Microservice API Evolution in Practice: A Study on Strategies and Challenges☆

Alexander Lercher *, Johann Glock, Christian Macho, Martin Pinzger

Department of Informatics Systems, University of Klagenfurt, Universitätsstraße 65-67, Klagenfurt, 9020, Austria

ARTICLE INFO

ABSTRACT

Nowadays, many companies design and develop their software systems as a set of loosely coupled microservices that communicate via their Application Programming Interfaces (APIs). While the loose coupling improves maintainability, scalability, and fault tolerance, it poses new challenges to the API evolution process. Related works identified communication and integration as major API evolution challenges but did not provide the underlying reasons and research directions to mitigate them. In this paper, we aim to identify microservice API evolution strategies and challenges in practice and gain a broader perspective of their relationships. We conducted 17 semi-structured interviews with developers, architects, and managers in 11 companies and analyzed the interviews with open coding used in grounded theory. In total, we identified six strategies and six challenges for REpresentational State Transfer (REST) and event-driven communication via message brokers. The strategies mainly focus on API backward compatibility, versioning, and close collaboration between teams. The challenges include change impact analysis efforts, ineffective communication of changes, and consumer reliance on outdated versions, leading to API design degradation. We defined two important problems in microservice API evolution resulting from the challenges and their coping strategies: tight organizational coupling and consumer lock-in. To mitigate these two problems, we propose automating the change impact analysis and investigating effective communication of changes as open research directions.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Many modern software systems are split into loosely coupled services to improve maintainability, scalability, and fault tolerance (Gos and Zabierowski, 2020). Service-oriented Architecture (SOA) (Krafzig et al., 2006) was one method to distribute a large monolithic software system into multiple smaller services. SOA relied on shared business models and centralized communication over an Enterprise Service Bus (ESB) (Cerny et al., 2018). Consequently, the individual services were tightly coupled, and introducing changes required integration and deployment coordination throughout the system's services (Bushong et al., 2021). This led companies to migrate from SOA to Microservice Architecture (MSA). MSA replaced the shared models of services with independent domain models exposed only via Application Programming Interfaces (APIs) for each so-called microservice, and the ESB with message brokers only forwarding serialized messages (Zhang et al., 2019). Microservices exposing their functionality via APIs are called *providers* and microservices calling and interacting with these

APIs are called *consumers*. The communication approach in MSA is referred to as "smart endpoints and dumb pipes" and loosely couples the microservices via such well-defined APIs, allowing them to evolve independently within a system (Wu et al., 2022).

However, maintaining the overall systems' functionality requires more synchronization efforts between the development teams as each microservice's data structures and business logic, i.e., behaviors, evolve independently (Ma et al., 2019). Unlike a single monolithic code base or shared SOA interfaces, the loose coupling prohibits developers from learning about individual microservice API changes at compile time. If provider teams do not notify consumer teams about changes in advance, the breaking changes only manifest during the first actual API call at runtime. Changes in the provider's API could then result in unexpected behavior and potentially break the execution of dependent consumers interacting with that API.

Previous studies (Alshuqayran et al., 2016; Söylemez et al., 2022) identified communication and integration as major challenges in the

---

MSA. Similarly, Cerny et al. (2018) identified communicating the API changes to dependent teams, i.e., consumers, and testing for incompatibilities as the primary two open research challenges in service integration. According to Zdun et al. (2020), the API evolution process still misses effective communication and support for consumers affected by changes. Assunção et al. (2023) found that many developers waste their time with implementing technical API changes and updates instead of focusing on business logic. Hence, microservice API evolution requires more research (Lamothe et al., 2021). While related works acknowledged the challenges of API evolution, integration, and communication of changes, they did not provide the underlying reasons or how to solve them sustainably.

In this paper, we aim to understand current microservice API evolution strategies and challenges in practice and to gain a broader perspective of their relationships for future research directions. For this, we defined the following three research questions:

RQ1 What means do developers use to exchange messages between services, and how do they document them?
RQ2 Which strategies do developers follow to introduce and communicate API changes in loosely coupled systems?
RQ3 Which challenges do developers face when introducing and communicating API changes in loosely coupled systems?

For answering the three questions, we conducted semi-structured interviews (Adams, 2015) with practitioners from multiple companies, analyzed the interviews with open coding (Corbin and Strauss, 1990), and grounded our qualitative research results with related literature. Through this, we identified (a) REpresentational State Transfer and event-driven communication as the main communication techniques in MSA, (b) six API evolution strategies formulated as best practices for practitioners, (c) six API evolution challenges to consider as pitfalls when designing an MSA, (d) two important problems namely tight organizational coupling and consumer lock-in, and (e) two directions for future research to address these problems and improve microservice API evolution. We provide a replication package (Lercher et al., 2023) comprising the interview guide and resulting code book.

To the best of our knowledge, this is the first study investigating microservice API evolution strategies and challenges in practice to create a comprehensive list of best practices and pitfalls and to derive two important underlying problems and open research directions to mitigate them. We focus on loosely coupled systems based on the MSA communication approach "smart endpoints and dumb pipes" and use the terms *service* and *microservice* interchangeably. Furthermore, we use the term *API* when referring to the communication techniques in MSA for simplicity.

The remainder of this paper is structured as follows. Section 2 describes the method of semi-structured interviews, our study design, data analysis, and participant selection. Section 3 presents the message exchange techniques used in practice and answers RQ1. In Sections 4 and 5, we present the identified evolution strategies and challenges in practice and answer RQ2 and RQ3. Section 6 presents the overall theory of our study results by defining two important problems in API evolution. We propose future research directions and discuss the limitations in Section 7. Section 8 presents related works on API evolution and Section 9 concludes our study.

2. Methodology

In this section, we describe our study design, data analysis, and participant selection.

2.1. Study design

Due to the open-ended and explorative nature of our research questions, we conducted semi-structured interviews (Adams, 2015; Gudkova, 2018) with developers, architects, and management roles directly working on MSA or similar loosely coupled systems. The semi-structured format enabled the participants to express their thoughts freely and provided us the opportunity to ask follow-up questions and discuss emerging topics ad-hoc while maintaining the desired dialogue direction. This allowed us to not only learn about applied strategies and encountered challenges individually but also to understand their underlying reasons and consequences.

We formulated an interview guide focusing on answering our research questions to serve as an orientation during the interviews. The interview guide consisted of five question categories: (a) background, (b) communication, targeting RQ1, (c) API evolution as provider, targeting RQ2 and RQ3, (d) API evolution as consumer, targeting RQ2 and RQ3 from a different perspective explained below, and (e) additional thoughts.

The background category elicits the participants' education, experience, subjective definition of a microservice, and details about their work environments. These questions help to set the context and clarify the terminology used by the interviewer and participant during each interview, thereby improving the construct validity of our study (Wohlin et al., 2012).

The communication category aims at answering RQ1. It focuses on the communication approaches, which and how microservice APIs are exposed, and how they are documented.

The two API evolution categories explicitly illuminate the provider and consumer sides in API evolution to answer RQ2 and RQ3. During preliminary discussions, we realized that developers do not think about the evolution of external APIs but expect unlimited availability of the consumed API version. Hence, we decided to explicitly split the perspectives on provided APIs, i.e., APIs developed and maintained by the interview participants' teams, and consumed APIs, i.e., APIs interacted with by the participants' teams without access to the source code or runtime environment. The categories contain open questions regarding the frequency of provided and consumed APIs' changes, the reasons for these changes, the strategies for communicating and implementing them, the strategies for notifying other teams about changes, the strategies for getting informed on changes, the challenges they encountered during each of these tasks, and general improvement ideas.

Finally, the interview guide concludes with the question "Do you have additional thoughts you want to express?" sometimes triggering multiple more minutes of dialogue. We used this question to encourage the participants to discuss additional topics we did not consider but that they think are important.

We followed established guidelines for qualitative research (Adams, 2015; Goodrick and Rogers, 2015) and refined the interview guide two times. After an initial pilot interview, we moved the questions for general improvement ideas from the last category into the provider and consumer API change categories to improve the interview flow. After the fourth interview, we added a background question about the used development and deployment technologies to have a clear picture of the participants' systems.

In total, we conducted 20 interviews but excluded three from the results (cf. Section 2.3). We designed the interview guide to last between 60-90 minutes. Depending on the available time frame and involvement of the participants, the interviews lasted around 71 min (min = 52; max = 92; mean = 70.9; median = 71 minutes). Due to the SARS-CoV-2 pandemic and physical distance, we conducted 11 interviews via online videoconference and 6 interviews in person. We did not observe noticeable differences in the openness or involvement of the participants between these two modes.

2.2. Interview analysis

We analyzed the interviews qualitatively to answer the open-ended research questions. First, we recorded each interview and transcribed it verbatim. Then, we applied open coding (Corbin and Strauss, 1990) used in grounded theory (Glaser and Strauss, 1967; Adolph et al., 2011). In this method, individual interview statements, e.g., *"You already have a set of test cases that you can run against the old interface. You will see immediately when you introduce something that breaks it"*, are labeled with matching codes, e.g., test the interface on changes. The codes are then assigned to categories, e.g., contract testing, which themselves form a hierarchy, e.g., contract testing is a subcategory of the regression testing strategy.

The first author analyzed all interview transcripts statement by statement, identified the codes, and organized them into a hierarchy of categories. We applied investigator triangulation (Campbell et al., 2013), i.e., the second and third authors analyzed two random interview transcripts independently and we discussed the identified categories to increase the result quality (O'Connor and Joffe, 2020). We achieved coder agreement after short discussions, mainly on the phrasing of categories with the same meaning. We analyzed all interviews iteratively, i.e., using the resulting codebook from the previous session for the next interview transcript. After 12 analyzed interviews, the codebook began to stabilize, i.e., we only found a few new categories for the following two interviews and the last three interviews did not add any new categories but instead only repeated existing ones. Hence, we reached theoretical saturation (van Rijnsoever, 2017).

We structured the categories into the following topics: background, communication and documentation, API evolution strategies, API evolution challenges, and improvement ideas. This structure allowed us to answer the research questions directly from the code book. Additionally, we used the findings to build an overall theory of two important underlying problems in microservice API evolution. In this paper, we used the format (i/17) to indicate the number i of participants supporting a finding.

We applied member checking (Runeson et al., 2012) by sharing the study results with our interview participants for feedback and validation. Therefore, we created a draft report and per participant highlighted all findings and statements where we considered their answers. We sent out the 17 individually highlighted reports and received 13 responses. Two participants had minor remarks which we incorporated and the others fully agreed with our interpretations.

Finally, we identified related works and literature for our findings and included them per result category. This approach helped to support or reject our qualitative results, put them into relation, and strengthen the overall theory.

