1. Edition). Hoboken, Wiley.

Pahl, C., Jamshidi, P., & Zimmermann, O. (2017). Architectural Principles for Cloud Software. *ACM Transactions on Internet Technology*, *18*. https://doi.org/10.1145/3104028

Pallis, G. (2010). Cloud Computing: The New Frontier of Internet Computing. *IEEE Internet Computing*, *14*(5), 70–73. https://doi.org/10.1109/MIC.2010.113

Papazoglou, M. P., & Georgakopoulos, D. (2003). Introduction: Service-oriented computing. *Communications of the ACM*, *46*(10), 24–28. https://doi.org/10.1145/944217.944233

Papazoglou, M. P. (2008). *Web services: Principles and technology*. Harlow, Pearson-/Prentice Hall
OCLC: 255863191.

Papazoglou, M. P., & van den Heuvel, W.-J. (2007). Service oriented architectures: Approaches, technologies and research issues. *The VLDB Journal*, *16*(3), 389–415. https://doi.org/10.1007/s00778-007-0044-3

Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., & Josuttis, N. (2017). Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software*, *34*(1), 91–98. https://doi.org/10.1109/MS.2017.24

Pautasso, C., Ivanchikj, A., & Schreier, S. (2016). A pattern language for RESTful conversations, In *Proceedings of the 21st European Conference on Pattern Languages of Programs*, New York, NY, USA, Association for Computing Machinery. https://doi.org/10.1145/3011784.3011788

Plattner, H., Meinel, C., & Leifer, L. (2010). *Design Thinking: Understand – Improve – Apply* (2011. Edition). Berlin, Springer.

Ranabahu, A., Patel, P., & Sheth, A. (2009). *Service Level Agreement in Cloud Computing*.

RapidAPI. (2019). *API vs SDK*. https://rapidapi.com/blog/api-vs-sdk/

Red Hat, Inc. (2021). *What is API management?* https://www.redhat.com/en/topics/api/what-is-api-management

Ries, E. (2011). *The Lean Startup: How Constant Innovation Creates Radically Successful Businesses* (Trade Paperback Edition). London, Portfolio Penguin.

Roberts, M., & Chapin, J. (2017). *What is Serverless?* O'Reilly Media, Inc.

Rotem-Gal-Oz, A. (2012). *SOA Patterns* (1st Edition). Shelter Island, NY, Manning Publications.

Scarbrough, H. (1995). Blackboxes, Hostages and Prisoners. *Organization Studies - ORGAN STUD*, *16*, 991–1019. https://doi.org/10.1177/017084069501600604

Shnier, M. (1996). *Dictionary of PC hardware and data communications terms.*

Skog, D. A., Wimelius, H., & Sandberg, J. (2018). Digital Service Platform Evolution: How Spotify Leveraged Boundary Resources to Become a Global Leader in Music Streaming, In *51*.

Sohan, S., Anslow, C., & Maurer, F. (2015). A Case Study of Web API Evolution, In *2015 IEEE World Congress on Services*, New York City, NY, USA, IEEE. https://doi.org/10.1109/SERVICES.2015.43

Sørensen, C., & Snis, U. (2001). Innovation through Knowledge Codification. *Journal of Information Technology*, *16*, 83–97. https://doi.org/10.1080/02683960011005477I

Strauss, A., & Corbin, J. (1998). *Basics of qualitative research: Techniques and procedures for developing grounded theory, 2nd ed*. Thousand Oaks, CA, US, Sage Publications, Inc.

Takeuchi, H., & Nonaka, I. (1986). The New New Product Development Game. *Harvard Business Review*.