2.3. Participant selection

Our participants had to be developers, architects, or managers working on developing loosely coupled services exposing an API, e.g., Representational State Transfer (REST) or event-driven communication, for at least one year. Similarly to other studies (Safwan and Servant, 2019; García et al., 2020), we contacted previous colleagues and applied snowball sampling (Biernacki and Waldorf, 1981), i.e., asked them to forward our interview request to their peers matching our requirements as potential participants. Considering the explorative nature of the study, this sampling technique is sufficiently effective for theoretical saturation (Baltes and Ralph, 2022).

We continuously advertised our call for interview participants to colleagues while conducting and analyzing the scheduled interviews. In total, we contacted 25 colleagues directly and stopped sending out additional requests once our codebook reached saturation. We only accepted a maximum of three interview partners per company on a first-come, first-served basis. Through the snowball sampling, we conducted 20 interviews with participants from 12 companies but excluded

three of the interviews from the results. One excluded participant (C4-P1) worked in a team of only two developers, who created their API solely for the front end and, hence, handled API evolution like any other internal source code change. Another participant did not introduce breaking changes to their product's APIs yet and did not consume any external APIs. The third excluded participant learned of our intermediate results and was excluded to avoid biased answers.

In total, we report on the results of $n = 17$ interviews from 11 companies. All participants are industry practitioners with an average practical experience of 10 years (min = 2; max = 25; mean = 10.2; median = 10 years) and an average of 4.5 years of practical experience with loosely coupled systems (min = 1; max = 7; mean = 4.6; median = 5 years). Their highest relevant education ranges from a technical high school diploma to a doctoral degree (Ph.D.). The technical roles include six developers, four senior developers, and seven architects, including two principal architects. The participants work on three types of loosely coupled systems: MSA, loosely coupled services that they consider too large to be called microservices, and self-contained systems (Cerny et al., 2017) where multiple microservices share a code base. We consider all systems loosely coupled because they use the MSA's smart endpoints and dumb pipes communication approach, which allows the components to evolve independently. Table 1 contains the details about the individual participants. Their companies focus on various industry fields, and we classified the company sizes based on their staff headcount according to the European Union small and medium-sized enterprises definition.[1] Table 2 contains the company details.

3. Message exchange techniques (RQ1)

This section presents the message exchange techniques used in practice and their corresponding documentation techniques, which we elicited with the communication questions of our interview guide. Hence, this section answers RQ1: *What means do developers use to exchange messages between services, and how do they document them?*

3.1. Answer to RQ1

The two most popular message exchange techniques among the participants are Representational State Transfer (REST) APIs and event-driven communication. On average, REST APIs make up 66.8% of the total communication (min = 5%; max = 100%; mean = 66.8%; median = 85%) and event-driven communication makes up 22.6% of the total communication (min = 0%; max = 95%; mean = 22.6%; median = 10%). Some participants provide and maintain Simple Object Access Protocol (SOAP) APIs (min = 0%; max = 60%; mean = 12.8%; median = 0% of the total communication). However, they no longer develop new SOAP APIs but only maintain existing ones for legacy consumers and plan to discontinue them once all consumers migrated. Fig. 1 visualizes the proportion of the three communication techniques among the interview participants as violin plots. All participants use Open-API and Swagger tools to document their REST APIs automatically. Additionally, many participants manually supplement this documentation with wiki pages or in-line source code documentation. The participants refrain from documenting the event-driven communication formally because it targets system-internal services with well-known maintainers.

Notably, we heard of specialized protocols such as GraphQL[2] for API querying, Websockets for bidirectional communication, and Google Protocol Buffers[3] for serialization. However, only a maximum of two participants mentioned them, and hence, we did not include them in the detailed results. In the following, we present the details of the two main communication techniques.

---

[1] https://single-market-economy.ec.europa.eu/smes/sme-definition_en.
[2] https://graphql.org/.
[3] https://protobuf.dev.

Table 1
Backgrounds of the interviewed participants. The principal architects coordinate multiple systems and teams.

Participant code	Highest education	Technical role	Exp[a] (yrs)	Exp[b] (yrs)	System architecture	System size[c]	Team size
C1-P1	Bachelor	Developer	7	3	MSA and monolith	25	6
C2-P1	Ph.D.	Principal architect	13	6	MSA	–	–
C2-P2	Ph.D.	Architect	10	6	MSA	50	6
C2-P3	Master	Architect	10	6	MSA	50	6
C3-P1	Master	Architect	10	3	Services	6	10
C3-P2	Bachelor	Developer	4	4	Services	7	10
C3-P3	Technical high school	Developer	7	5	Services	20	17
C5-P1	Master	Senior developer/Technical lead	10	4	Services	30	5
C5-P2	Technical high school	Senior developer	7	6	Services	30	5
C6-P1	Bachelor	Senior developer/Technical lead	15	3	MSA	15	15
C7-P1	Master	Developer	4	3	Services and monolith	3	6
C8-P1	Bachelor	Developer	6	3	MSA	10	10
C9-P1	Ph.D.	Developer	2	1	MSA and Function-as-a-Service	70	3
C10-P1	Master	Architect	14	7	MSA and monolith	15	7
C11-P1	Master	Architect	9	7	Self-contained systems and monolith	20	6
C11-P2	Master	Principal architect/Department head	20	7	Self-contained systems and monolith	–	–
C12-P1	Technical high school	Senior developer/Product manager	25	5	Services	40	13

[a] Total practical experience.
[b] Practical experience with loosely coupled systems.
[c] Number of loosely coupled components according to the system architecture.

Table 2
Backgrounds of the companies with the number of interview participants.

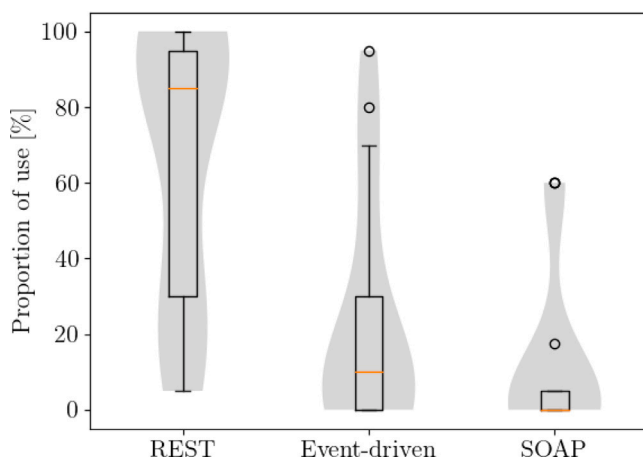| Company code	Industry field	Company size	# P
C1	Construction	Large	1
C2	Access management	Large	3
C3	Automotive	Large	3
C5	Video processing	Medium-sized	2
C6	Retail	Large	1
C7	Monitoring	Large	1
C8	Process digitization	Small	1
C9	E-commerce	Small	1
C10	Traffic management	Large	1
C11	Research and higher education	Large	2
C12	E-mobility	Large	1

Fig. 1. Proportion of communication techniques of the total communication among interview participants.

3.2. Representational state transfer (REST)

All interview participants (17/17) provide REST APIs for their services and transfer messages serialized into JavaScript Object Notation (JSON).

3.2.1. REST APIs

The participants consider REST APIs a de facto standard for service communication. They highlighted the ease of use and little setup time for consumers, considering most developers are already familiar with REST. Furthermore, most REST API frameworks support authorization protocols such as OAuth 2.0 out-of-the-box. Hence, many participants (9/17) explicitly state that they exclusively use REST for public-facing APIs to customers. A few participants (3/17) provide client SDKs abstracting the REST calls, but this approach increases the maintenance overhead with each additionally supported development language.

Similar to the participants, the literature considers REST (Fielding, 2000) the de facto standard to publicly expose request–response APIs for accessing data and computing resources of web services (Kratzke and Quint, 2017; Zimmermann et al., 2020).

According to Aksakalli et al. (2021), REST APIs have the advantage of easy implementation but require well-defined request–response data structures and the availability of both the provider and consumer services at call time to process a request correctly.

3.2.2. REST API gateways

Many participants (9/17) implement dedicated API gateways that handle all incoming REST API requests. Their API gateways abstract the individual services' APIs and versioning by hiding the internal architecture and only providing a single access point for external consumers, enabling system transparency and loose coupling. Furthermore, the API gateways centrally manage the authentication of requests and eliminate the redundancy of implementing it for each service's API individually. Interestingly, most developers and senior developers (8/10) described an API gateway, while only one architect (1/7) mentioned using it. We assume that the REST API gateway is a typical implementation detail and is mainly used and considered by developers.

A REST API gateway implements the *Facade* pattern (Gamma et al., 1995) on component level (Zdun et al., 2017). Taibi et al. (2018) recommend the API gateway as an extensible and backward-compatible orchestration and coordination pattern, and an MSA without an API gateway is considered bad practice (Taibi and Lenarduzzi, 2018; Akbulut and Perros, 2019). As a disadvantage, the single API gateway is a potential bottleneck. Load balancing techniques (Taibi et al., 2018) and resiliency patterns (Mendonca et al., 2020), such as Retry and Circuit Breakers, mitigate this disadvantage but increase the development and runtime complexity.

3.2.3. OpenAPI and Swagger documentation

All participants (17/17) use the OpenAPI specification[4] to define and document their REST APIs formally. The OpenAPI specification allows the clear documentation and versioning of the REST API. They further use Swagger tools[5] to generate and visualize the OpenAPI specification in the browser automatically. Many participants (9/17) share the OpenAPI specification also with external consumers, who are typically familiar with the format or even use OpenAPI themselves. In particular, many developers (7/10) recommended sharing the OpenAPI specification, including all four senior developers. Moreover, some participants (5/17) use the OpenAPI and Swagger capabilities to automatically generate server code, consumer code, contract tests, and client SDKs.

OpenAPI is a vendor-neutral description format by the Linux Foundation and de-facto standard in the industry. Neumann et al. (2021) analyzed 500 REST APIs and found that almost half automatically generate the specification with Swagger. Various practical tools[6] and research approaches (Koren and Klamma, 2018; Peng et al., 2018; Ed-Douibi et al., 2020; Karlsson et al., 2020) utilize the OpenAPI specification format. However, the OpenAPI specification only describes the API structure, not the API behavior, e.g., authentication and relations between message fields.

3.2.4. Supplementary manual documentation

Many participants (9/17) supplement the OpenAPI documentation with manual documentation about the REST API behavior in written form or UML diagrams. *"It's not enough to just show the REST interface and the parameters, you have to know some business context around it"* C3-P1. The participants mainly use wiki pages, e.g., Confluence (5/17), to manage supplementary documentation, where they also link to previous API versions and the OpenAPI specification. The supplementary documentation explains the authentication processes, message fields' semantics and relationships, and error handling and recovery options. While the senior developers did not mention manual documentation, both principal architects and another architect even maintain manual documentation of the REST API structure, e.g., REST endpoints, parameters, and example calls. Swagger tools could generate such documentation automatically, however, in a more technical format. Hence, we presume architects prefer a manually created overview compared to the technical OpenAPI documentation with implementation details.

OpenAPI only serves as a specification for a REST API's structure and input and output formats. The API's behavior and semantics are often documented in natural language or even missing. For instance, the relationship between the two fields balance value and tax flag could vary. If the tax flag is set, it could mean the tax value was added to the balance. Alternatively, it could mean that the balance is deductible. Such resulting ambiguity of behavior and semantics complicates integrating multiple services with different contexts and domain vocabulary (Cremaschi and De Paoli, 2017). Schwichtenberg et al. (2017) proposed an approach to derive the semantics by semi-automatically matching the OpenAPI structure with public ontology concepts. Still, the OpenAPI does not document the API behavior.

3.2.5. Internal source code documentation

Notably, some participants (6/17) do not document system-internal REST APIs on the API level. Instead, they prefer reading the source code and in-line code documentation directly, which they consider faster than loading the Swagger-generated documentation and manually identifying the semantics and behavior from it. This preference was only expressed by participants not working on MSA but larger services,

where the whole code base is already loaded in the development environment.

This documentation strategy for system-internal functionality follows conventional development practices and is unrelated to loosely coupled systems. Hence, we refer to conventional source code documentation research, e.g., Shmerlin et al. (2015).

3.3. Event-driven communication

Many participants (13/17) use event-driven communication patterns, such as publish–subscribe and message queues, system-internally to send asynchronous messages via message brokers. We identified that the participants with a team size of more than ten do not utilize event-driven communication. Considering its typically limited documentation, we presume that this communication technique creates more implicit knowledge than REST APIs, posing problems for larger teams.

3.3.1. Asynchronous messaging and message brokers

The participants (13/17) use asynchronous messaging whenever real-time responses are not required and eventual consistency is acceptable, e.g., on state updates or completion notifications of long-running processes. *"It's the cloud. It's async anyways, why not make it explicit?"* C5-P1. The participants mainly use the two message broker technologies RabbitMQ[7] (6/17) and Apache Kafka[8] (3/17). Some participants (6/17) emphasize that this abstraction layer helps to loosely couple the system. New services are easily added and removed as publishers and subscribers without adapting any other services. Similarly, asynchronous messaging helps integrate services with an existing monolith by notifying them about internal events without altering the monolith's original program flow. Consequently, all participants maintaining a monolith as part of their system use event-driven communication, except for one (4/5).

Asynchronous communication via message brokers, a type of simple message-oriented middleware (Yongguo et al., 2019; Sommer et al., 2018), decouples services during development, deployment, and runtime. RabbitMQ and Apache Kafka are two of the most popular message broker technologies (Yongguo et al., 2019). They automatically store and distribute messages for immediate or later consumption by any subscribed service. Contrary, calling a REST API requires the explicit knowledge and availability of all directly called services. As a downside, introducing message brokers increases the system's complexity because they hide communication paths and dependencies between the services (Aksakalli et al., 2021).

3.3.2. Limited documentation of event-driven communication

Contrary to REST API documentation, only a few participants (3/17) explicitly document the event-driven communication, and only one of them uses the AsyncAPI[9] specification. We theorize most participants do not formally document their event-driven communication because it aims at system-internal communication. Hence, the audience for such documentation consists mainly of other company-internal developers with access to the source code (cf. Section 3.2.5). *"Actually, it's quite easy to look into the services to get an idea how the payloads look like and so on"* C5-P2. In contrast, REST APIs are open to external consumers and customers with less technical or domain-specific backgrounds.

AsnycAPI is the de-facto industry standard for documenting message-based communication (Gómez et al., 2020). Interestingly, it is not well recognized by our interview participants who rely on the source code instead. Aksakalli et al. (2021) stated that the advantage of the MSA is eliminated once the system dependencies become

---

[4] https://spec.openapis.org/oas/v3.1.0.
[5] https://swagger.io/tools/.
[6] https://openapi.tools.

[7] https://www.rabbitmq.com.
[8] https://kafka.apache.org.
[9] https://www.asyncapi.com/docs/reference/specification/v2.6.0.

Table 3
The API evolution strategies with participant and company counts.

API evolution strategy	# P	# C	Perspective
Accept necessary breaking changes	17	11	Provider
Understand the reasons for breaking changes	17	11	Provider
Consider structural and behavioral changes	5	4	Provider
Stay compatible and avoid unexpected breaking changes	17	11	Provider
Work around breaking changes	17	11	Provider
Regression test the API	10	8	Provider
Think ahead and design a dynamic API	6	6	Provider
Version the API	17	11	Provider
Create a new version on breaking changes	17	11	Provider
Expose multiple versions simultaneously	13	8	Provider
Collaborate with other teams	15	9	Both
Actively involve consumer teams	14	8	Provider
Follow the API-first approach	11	8	Both
Internally, just break (and fix) it	11	10	Both
Abstract external systems' APIs	6	5	Consumer

unmanageable. Accordingly, dynamic monitoring approaches such as Helios (Popescu, 2010; Popescu et al., 2012) and D^2Abs (Cai and Thain, 2016; Cai and Fu, 2022) are used to identify or recover undocumented dependencies and potential change impacts between services.

4. API evolution strategies (RQ2)

This section presents the API evolution strategies that we found with the provider and consumer API evolution questions and the additional thoughts discussions of our interview guide. It answers RQ2: *Which strategies do developers follow to introduce and communicate API changes in loosely coupled systems?*

4.1. Answer to RQ2

The interview participants apply five strategies to evolve the provided microservice APIs and one to handle the evolution of consumed APIs. Table 3 contains the complete list formulated as comprehensive best practices that practitioners should follow when evolving microservice APIs.

First off, all participants must deal with breaking changes from adding or improving provided functionality and system maintenance efforts. All participants stay compatible with existing consumers and actively avoid introducing unexpected breaking changes. Many participants apply regression testing to detect unintentional breaking changes before release. Some participants implement dynamic APIs allowing custom queries where consumers decide the message fields in the response. All participants version their APIs and indicate breaking changes with increased version numbers. Many provide multiple API versions allowing consumers to migrate at their own pace. Most participants collaborate with dependent teams by discussing the planned API changes before implementation. They focus on the API definition before implementing the underlying functionality in parallel. Many participants agree that system-internal changes without impact on the public API do not require special handling, e.g., formal planning or versioning. From the consumer perspective, some participants promote an abstraction layer for external systems that handles authentication and message translations. In the following, we present the details of each strategy.

4.2. Accept necessary breaking changes

According to Lehman's laws of software evolution (Lehman, 1979), real-world software systems require maintenance and evolution to stay relevant. Consequently, all participants (17/17) must deal with breaking API changes from the provider perspective.

4.2.1. Understand the reasons for breaking changes

From the interviews, we identified four main reasons for breaking changes: (a) introducing new functionality (12/17), e.g., extending existing workflows or providing more diverse workflows and APIs, (b) changing the underlying technology (8/17), e.g., migrating to a new cloud provider or updating the programming language version, (c) improving existing functionality (6/17), e.g., merging similar workflows or improving performance, and (d) improving the API design (6/17), e.g., removing outdated workflows or restructuring the exposed API. Other reasons include bugfixing (4/17), introducing or changing security and authentication techniques (4/17), and migrating to a changed external system's API (4/17). We did not identify any notable differences between the participants' backgrounds and the mentioned breaking change reasons. Hence, we consider the top four reasons generally applicable. While the participants frequently introduce non-breaking changes, they only introduce breaking changes quarterly to half-yearly to provide enough lead time for affected consumers. Two exceptions are bugfixes and external API migrations, which require timely breaking changes to retain a stable system again.

Xavier et al. (2017) analyzed changes in Java library APIs and identified 28% as breaking. Brito et al. (2020) found three main motivations for breaking library API changes, which are similar to our findings: implementing new features, simplifying the API, and improving maintainability. Li et al. (2013) stated that more than 80% of web service API changes are refactorings, matching our findings for improving the existing functionality and API design.

4.2.2. Consider structural and behavioral changes

Our participants described two types of breaking changes: structural and behavioral. All participants (17/17) considered structural changes, e.g., deletions and renamings, breaking changes. Interestingly, only a few participants (5/17) identified behavioral changes, e.g., changing a timestamp's timezone or returning unexpected values, as harmful and handled them as potential breaking changes. *"The customer still recognizes that the values changed from last time and then triggers a support ticket to ask about it"* C3-P2. Some participants (6/17) intentionally introduce behavioral changes to avoid structural changes and use organizational strategies as justification, e.g., declaring all fields optional (cf. Section 4.3.3).

Newman (2021) called the two breaking API change types structural and semantic, and the service's internal behavior influences the semantics. Dig and Johnson (2006) considered behavioral changes in Java libraries breaking because changed computation results require different consumer-side handling. Similarly, Fokaefs and Stroulia (2014) classified web API changes as no-effect, adaptable, and non-recoverable, representing internal, structural, and behavioral changes, respectively. They justified their terminology because structural changes are recoverable by adding wrappers around consumer services, but behavioral changes require code changes and re-deployment to handle the changed computation results.

4.3. Stay compatible and avoid unexpected breaking changes

The main strategy from the provider perspective, followed by all participants (17/17), is to avoid unexpected breaking changes and to stay backward compatible with consumers. *"We ensure that we don't have any breaking changes, unless it's absolutely necessary"* C2-P2. The participants do not actively notify API consumers about backward compatible changes, except for the persons requesting the new functionality.

4.3.1. Work around breaking changes

Whenever possible, the participants (17/17) plan and implement workarounds to avoid breaking changes to their APIs. *"We would discuss this, what it means, what is affected by it, and then we try to find a solution that does not change something for the [existing consumers]"* C10-P1. Many participants (11/17), especially the ones with smaller system sizes, recommend extending an API by adding or duplicating endpoints, messages, and fields instead of changing them to ensure compatibility of the new functionality with existing consumers. Similar to the individual APIs, the overall system's business workflow should not break after introducing changes. The execution order and results for each step in a workflow should stay compatible with all existing consumers. This is especially important for long-running workflows, e.g., ordering, preparing, and shipping a product, where breaking workflow changes would disrupt any open orders. However, one participant warns of introducing tailored APIs for single use cases to avoid breaking changes in others. In their experience, this fragments the APIs and increases the system's complexity.