Tan, W., Fan, Y., Ghoneim, A., Hossain, M. A., & Dustdar, S. (2016). From the Service-Oriented Architecture to the Web API Economy. *IEEE Internet Computing*, *20*(4), 64–68. https://doi.org/10.1109/MIC.2016.74

Thomas, L., Autio, E., & Gann, D. (2014). Architectural Leverage: Putting Platforms in Context. *Academy of Management Executive*, *28*, 198–219. https://doi.org/10.5465/amp.2011.0105

Uludağ, Ö., Harders, N.-M., & Matthes, F. (2019). Documenting recurring concerns and patterns in large-scale agile development, In *Proceedings of the 24th European Conference on Pattern Languages of Programs - EuroPLop '19*, Irsee, Germany, ACM Press. https://doi.org/10.1145/3361149.3361176

Urquhart, C., Lehmann, H., & Myers, M. D. (2009). Putting the 'theory' back into grounded theory: Guidelines for grounded theory studies in information systems: Guidelines for grounded theory studies in information systems. *Information Systems Journal*, *20*(4), 357–381. https://doi.org/10.1111/j.1365-2575.2009.00328.x

Voelter, M., Kircher, M., & Zdun, U. (2004). *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. Chichester, West Sussex, England ; Hoboken, NJ, John Wiley & Sons Ltd.

Walker, G. (2001). *IT Problem Management* (1. Edition). Upper Saddle River, NJ, Prentice Hall.

Webster, J., & Watson, R. T. (2002). Analyzing the Past to Prepare for the Future: Writing a Literature Review. *MIS Quarterly*, *26*(2), xiii–xxiii.

Weir, L., & Nemec, Z. ". (2019). *Enterprise API Management: Design and deliver valuable business APIs*. Packt Publishing.

Wiesche, M., Jurisch, M. C., City of Munich, Yetton, P. W., Deaken University, Krcmar, H., & Technische Universität München. (2017). Grounded Theory Methodology in Information Systems Research. *MIS Quarterly*, *41*(3), 685–701. https://doi.org/10.25300/MISQ/2017/41.3.02

Yoo, Y., Henfridsson, O., & Lyytinen, K. (2010). Research Commentary —The New Organizing Logic of Digital Innovation: An Agenda for Information Systems Research. *Information Systems Research*, *21*(4), 724–735. https://doi.org/10.1287/isre.1100.0322

Yu, S., & Woodard, C. J. (2009). Innovation in the Programmable Web: Characterizing the Mashup Ecosystem. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, B. J. Krämer, K.-J. Lin, & P. Narasimhan (Eds.), *Service-Oriented Computing – ICSOC 2007* (pp. 136–147). Berlin, Heidelberg, Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-01247-1_13

Zhu, W.-D., Andrew J, D., Andrew A, D., Dickerson, S., Falkl, J., Sanders, K., Shetty, D. G., & Wood, C. (2014). *Exposing and Managing Enterprise Services With IBM API Management*. Poughkeepsie, N.Y., Redbooks.

Zimmermann, O. (2017). Microservices tenets. *Computer Science - Research and Development*, *32*(3-4), 301–310. https://doi.org/10.1007/s00450-016-0337-0

Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., & Zdun, U. (2020). Introduction to Microservice API Patterns (MAP), 17 pages. https://doi.org/10.4230/OASICS.MICROSERVICES.2017-2019.4

Zimmermann, O., Stocker, M., Lübke, D., & Zdun, U. (2017). Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. https://doi.org/10.1145/3147704.3147734

Zittrain, J. (2006). The Generative Internet. *Harvard Law Review*. https://doi.org/10.1145/1435417.1435426

# Appendix

## 1 Interview Guide

### Introduction

A growing number of companies offer resources through web APIs instigating the API Economy. Web APIs enable value-adding composition of services that allow new business models. API providers have to manage web APIs carefully to incorporate changes in the ecosystems while securing internal interests. Key papers have identified a lack of research about web APIs and stress the importance of longitudinal data. This thesis aims to identify day-to-day issues and actions of API providers through a longitudinal study. The findings will be used to develop pattern candidates that have been discussed with industry experts and API providers.