Theoretically, this compatibility strategy provides the best results for providers, who only maintain a single API version, and consumers, who do not need to change their implementation. In practice, avoiding breaking API changes is not always possible (Xavier et al., 2017; Brito et al., 2020). Daigneau (2012) advocated the *Tolerant Reader* pattern for consumers, e.g., only accessing needed message fields, not relying on sorting but identifiers, and wrapping domain-specific objects in more general structures such as lists and maps, to reduce consumers' susceptibility to breaking changes.

4.3.2. Regression test the API

Many participants (10/17), including all four senior developers, recommend regression testing to detect accidentally introduced breaking changes before re-deploying their services. *"You already have a set of test cases that you can run against the old interface. You will see immediately when you introduce something that breaks it"* C2-P3. *"Before releasing, we run each and every test we have - and this is a quite huge test suite - over the last release version again to make sure nothing did break in between, since the last release"* C5-P2. We found that most participants following MSA and all participants with a system size of 30 components or more apply this strategy. We presume that the loose coupling and increased system complexity require such automated testing approaches to identify inestimable breaking changes. Unit testing of the source code detects behavioral changes, e.g., changed result values, and contract tests detect structural and behavioral changes in the API, e.g., required parameters or unexpected response objects. Finally, a few participants (3/17) execute complete end-to-end tests to ensure functionality and backward compatibility for the most important workflows. When introducing planned breaking changes, the loose coupling of services requires the developers to adapt their contract test doubles to the changes manually. *"At the point of writing, my test data has the format that you would say it will have. If you change it, then I will have to change my test data as well"* C6-P1.

Regression testing is an established practice to raise confidence that program modifications have no unexpected adverse effects (Leung and White, 1989; Wong et al., 1997). Biswas et al. (2011) concluded that regression testing component-based systems helps to detect indirectly modified APIs after behavioral changes in the business logic. While only a few participants stated the types of tests they employed, we expect them to use the functional tests for MSA recommended by Richardson (2018): unit tests, integration or contract tests, component tests, and end-to-end tests. Chen et al. (2021) identified test case generation as an open concern in grey literature. Godefroid et al. (2020) proposed an approach for regression testing structural and behavioral REST API changes by automatically generating requests and comparing the responses for multiple service and consumer version combinations.

Frameworks, such as Spring Cloud Contract[10] for Java and pact-net[11] for .NET, simplify REST API testing. They generate server stubs for consumer-side testing and define a domain-specific language to write requests for server-side tests. Demircioğlu and Kalipsiz (2022) proposed regression testing for message-driven APIs. They extracted low-level TCP and UDP package payloads and reverse-engineered the request and response messages into future regression test cases.

4.3.3. Think ahead and design a dynamic API

Some participants (6/17) recommend that providers design dynamic APIs with the goal of flexibility, resulting in fewer breaking changes. Dynamic APIs publish all available fields of a response object, and consumers pre-filter them as part of the request. The consumers then only receive their subset of fields, potentially containing null values. The API providers must plan ahead and consider current and future use cases and their expected responses to allow such dynamic APIs. *"Therefore, in general, we [...] think ahead and we try to add many times also attributes in advance"* C9-P1. With this approach, the developers design a clear, extensible, multi-purpose API for the underlying functionality instead of a specific API tailored to the current use cases. The participants recommend JSON objects compared to strings or binary because JSON allows for hierarchies, lists, and null values. We found that none of the participants following this strategy mentioned improving existing functionality as a reason for breaking changes. Instead, such changes extend the dynamic API or its filter options and response objects may contain additional null values. A few participants (3/17) declare most fields optional to avoid future breaking changes. Notably, all advocates of dynamic APIs also apply the regression testing strategy. We attribute this to the increased size and complexity of dynamic APIs to support querying and filtering. Accordingly, the evolution of dynamic APIs creates more maintenance overhead than simpler APIs. Surprisingly, only one participant mentioned GraphQL, a query language specialized in querying a subset of response fields. The others implemented the dynamic APIs with REST.

Bloch (2006) advised self-explanatory and extensible APIs. They should not overconstrain but serve multiple use cases. Consumers adhering to the *Tolerant Reader* pattern (Daigneau, 2012) can react to changes in list sizes, hierarchical structures, and null values of APIs gracefully and might even recover from moved fields. Brito and Valente (2020) showed that GraphQL queries required less implementation time than REST, with improved results for increased query complexity. Brito et al. (2019) found that client-specific GraphQL queries allowed a reduction of JSON response fields by 94% compared to REST. Wittern et al. (2019) revealed exponential response times and sizes for GraphQL queries in practice and recommended throttling requests and pagination techniques. Similarly, Quiña-Mera et al. (2023) concluded that GraphQL requires more best practice examples and improvements in query complexity, code generation, and security. These findings might explain our study results, where only one participant used GraphQL while others preferred custom REST implementations.

4.4. Version the API

All participants (17/17) apply versioning as providers to evolve their APIs and consumers use the version information in requests and messages to access the corresponding API version of a service. *"We have defined that for every API that is accessible from the outside we do have versioning"* C3-P1. Notably, the participants focused on REST APIs when discussing versioning and we identified versioning for event-driven communication as a challenge (cf. Section 5.7).

---

[10] https://spring.io/projects/spring-cloud-contract.
[11] https://github.com/pact-foundation/pact-net.

4.4.1. Create a new version on breaking changes

All participants (17/17) increase the API version when introducing breaking changes as providers. Non-breaking changes, such as exposing new endpoints or extending message objects, are implemented in the latest API version directly. The participants use increasing integer values for enumerating the API versions, e.g., $v1$, $v2$. A few participants (5/17) explicitly mentioned semantic versioning,[12] all of whom hold architectural roles. We assume they are used to understanding and emphasizing the exact versions during discussions. Still, only the major version number in semantic versioning indicates breaking API changes and is relevant for consumers. Hence, the requests to a versioned API only require the major version number to indicate compatibility, independently of the internal versioning granularity.

Semantic versioning or integer versioning are well-known strategies for indicating breaking changes in API management (Koçi et al., 2019; Knoche and Hasselbring, 2021). Neumann et al. (2021) analyzed 500 REST APIs and found that 65.4% exposed the major version within the request call. Similarly, Serbout and Pautasso (2023) analyzed 7,114 REST APIs, and the majority used static versioning in the URI or request metadata (70.1%) or dynamic version discovery through a dedicated endpoint (3.1%). Taibi and Lenarduzzi (2018) considered not having API versioning as an MSA smell.

4.4.2. Expose multiple versions simultaneously

Ideally, a new API version supersedes the previous version, and provider teams only maintain the latest version as a single source of truth. In reality, many participants (13/17) expose multiple API versions simultaneously to serve consumers who do not or only infrequently update their API calls. The newest features are only available in the latest API version and all consumers requiring these features must update their calls. Other consumers are unaffected and continue using the previous API versions. The participants follow two approaches for running multiple API versions in parallel: exposing all API versions in the same service instance (8/17), and deploying each service version separately (5/17). Most architects (6/7) prefer exposing all API versions within the same service. They consider it easier to maintain because the underlying business logic stays consistent. *"So just this simple mapping of DTOs. When you use the right technologies it's quite OK and not that much of effort"* C2-P1. In contrast, deploying each service and API version separately duplicates the source code base and requires more complex message routing. Still, some participants (5/17) prefer the smaller service instances and freedom in changing the source code per version, including both participants working at small companies. We did not discover notable differences for the participants' approaches with respect to their system architecture or size. The number of simultaneously exposed API versions typically ranges from 2 to 8. *"We have to keep the last three versions running, not more"* C8-P1. *"For core systems we have about 7 to 8 breaking versions"* C2-P1. However, the participants only remove old APIs once all consumers migrated to the newer version. After all, they must support all customers independently of the used request versions. This sometimes requires the participants to support old API versions indefinitely, especially for important and slowly responding customers (cf. Section 5.3).

Newman (2021) proposed the same two strategies we found for running multiple API versions in parallel: *emulating the old interface*, i.e., exposing all API versions in the same service instance, and *co-existing incompatible microservice versions*, i.e., deploying each service version separately. Like our participants, he recommended emulating the old interface because this approach is easier to maintain, evolve, and monitor. Neumann et al. (2021) reported that about two-thirds of the 500 analyzed REST APIs supported API version selection, indicating multiple active versions. The *Parallel Change* pattern (Sato, 2014) requires both the old and new versions running for the consumers to

migrate at their own pace. Providers remove the old version once the consumers finished the migration. Wang et al. (2014) found that web APIs follow such deprecate-replace-remove cycles in practice. Serbout and Pautasso (2023) encountered 135 out of 7,114 REST APIs with multiple active versions and a maximum number of 14 coexisting versions. We explain the low number of 135 compared to our qualitative result with their automated extraction approach. They extracted the version information from the OpenAPI specifications, where we expect providers to motivate consumers to use only the latest version (cf. Section 5.5.1).

4.5. Collaborate with other teams

Most participants (14/17) closely collaborate with teams of consumer services during the API evolution process by providing change previews, receiving early feedback, and synchronizing integration. While they are finally responsible for the evolution of their APIs, they value consumers' feedback. *"So, if other product teams, for example, are affected by this [change], then we first have some discussion rounds about it"* C5-P2.

4.5.1. Actively involve consumer teams

Most participants (14/17) discuss planned changes in provided APIs with consumer teams and use the feedback to improve the underlying workflow and API design before release. Participants with more experience in loosely coupled systems were more likely to apply this proactive collaboration strategy. Many participants (11/17) schedule meetings for the discussions. According to them, one or two people per dependent system are involved in such meetings, which take up to one hour. A few participants (3/17) distribute API version previews for asynchronous feedback loops. The architects, including both principal architects, prefer meetings, while three of the four senior developers prefer the API previews for asynchronous feedback. We presume that senior developers want to take their time to fully understand the specification, whereas architectural meetings address the API use more broadly.

Once all involved teams accept the proposed API design, they implement the services and consumers in parallel and add them to the testing environment as soon as possible for additional feedback. Encountering problems with the agreed-on specification during the implementation phase triggers additional follow-up discussions. Finally, the developers of multiple teams jointly write contract tests and plan the deployment of the individual components. *"But we also make the meetings to describe the changes and make tests together on the QA systems and define a date where they switch over to the new interface. And we monitor if it works for them"* C3-P1.

Richardson (2018) acknowledged that features spanning multiple services require careful coordination between development teams. Bogart et al. (2021) found that developers considered breaking changes the provider's responsibility, who felt personally obligated to help resolve them. This explains our participants' close and proactive collaboration approach. The collaboration results from the loosely coupled system architecture, which, by itself, does not provide any immediate feedback on the implementation's and integration's correctness (Pautasso and Wilde, 2009).

4.5.2. Follow the API-first approach

To simplify the collaboration efforts, many participants (11/17) discuss and agree on an API definition before starting the implementation. This API-first approach continuously improves the API design based on encountered design issues and consumer feedback without having to implement the actual logic behind the interfaces. *"It's an iterative process. There is no shame in having a final-v2"* C6-P1. The main goal is to create a well-defined API, not a functional prototype. According to the participants, the API-first approach improves the overall design by focusing on readable, self-documenting, and reusable APIs. *"Both*

---

[12] https://semver.org.

the [internal developers] and the customer using the public API have a nice experience and get all the same information" C7-P1. Furthermore, changing a preliminary API definition requires less effort than changing a partially implemented system. Most participants following MSA apply the API-first approach (7/8). We expect that the small microservice size facilitates the creation of a complete API specification, which, in turn, allows stubbing the actual functionality during development.

The participants use the OpenAPI specification (cf. Section 3.2.3) to document and distribute the REST API definition when following the API-first approach. All participants applying the API-first strategy also actively involve consumer teams (cf. Section 4.5.1), except for one (10/11). We presume that the formalized OpenAPI specification simplifies such collaborations. Some participants (5/17) further use the OpenAPI specification to automatically generate server code, consumer code, contract tests, and client SDKs. Interestingly, this was mentioned by most senior developers (3/4) and two architects, but not by other developers who typically work at this design level.

Kopecký et al. (2014) described the API-first approach as first building the functionality as API and only then creating clients for that API. Hence, developers design APIs to provide their business functionality to the outside, not only to support specific use cases (Wilde and Amundsen, 2019). Beaulieu et al. (2022) concluded that the API-first approach creates clear and well-defined APIs that expose business capabilities, reduce the domain coupling with consumers, and allow parallel development.

4.6. Internally, just break (and fix) it

Many participants (11/17) agree that internal breaking changes are easier to implement and integrate from a provider perspective and, hence, occur more frequently. *Internally* refers to the accessibility scope of the breaking API, i.e., the affected consumers are company-internal, well-known, or their source code is directly accessible. In this case, many developers (9/17) introducing breaking changes also change all the consumers and the test suites. Some developers (6/17) are in close contact with the colleagues maintaining the consumers, or directly create pull requests for the consumers' source code. Accordingly, a few participants (4/17) explicitly stated they do not version internal APIs, but update, test, and redeploy them directly. We could not identify notable differences in the answers for this strategy regarding the participants' backgrounds and assume that it is generally applicable.

However, this evolution strategy for internal APIs is unrelated to the MSA. Hence, we refer to conventional source code evolution research, e.g., Brito et al. (2020).

4.7. Abstract external systems' APIs

From a consumer perspective, some participants (6/17) use dedicated *integration services* to abstract communication with external systems. These services handle the authentication with the external systems and translate request and response field names to the internal domain names. The internal services then do not know about the actual external systems they communicate with. This allows the integration services to partially handle breaking changes in external systems, e.g., changed authentication, moved or renamed fields, and some semantic changes, and convert them back to the expected values. Hence, they minimize error propagation, and an external API change might not affect other internal services. Notably, only participants working in large companies mentioned this consumer strategy.

The integration service is the consumer-side counterpart of the API gateway (cf. Section 3.2.2). This abstraction layer follows the *Proxy* and *Facade* patterns (Gamma et al., 1995) on the component level, e.g., implementing access functionality and simplifying the external interfaces. Espinha et al. (2015) conducted six interviews where the developers advised to contain external web API changes to a small set of files. Similarly to our participants, Fokaefs and Stroulia (2014) considered structural changes recoverable by adding a wrapper to the original consumer service. Similarly, Wu et al. (2016) recommended encapsulating external libraries to reduce the potential change impact.

Table 4
The API evolution challenges with participant and company counts.

API evolution challenge	# P	# C	Perspective
Manual change impact analysis is error-prone	14	11	Both
Code changes affect the API unexpectedly	9	7	Provider
Understanding consumed APIs' changes is effort	9	7	Consumer
Consumers rely on API compatibility	12	7	Provider
Communication with other teams lacks clarity	9	7	Both
Consumers might be unknown	7	5	Provider
Informal communication channels	17	11	Both
Communication suffers from hierarchy	6	4	Both
API maintainability and usability degrade over time	14	9	Provider
Outdated API versions add maintenance overhead	10	8	Provider
Backward compatibility increases technical debt	9	6	Provider
Governmental services are uncooperative	6	4	Consumer
Event-driven communication evolution is disregarded	7	4	Both

5. API evolution challenges (RQ3)

This section presents the API evolution challenges that the participants encountered. We elicited them with the provider and consumer API evolution questions and the additional thoughts question of our interview guide. This section answers RQ3: *Which challenges do developers face when introducing and communicating API changes in loosely coupled systems?*

5.1. Answer to RQ3

We identified six challenges in the API evolution process, out of which three result in degrading API maintainability and usability. Table 4 contains the complete list formulated as comprehensive pitfalls for practitioners.

First, most participants encounter problems in understanding the impact of source code changes on their own APIs as providers and the impact of external API changes on their services as consumers. Second, consumers of many participants fully rely on API compatibility and refrain from migrating to a new version. Third, many participants consider communicating with other teams challenging, especially company-externally. They follow no general communication strategy and suffer from hierarchical communication. As a result of these challenges, the API cannot evolve sustainably, and most participants report degrading API design and increasing technical debt. In contrast, governmental services choose to evolve APIs regardless of consumer concerns, which poses a challenge for some participants consuming them. Finally, we noticed that participants hesitated to discuss event-driven communication, and some deemed evolving event-driven communication challenging. In the following, we present the details of each challenge.

5.2. Manual change impact analysis is error-prone

Most participants (14/17) find assessing source code and API change impact challenging. Based on the provider and consumer perspectives, we split this challenge into two: understanding the impact of source code changes on the provided APIs and understanding the impact of external API changes on the consuming services.

5.2.1. Code changes affect the API unexpectedly

Many participants (9/17) state that development teams must manually assess the impact of source code changes on the APIs they provide. They experienced that the developers sometimes overlooked that they introduced breaking changes to the API and published them without versioning or notifications. *"From time to time we face problems, but mainly because some team has overlooked that it has been doing a breaking change"* C2-P1. We found that most participants dealing with

this challenge recommend the collaboration strategy (cf. Section 4.5), which helps to resolve any breaking changes that are only identified by consumers (8/9).

Moreover, technology changes, such as library or language version updates, may result in unexpected side effects for the API that are not visible in the code base. We found that all participants mentioning changing the underlying technology as a reason for breaking changes (cf. Section 4.2.1) encountered this challenge, except for one (7/8). As a solution, one participant uses git diff to extract the changes between two generated OpenAPI specification versions manually to identify overlooked structural breaking changes. However, developers lack the tools to identify behavioral changes and, hence, a few participants (4/17) consider them especially challenging. Static analysis tools have problems detecting behavioral changes, e.g., changes in the return values of methods, and automated tests cannot cover all execution paths. *"When you go into this [...] topic, it's not so easy to test all constellations"* C10-P1.

Static analysis tools, e.g., openapi-diff,[13] extract structural changes between two OpenAPI specification versions. However, Rubin and Rinard (2016) conducted 35 interviews with software developers and reported that integration challenges are mainly related to semantic and behavioral changes introducing unpredicted side effects. Sorgalla et al. (2018) proposed model-driven microservice development. They assessed the impact of model changes by assembling the system to execute integration tests and marked the conflicting microservices. Ma et al. (2019) automatically prioritized contract and unit tests based on the service dependencies to identify and prevent unexpected breaking changes in the MSA faster. However, these approaches require access to the whole system's source code and sufficient test coverage. Chaturvedi and Binkley (2021) applied web service slicing by identifying changed WSDL operations based on the source code's behavioral changes and used the slice for regression test selection. Static analysis approaches for the source code level (Hanam et al., 2019; Nguyen et al., 2019) used control and data flow analysis techniques to extract semantic changes and their impact without relying on test execution. However, they do not apply to the service API level.

5.2.2. Understanding consumed APIs' changes is effort

Many participants (9/17) encountered problems with analyzing external API changes from a consumer perspective. We found that the participants do not follow a generalizable strategy when filtering external change notifications for relevancy or assessing the change impacts, except that they do it manually. Sometimes (6/17), the development teams are the ones who assess the impact of external changes directly. The developers identify the dependencies and relevant service changes by reading the change notifications, external documentation, and their own source code. Participants belonging to larger teams especially mentioned this time-consuming approach, including all three with a team size above ten. Understanding the external changes strongly depends on the notification and documentation quality. *"Because if it just says: API extension, there's a new field in there, you think, yeah, for what?"* C12-P1. If the notifications contain the thoughts and reasons for the API evolution it is easier to identify, understand, and integrate relevant changes. *"The only challenge then is really to find any edge cases that are not described in the documentation, and which then will cause errors in our system"* C11-P1. A few participants (4/17) acknowledged that provider teams pre-filter the notifications of breaking changes for consumer teams. While this approach reduces unnecessary communication, misjudgments result in system failures. *"They just missed out on one change, because they didn't know that it was important for us"* C3-P1. A few participants (2/17) rely on dedicated roles, e.g., product owners or architects, who know the service dependencies and actively inform

the teams of external changes. As a downside, this approach relies on individual peoples' knowledge.

Xavier et al. (2017) found that, once known, breaking Java API changes only impacted less than 3% of their consumers. Bogart et al. (2021) discovered that most participants felt overwhelmed by the number of change notifications and considered integrating them risky. Dynamic analysis approaches for loosely coupled systems (Liu et al., 2019; Guo et al., 2020) trace service calls at runtime to generate a service dependency graph (SDG) and analyze behavioral changes and performance issues over time. Similarly, Helios (Popescu, 2010; Popescu et al., 2012) and D^2Abs (Cai and Thain, 2016; Cai and Fu, 2022) identified dependencies and potential change impacts of services at runtime by analyzing event-driven message handling. As a limitation, these approaches require access to the system's runtime environment, which is not available to external consumers. Furthermore, they cannot identify the actual impact of changes but only the potentially affected services and methods.

5.3. Consumers rely on API compatibility

Many participants (12/17) report the challenge of convincing consumers to update their API calls to the new version after introducing breaking changes. As providers, the participants prefer removing outdated API versions and only focusing on the latest. Still, consumers rely on previous API versions even after receiving requests to update their calls within some timespan and, hence, hinder the clean-up process. *"And then, if they changed, we can remove the old version finally. But that's always a bit more work because you have to keep the old version compatible"* C3-P1. We found that both participants working in small companies did not report this challenge. Further, three participants from only two large companies did not encounter the challenge, with one of them not supporting any external consumers. Hence, we expect this challenge to be generally applicable to larger companies that typically support more consumers. We found two main reasons for the consumers' reluctance: the teams do not have enough resources to update the API calls in the near future (9/17), or do not prioritize changes to already working functionality (7/17). *"Most of the customers don't touch the code anymore for one year or 1.5 years if it works"* C5-P2. Most participants suffering from this challenge apply the collaboration strategy (cf. Section 4.5) (11/12). We presume that they attempt to convince consumers to update their calls by discussing the planned changes in advance and actively supporting the migration steps.