### Terminology

- **API provider**, the entity that provides an API
- **API consumer**, the costumer that accesses the capabilities of the API
- **Web API**, APIs that are accessible over the web
- **Public API**, APIs that are accessible to third-party developers outside the organization
- **Private API**, APIs that are accessible inside the organization

### Motivation and Format

The purpose of this interview is to identify common tasks and challenges of API management and corresponding solution approaches. The interview is planned to be 30 minutes. The interviewee can agree to a set of follow-up interviews to discuss issues, solutions, and activities that emerged since the last meeting. The follow-up interview is meant to be 15-30 minutes.

### Terms of Confidentiality

The study data will be completely anonymized. We will only connect the following information to the results:

- A short classification of your company
- Your role(s)

This interview will be recorded to be transcribed right after the interview. We will delete the audio/video recording afterwards. Do you agree to recording of this interview? (Yes / No)

Do you have any questions before we start the interview?

### Kick-off Questions

- How long have you been working in IT?
- How old is the API you are working on? Is it released yet?
- Who is involved in the maintenance and development of the API?
- What processes are used for change requests and where do the requirements come from?
- Who is using the API that you are developing?
- How does the communication and collaboration with the API consumers look like?

### Current Work

- What are you and your team currently working on?

### Follow-up Questions

- Did you resolve the issue?
- Did it take more or less time than expected? Why do you think that happened?
- Did you communicate the updates with your API consumers? How?
- Were any lessons learned from fixing those issues?

# 2 Pattern Catalog

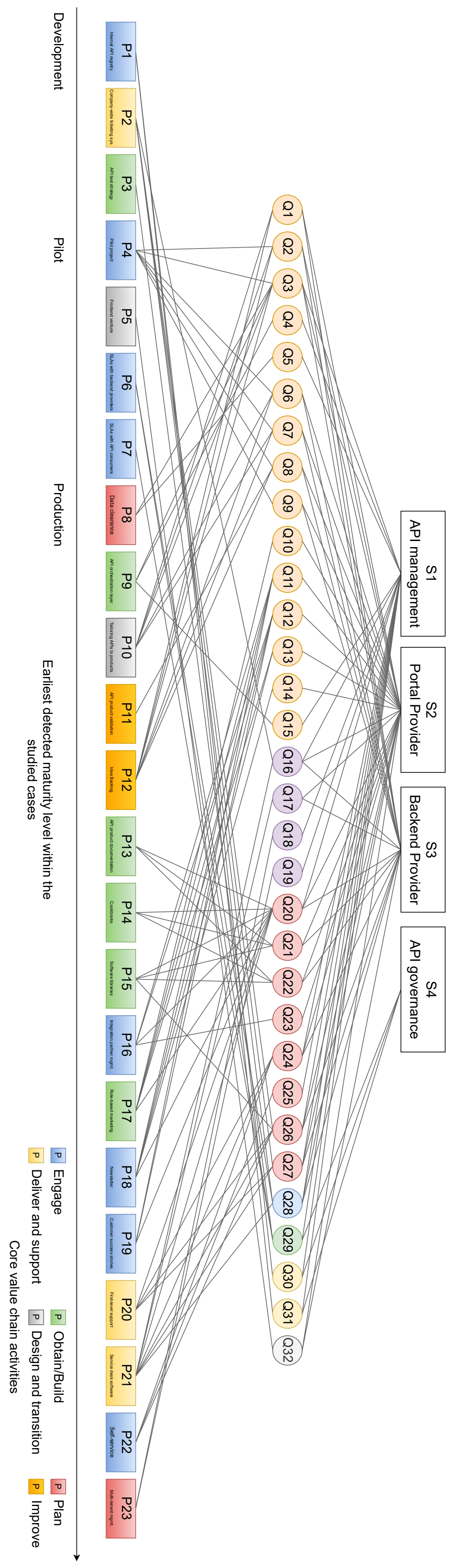


Figure 1: Pattern Catalog Taxonomy - Appendix Version