Some participants (5/17) explicitly stated they follow the *never change a running system* strategy as consumers themselves and only migrate API calls if they require the new functionality. *"What for, I don't need anything from 2.0 to 4.0, my world is running"* C12-P1. This approach was only mentioned by developers and senior developers, possibly due to their direct exposure to developmental strain. While it reduces the development effort from a consumer perspective, the same teams suffer from the slow and rigid migration strategy when their consumers apply it. Eventually, some participants (5/17) force their consumers to migrate their calls by turning off the outdated API version with a fixed, non-negotiable deadline, but this measure is not feasible for business-critical APIs. *"Yeah, we're earning money with them so you cannot just say: sorry you cannot use it anymore"* C2-P3.

In the early years, web services introduced breaking API changes without versioning or with short deprecation periods, e.g., of three months (Li et al., 2013; Fokaefs and Stroulia, 2014; Wang et al., 2014; Espinha et al., 2015). Espinha et al. (2015) found that developers preferred longer deprecation periods after conducting six interviews. Neumann et al. (2021) reported that two-thirds of 500 analyzed REST APIs supported version selection, indicating that breaking changes in newer versions did not immediately affect old consumers. Hora et al. (2018) studied the impact of library API changes. While more than half of the investigated systems were potentially affected by changes, the majority did not react and continued using the previous version. To prevent this behavior, de Toledo et al. (2021) suggested a clear period of support that should not be extended. However, this is often not feasible in practice.

---

[13] https://github.com/OpenAPITools/openapi-diff.

5.4. Communication with other teams lacks clarity

Many participants (9/17) encountered problems in communicating changes with other teams. Developers do not know whom to inform, forget to notify the consumer teams, or convey the change information incorrectly. *"If something goes wrong, it's communication"* C3-P3.

5.4.1. Consumers might be unknown

The provider teams introducing breaking changes should know which consumers are affected and how to contact the corresponding teams, but some participants (7/17) miss knowledge or appropriate documentation of consuming services and contact partners. Most participants encountering this challenge reported that their consumers rely on API compatibility (cf. Section 5.3) (6/7). After all, the participants cannot reach out to notify them about breaking changes or request migrations. The challenge is more prominent in larger teams, where the knowledge may be more distributed. Interestingly, all participants encountering this challenge also face difficulties in understanding the impact of external API changes (cf. Section 5.2.2). If they are not informed about breaking changes as consumers, the services exhibit unexpected behavior or failures after the update and require manual investigation of the problem. *"We got a 503 - Service Unavailable and so I called the product and asked: what's the problem here? And then they told me: Oh, right, we changed the API for that"* C3-P2.

We could not find a generalizable consumer documentation strategy to recommend during our study. We noticed that most architects (6/7) seem to know their consumers and did not mention this challenge. Arguably, these roles might have a different understanding of *consumers*, e.g., on a department or company level, whereas developers require individual contact persons whom they can reach out to. Accordingly, some participants (8/17) rely on their teams' or superiors' implicit knowledge. They discuss future API changes and potentially affected consumers with specific colleagues, team leads, or architects. *"He just remembers most of the time. Or maybe he has some documentation on his end. I'm not really 100% sure"* C3-P3. As a downside of this implicit knowledge, the information becomes siloed and can get lost if the individual leaves the company. Some participants (6/17) log REST API calls from consumers with tracing tools, e.g., Dynatrace,[14] Grafana,[15] or custom implementations. This allows them to look up all consumer IPs or hostnames for each REST API endpoint and provides information about its use. However, these participants still face the challenge of unknown consumers as they cannot infer actual contact persons from the logged calls. A few participants (4/17) utilize the credentials used for authenticating calls to their services to maintain a list of actively used APIs and their corresponding consumers. A few participants (3/17) even maintain manual documentation about the consumers and contact partners for each microservice. *"We document it in lists and I think this is not really ideal. So, which API is used by which. This is especially error-prone if we have to change something"* C8-P1.

de Toledo et al. (2021) recommended tracking internal and external users to directly request migrations. Consequently, related works build SDGs by analyzing the source code or runtime behavior of services (Bushong et al., 2021). Laverdière et al. (2015) proposed static analysis to construct a cross-service call graph by analyzing calls between SOAP services. Similarly, Ma et al. (2019) visualized and analyzed SDGs by statically extracting REST API calls from the source code. Cai and Thain (2016) proposed identifying method dependencies based on the execution order of event-driven messages at runtime, and we presented additional dynamic approaches in Section 5.2.2. As a limitation, these approaches require access to the source code or runtime environment of the services, which is not available for external partners.

---

[14] https://www.dynatrace.com.
[15] https://grafana.com.

5.4.2. Informal communication channels

During our study, we could not identify a generalizable strategy to inform consumer teams about API changes. The participants either follow their own ad hoc strategy or accept the overhead for manual communication. *"Sadly, there is no company standard for [communicating] versioning interfaces. It's up to the products to handle this"* C3-P1.

Company-internal, the main means for written communication are e-mails (9/17), followed by announcement channels (6/17) and instant messages (4/17), e.g., via Slack, Microsoft Teams, or Mattermost. *"And we do have people that actively have to look at these propagated changes. Are they relevant for the services I'm responsible for?"* C6-P1. Notably, e-mail communication is preferred within larger teams. Alternatively, API changes are verbally announced in formal meetings (5/17), e.g., co-ordination or sprint review meetings, and informal meetings (5/17), e.g., coffee talks. As a downside of such verbal communication, developers might miss the API change notifications or forget the details over time. Accordingly, all five participants who conduct formal meetings also maintain manual documentation for their APIs (cf. Section 3.2.4).

Company-external, some participants (6/17) use e-mails to communicate with partners and customers. Some participants (6/17) also mentioned dedicated roles responsible for communicating and managing the API changes. This role could belong to the product owner, a dedicated coordinator position, or even a dedicated team centrally managing the company's API integration. As a downside, this again constrains the knowledge to one or a few persons. A few participants (4/17) notify breaking and non-breaking API changes via release notes but simultaneously consider the natural language description too verbose for a technical assessment. *"I'm pretty sure that no customer is really looking at that"* C5-P1. A few participants (4/17) have to actively check for breaking changes as API consumers, especially for large API providers like Amazon Web Services. *"Finally, we need to ask all the time for changes or if they are changed and then there is a quite big delay until we can continue"* C9-P1. We could not identify any relationships between the participants' backgrounds and the employed communication channels. Hence, we conclude that no generally applicable internal or external communication strategy exists.

Espinha et al. (2015) identified e-mails as the main communication channel but found developers considered them unreliable. Additionally, large providers, e.g., Google and Twitter, sent upcoming changes via e-mail lists of registered accounts. Similarly, Bogart et al. (2021) reported that developers communicated pre-release announcements via e-mail and Twitter. Sohan et al. (2015) identified four communication channels: the API homepage, the API response, e.g., deprecation information in the header, customized e-mails, and newsfeeds. Yasmin et al. (2020) found only three out of 1,368 analyzed REST APIs proactively informed callers about deprecation in the response objects, where developers would directly see it during development or in log files. However, consumers who are no longer actively working on APIs they previously integrated (cf. Section 5.3) may not notice such deprecation information.

5.4.3. Communication suffers from hierarchy

Some participants (6/17) suffer from a high level of organizational abstraction, hindering effective communication. Communication with unfamiliar teams or external partners involves multiple developers, team leads, and company representatives, possibly altering the information with every pass down the chain. The involved people might overlook the details of API changes or mistakenly consider them unimportant. *"The problems arise when too many third parties are involved because it's like the telephone game. You pretty much get completely different results at the end"* C3-P3. From the consumer perspective, this reduces trust in the communicated changes. Furthermore, we discovered that most participants facing this challenge encountered difficulties in understanding the changes of consumed APIs (cf. Section 5.2.2) (5/6). Consequently, from the provider perspective, most participants with

this communication challenge observe that their consumers rely on previous API versions (cf. Section 5.3) (5/6).

Baškarada et al. (2020) conducted interviews with 19 software architects and reported API change communication and coordination with related services as challenging. Rubin and Rinard (2016) found that collaboration and software quality depended on the social boundaries of developing companies. We refer to Conway's law (Conway, 1968) stating that a system's structure follows the companies' communication structure.

5.5. API maintainability and usability degrade over time

As a result of the previous challenges, many participants (14/17) experience degrading API and source code quality in the services they provide. Consumers relying on a specific version and uncertainty about introducing breaking changes force providers to maintain the outdated versions and increase the technical debt with no clear resolution strategy.

5.5.1. Outdated API versions add maintenance overhead

Many participants (10/17) mentioned the overhead of maintaining old API versions to ensure backward compatibility with existing consumers. We discovered that most architects (6/7) and senior developers (3/4) acknowledge this challenge. *"That's a huge pain for us"* C5-P2. This challenge aggravates if consumers rely on previous API versions (cf. Section 5.3), with most participants facing this challenge also facing the other (8/10). Furthermore, we observed that the challenge becomes more prominent with larger system sizes. Some participants (5/17) consider the additional routing and handling logic based on the respective message version as an overhead. This logic converts the REST API requests and event-driven messages to the newest format or forwards them to the corresponding workflow version. When running coexisting incompatible microservice versions (cf. Section 4.4.2), the system requires a dedicated routing layer because the incoming requests and messages are processed by individual runtime components. A few participants (4/17) highlighted the increased code base size and complexity for the backward-compatible business logic and workflows, additional tests to verify each supported version and regression test any changes, and even for backported features, further complicating outdated workflows instead of removing them. When running coexisting incompatible microservice versions, the providers must maintain multiple code bases, one for each supported version, and synchronize them accordingly.

Some participants (5/17) feel that the overhead for ensuring backward compatibility interferes with developing new features. We found that these participants belong to small teams, where individual team members are likely to perceive the regular maintenance overhead more prominently. *"It holds you back if you want to change some other implementation, if you want to optimize something, or implement some new feature that doesn't work with an old way of transferring data or something like that. And it also slows you down or holds you back from developing any new features"* C2-P2. Finally, we found that most participants who provide dynamic APIs (cf. Section 4.3.3) face the maintenance overhead challenge (5/6). We presume that the inherent complexity of dynamic APIs further increases their maintenance overhead.

Bogart et al. (2021) called the overhead to maintain obsolete code and create workarounds for compatibility *opportunity cost*. This opportunity cost transforms into consumers' migration cost once the providers decide to break and clean up the interface. Espinha et al. (2015) recommended providers deprecate and remove the outdated APIs at some point to avoid increasing opportunity costs. Similarly, Lübke et al. (2019) proposed three deprecation patterns: eternal lifetime guarantee, limited lifetime guarantee, and aggressive obsolescence. The first pattern provides unlimited API support, the second provides a clear deadline as part of the API version release, and the third removes an outdated version with prior notice of a deadline, which is not necessarily known during release. The three patterns balance the forces of opportunity cost and consumer efforts. However, an API deadline is often not feasible in practice (cf. Section 5.3).

5.5.2. Backward compatibility increases technical debt

Many participants (9/17) experienced that, as a provider, avoiding breaking changes and favoring extensions for backward compatibility degrades the initial API design over time. *"That means we need to carry all the technical debt in our SDKs and our public APIs"* C5-P1. This challenge is more prominent for participants following a service architecture, where components are expected to have larger individual code bases. Eventually, the evolved APIs contain multiple workflows for the same functionality, outdated fields filled by old consumers but ignored when received, optional fields only processed by some consumers, and multiple equivalent endpoints fixing typos or supporting different languages. *"I mean, there's a developer perspective. You want to get rid of old, not really good working stuff, but in reality you just can't"* C5-P2. Almost half the participants following the duplication strategy to work around breaking changes (cf. Section 4.3.1) explicitly stated that this exacerbates the technical debt challenge by convoluting the API (5/11).

The technical debt increases implementation complexity for new functionality and regression testing efforts for identifying unexpected side effects. Similarly, the API usability degrades as the consumers try to understand the differences between duplicated workflows, requests, and fields. A few participants (3/17) created a new streamlined version once their APIs became too convoluted and tried to convince their consumers to move to this cleaned-up version. In the worst case, this improvement step creates yet another version to maintain. *"Of course, you can also deprecate it. The question is if the other colleagues will also take it seriously"* C11-P1. Accordingly, most participants dealing with unknown consumers (cf. Section 5.4.1) mentioned increased technical debt in their APIs (6/7) because they are unable to communicate the breaking changes resulting from the clean-up. Interestingly, we discovered that announcing breaking changes via face-to-face meetings (cf. Section 5.4.2) helps to mitigate the technical debt challenge, as most participants using this internal communication channel did not experience it (7/8). We presume that such direct meetings create a sense of responsibility with API consumers. However, we could not identify an external communication channel to mitigate the challenge.

de Toledo et al. (2021) identified poor REST API design as technical debt. It results in API instability, regular breaking changes, and increased difficulty in maintaining backward compatibility with newer versions, which is similar to our study's findings. Bogart et al. (2021) identified technical debt as a major driver for breaking changes from developer interviews. At some point, developers had to break the interface to introduce a clean version. Research on API maintainability and usability recommended following API standards, providing clear deprecation messages, and providing up-to-date documentation and usage examples (Lamothe et al., 2021).

5.6. Governmental service providers are uncooperative

Some participants (6/17) encountered problems as consumers of governmental services. These participants discussed ministries of governments in multiple European countries. Some participants (5/17) criticized that governments do not provide a direct line of communication and contact partners are hardly available. Instead, they introduce breaking changes on short notice or do not notify consumers in advance at all. *"Sometimes they don't do it, they just change their service. [I found out] when the application crashed"* C1-P1. Some participants (5/17) experienced an unwillingness to cooperate. Governmental services shut down with a fixed date, and consumer requests were disregarded. Furthermore, they regularly changed agreed-upon API specifications during development, and errors in the API were not investigated until consumers sent an example proving their claim. *"You have to prove to them that they are wrong because they always say that you are doing something wrong"* C3-P1. Consequently, the participants face challenges in understanding the governments' API changes as consumers (cf. Section 5.2.2). To reduce the impact of such changes, many participants

who consume governmental services abstract these external systems (cf. Section 4.7) (4/6).

We explain the uncooperative behavior with governments providing their services as a courtesy instead of a paid product with an underlying service agreement. *"If the ministry offers an API where you can upload your tax data, you don't pay for it. They offer it"* C12-P1. Hence, they introduce breaking changes with aggressive obsolescence (Lübke et al., 2019), prioritizing the provider's opportunity costs over the consumers' regular migration costs. This freedom allows governments to avoid most of the challenges we identified.

5.7. Event-driven communication evolution is disregarded

Finally, we discovered that participants refrained from discussing evolution strategies for event-driven communication. More than half of the participants using this communication technique (7/13) explicitly mentioned the challenge of versioning event-driven communication via message-oriented middleware, e.g., message queues and publish–subscribe. Especially, the architects (6/7) including both principal architects discussed this issue. They highlighted that protocols do not support versioning natively, and the lightweight frameworks do not implement versioning out of the box. Creating new queues for each version or utilizing message fields to store the version tags requires more manual intervention than versioning of REST APIs, where the frameworks automatically handle the version information in the URI or message header. Moreover, the asynchronous nature of event-driven communication requires subscribers to accept old message versions even after all producers migrated, because old messages might still wait in the queue for later processing. This prolonged backward compatibility requirement further degrades the API design (cf. Section 5.5). Consequently, the participants either migrate all producers and subscribers simultaneously and accept potential message loss (3/13), or start implementing their own version negotiation protocol once the message volume becomes too large or external components are involved (5/13). *"So, we basically have a small protocol for this version negotiation to ensure that the systems can talk to each other"* C2-P2. Interestingly, only one participant mentioned Apache Avro[16] to help with message serialization and versioning.

Baškarada et al. (2020) discovered that very few practitioners had experience with event-based architectures after conducting 19 interviews. Also, de Toledo et al. (2021) discovered developers preferred complex REST API calls over event-driven messaging and classified it as an inadequate use of APIs. This supports our finding that developers are less confident with event-driven communication and its evolution. Knoche and Hasselbring (2021) used Apache Thrift[17] and Apache Avro to define a custom description language for REST API message versioning and translation. This approach targets only the message structure and, hence, could be adapted for messages of event-driven communication in the future.

6. Tight organizational coupling and consumer lock-in

Based on our qualitative findings, we constructed an overall theory of two important underlying problems in microservice API evolution, especially for publicly accessible REST APIs. We found that while the identified strategies mitigate some of the challenges, they simultaneously raise further ones. Eventually, they result in tight organizational coupling and consumer lock-in. We visualize the relations between strategies and challenges in Fig. 2 and describe the two resulting problems in the following.
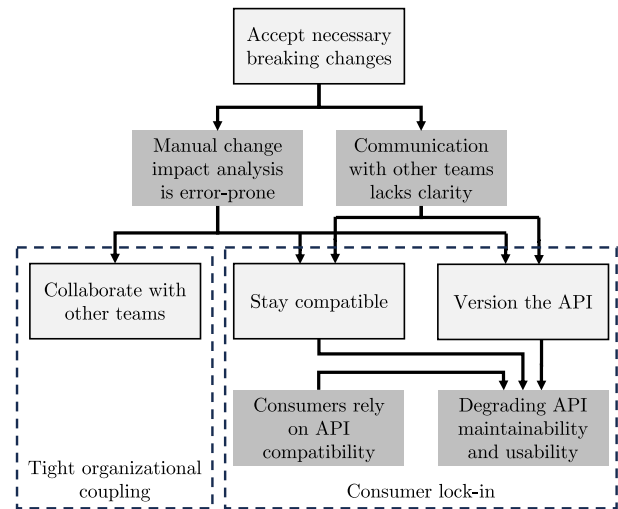
---

[16] https://avro.apache.org/docs/1.11.1/.

[17] https://thrift.apache.org.

Fig. 2. The relationships for a subset of strategies (light grey) and challenges (dark grey) resulting in two underlying problems (dashed).

6.1. Tight organizational coupling

While developers must accept and deal with breaking API changes during the software evolution process (cf. Section 4.2), they encounter challenges with understanding the impact of changes (cf. Section 5.2) and communicating with other teams (cf. Section 5.4). While tools like openapi-diff extract structural changes between two API versions, providers miss tools for automatically extracting API changes from changes in their implementation, especially behavioral changes, forcing them to assess the change impact manually. *"They [providers] also make changes to the interface, breaking changes, without even them knowing, so they just made mistakes. We have to tell them, hey, do you know that you changed your interface?"* C3-P1. Moreover, the providers cannot fully anticipate the potential impact of changes on consumers due to missing consumer documentation or accessibility. The change notifications might not reach all consumers or are forgotten, e.g., when contained in a larger mail or verbally mentioned during a meeting. If the consumers do receive a list of all the breaking API changes, they must review them and, again, assess the impact on their own system manually. These challenges complicate the truly independent API evolution in loosely coupled systems and organizations. Many participants (10/17) reported that their system or a consuming system broke before because the other end did not correctly assess or notify the breaking API changes. *"Fortunately that is quite rare, but maybe it happens once a year [in production]"* C6-P1. *"I would say maybe once a year, maybe twice. [...] Obviously, more often in the test system"* C3-P2. End users encountering a resulting failure typically report it to the developers of the system they are interacting with and hold them accountable, independently of the root cause.

As coping strategies, providers try to stay backward compatible (cf. Section 4.3), introduce breaking changes as new API versions (cf. Section 4.4), and closely collaborate with other teams during the migration (cf. Section 4.5). The close collaboration works well for cooperative teams and system-internal event-driven communication, and helps to resolve failures during integration and in production more quickly. As a downside, it shifts the overhead towards organizational communication and regular meetings, and creates implicit knowledge distributed between team members. *"You see, the whole topic is really organizational-heavy, organizational and planning-heavy. [...] The technical part is then just doing it"* C12-P1. We call this problem *tight organizational coupling*.

6.2. Consumer lock-in

Communication often lacks clarity (cf. Section 5.4) for external and uncooperative consumer teams, making close collaboration impossible. Instead, providers try to stay backward compatible (cf. Section 4.3) and introduce breaking changes as a new API version (cf. Section 4.4). However, multiple versions increase the source code and API size and complexity by continuously adding functionality or workflows to the initial design. A dynamic API design improves structural backward compatibility but often shifts the problem, i.e., further increases the complexity and creates semantic breaking changes instead of structural ones. *"And these are often the problems that are only noticed once it [the system] is not working anymore. And you don't know why"* C12-P1. The increasing source code complexity requires extensive regression testing to ensure compatibility with old API versions. *"Before releasing, we run each and every test we have - and this is a quite huge test suite - over the last release version again to make sure nothing did break in between, since the last release"* C5-P2. These old API versions could be shut down once all consumers migrated their calls. *"So you have to reach out to everyone who's using your API and hope for the best. Hope that they will update"* C3-P1. Still, many consumers rely on the API version they integrated with and do not migrate to the new version (cf. Section 5.3).

This migration reluctance forces providers to maintain an increasingly large number of API versions with every breaking change. *"We want to get rid of some things in our API that we already removed for newer versions or refactored for newer versions, but we just can't because this customer is still using it"* C5-P2. Hence, the API cannot evolve sustainably and continuously degrades the initial API design, maintainability, and usability (cf. Section 5.5). We call this problem *consumer lock-in* because the consumers force their providers to continue supporting all outdated API versions in use. It complicates the development of new features which would break existing calls or workflows, and increases the implementation and maintenance overhead and the technical debt with each additional API version. *"The cost of these workarounds that we do, I don't know. I don't dare to estimate that"* C2-P1.

We observed that participants avoid the consumer lock-in internally, where they regularly break the API and migrate the calls themselves (cf. Section 4.6). Governmental services avoid both tight organizational coupling and consumer lock-in by regularly introducing breaking changes (cf. Section 5.6). This gives full freedom to the provider but dissatisfies the consumers and is hence unfeasible in the context of business relationships.

7. Discussion

This section proposes two research directions to address the two main challenges that result in tight organizational coupling and consumer lock-in. Furthermore, we discuss the threats to the validity of our study.

7.1. Open research directions

We propose two open research directions aiming at mitigating both underlying problems of API evolution, tight organizational coupling and consumer lock-in. Considering the two main causes, we propose automating the change impact analysis to improve change notification accuracy and trustworthiness and researching effective ways to communicate changes to other teams to improve notification reliability and clarity.

7.1.1. Automating change impact analysis in MSA

The manual change impact analysis challenge (cf. Section 5.2) leads to tight organizational coupling and hesitant consumer migrations resulting in consumer lock-in. Multiple approaches constructed SDGs statically (Laverdière et al., 2015; Ma et al., 2019) or dynamically (Liu et al., 2019; Guo et al., 2020) to perform change impact analysis on

the service level. As a limitation, they do not consider individual API calls or behavioral breaking changes. Other approaches trace method invocations at runtime (Cai and Thain, 2016; Popescu et al., 2012; Cai and Fu, 2022) to construct more detailed call graphs. However, they require access to the runtime environment of the services, which is not available for external consumers and again would require tight organizational coupling. A different approach proposed by Chaturvedi and Binkley (2021) identifies changed WSDL operations based on the source code's structural and behavioral changes.

Based on the approach by Chaturvedi and Binkley (2021), we motivate researchers to split the change impact analysis based on the service boundaries: first, analyzing the impact of the provider's source code changes on the provider's APIs, and second, the impact of the provider's API changes on the consumers' source code. This enables providers to publish a complete list of API changes and allows consumers to migrate on their own terms, reducing the tight organizational coupling. Furthermore, an accurate change impact analysis mitigates the risk of unexpected changes breaking the system during migration and therefore increases consumers' trust and reduces their hesitation to migrate, which currently results in consumer lock-in.

7.1.2. Providing effective change communication for teams

The communication challenge (cf. Section 5.4) leads to the backward compatibility requirement and consumer lock-in. Similar to our results, multiple studies (Espinha et al., 2015; Sohan et al., 2015; Bogart et al., 2021) identified e-mails and online platforms, e.g., Twitter and homepages, as the main communication channels for change notifications. Bogart et al. (2021) found that most developers felt overwhelmed by the number of change notifications and rather participated in planned migrations, where providers felt personally obligated to help resolve breaking changes. Hora et al. (2018) found that deprecated library APIs, which produce warning messages during development, caused 50% more reactions than deprecated REST APIs.

Hence, we motivate the research of effective and efficient communication approaches for providers to communicate API changes in MSA. *"Yeah, for example, a system where you register your APIs and where you can publish updates. For example, where you can say, hey we are removing this field, and it automatically notifies everyone that needs these APIs"* C3-P3. Addressing the communication challenge alleviates consumer lock-in by reliably notifying affected consumers with customized change logs instead of flooding them with a generic list of changes that are potentially incorrect on arrival. Reliable and targeted change notifications that require less time to understand motivate consumers to review and implement them.

7.2. Threats to validity

In this section, we describe the threats to the validity of our study (Wohlin et al., 2012), their implications, and explain our mitigation strategies.

7.2.1. Construct validity

We may have misinterpreted participants' answers or misunderstood their terminology. We added multiple background questions to our interview guide to mitigate this threat (Wohlin et al., 2012). The questions set the context and clarified the terminology used by the interviewer and participant during each interview, e.g., what they consider a microservice, API, endpoint, team, or consumer. Furthermore, we scheduled enough time to conduct each interview, allowed the participants to speak freely, asked follow-up questions, and repeated important statements back to them to make sure we understood them correctly (Adams, 2015). Finally, we shared the study results with all 17 participants for feedback and validation (Runeson et al., 2012). We received 13 responses, whereof two participants had minor remarks which we incorporated, and the others fully agreed with our interpretations.

7.2.2. Internal validity

Our study design may have generated incorrect or incomplete results. We mitigated this threat by following guidelines for qualitative studies (Seaman, 2008; Adams, 2015), e.g., asking open-ended non-judgemental questions during the interviews, encouraging participants to speak freely, and improving the interview guide after gaining insights from previous interviews. We finished each interview with the open question "Do you have additional thoughts you want to express?" to encourage the participants to discuss any additional topics they considered essential for us.

Another threat to the internal validity is that we did not consider the details of individual APIs that the participants maintain, e.g., the API size or usage frequencies. Such API details might influence the applicable strategies and perceived challenges. While many participants discussed that working around breaking changes (cf. Section 4.3.1) and exposing multiple API versions (cf. Section 4.4.2) increases the API size over time, we do not know their actual magnitudes, e.g., the number of REST API endpoints or message queues. The challenge of understanding change impacts (cf. Section 5.2) and the maintenance and usability degradation (cf. Section 5.5) may exacerbate with increasing API sizes and complexities. Still, both challenges are already acknowledged by many participants throughout multiple companies and system architectures. Similarly, we did not explicitly ask for the usage frequencies of the participants' APIs. However, only one participant mentioned increased performance when discussing event-driven communication, and another participant found excessive requests a networking challenge. Hence, we presume that this API detail would not significantly influence our results as the participants did not primarily consider the frequency of use when discussing microservice API evolution strategies and challenges. Instead, they focused on the design and organizational topics, independent of the runtime usage.

As a third threat, we did not consider the number and type of consumed systems from the consumer perspective. While we explicitly asked participants about the consumed APIs, we learned that they often did not know the details and removed the question during the analysis. However, these details would enrich some of our results by better setting them into relation. For instance, only some participants (6/17) abstract external APIs they consume (cf. Section 4.7), but we do not know the total number of participants consuming company-external APIs. Similarly, we do not know the total number of participants consuming governmental services (cf. Section 5.6). A lower total number of potentially affected participants would indicate a higher support and relevancy of these findings. From the provider perspective, we did not explicitly ask for the number and type of consumers. However, the participants discussed the types of consumers whenever they considered it noteworthy. For instance, they observed that company-external consumers are more cautious and reluctant when migrating to a new API version (cf. Section 5.3) and communication with company-external or unfamiliar teams lacks clarity (cf. Section 5.4). However, some findings could benefit from additional context, e.g., how exactly the participants collaborate with different types of consumers (cf. Section 4.5) or if company-internal consumers rely on API compatibility as often as external ones (cf. Section 5.3). We propose a follow-up study investigating such specifics of the identified strategies and challenges as future work.

Finally, due to our iterative study design, the participants could not estimate the severities of the individual challenges. We did not have the complete findings during earlier interviews, and decided to reduce observer bias (Glaser and Strauss, 1967) by avoiding leading questions during later interviews. Instead, we included interview quotes to convey the subjective importance expressed by participants. We then used the qualitative results to construct our overall theory based on the most prominent strategies and challenges.

7.2.3. External validity

Our study results may not be generalizable to other teams and organizations. To mitigate this threat, we sampled practitioners from 11 companies with various industry fields and sizes, and whereof 8 are international companies. Similarly, our 17 interview participants have diverse educational backgrounds, years of experience, and technical roles. We achieved this by contacting colleagues with diverse technical roles and backgrounds in multiple industry fields during the snowball sampling process (Baltes and Ralph, 2022). We further mitigated the study's threat to external validity by conducting interviews until our results became stable throughout the various companies and interview partners, indicating theoretical saturation (van Rijnsoever, 2017). We did not report on strategies and challenges mentioned by less than five participants or three companies, i.e., less than a quarter each. One challenge missed this threshold by one participant: low-quality documentation of external APIs (4/17). Finally, we grounded our findings by connecting them to previous and related works, thereby supporting and strengthening the results.

7.2.4. Reliability

We may have generated subjective and unreproducible results. We mitigated this threat by formulating and continuously refining an interview guide (Adams, 2015). Additionally, we tested the questions' clarity with a pilot interview. We followed established coding guidelines (Corbin and Strauss, 1990; Goodrick and Rogers, 2015) to analyze the interview transcripts and create the codes, categories, and overall theory. Furthermore, we applied researcher triangulation to reduce subjective bias (Runeson et al., 2012). Therefore, the second and third authors independently analyzed two random interviews and we discussed the identified codes and categories, further refining them until we reached a coder agreement (Campbell et al., 2013). In this paper, we only reported results mentioned by at least five interview partners from at least three companies to mitigate observer bias. Finally, we provide a replication package (Lercher et al., 2023) containing the interview guide and final code book.

8. Related work

In the following, we report on existing literature with a focus on studies that investigated API evolution strategies and challenges. Note, we present and discuss further literature related to the individual communication techniques, evolution strategies, and evolution challenges that we identified during our study in the corresponding sections.

API evolution is extensively studied. Though, Lamothe et al. (2021) found that 63.9% of analyzed survey papers focused on API evolution in Java libraries. Dig and Johnson (2006) introduced the terms breaking and non-breaking changes when studying Java API changes. They proposed five strategies for introducing backward-compatible changes. Deprecation instead of deletion of functionality allows consumers to use previous versions while marking them as outdated. Delegation forwards outdated method calls to successor methods. Naming conventions, e.g., version numbers in method and class names, help developers to navigate versioned APIs. Runtime switches dynamically load old library versions instead of raising runtime errors. Interface querying allows consumers to request a specific method version via a facade object. Bogart et al. (2021) conducted interviews and surveys with developers and identified several strategies to reduce or delay consumer-side breaking change impacts. Providers maintained old interfaces to prolong the transition period for consumers. They released major and minor versions for features and fixes in parallel. Consumers then decided when to migrate to the next major version. As a resulting challenge, providers had to maintain several separate interfaces, called opportunity cost. Wu et al. (2016) analyzed Java API changes, and they and Bogart et al. (2021) recommended consumers to encapsulate external API dependencies within a facade object to reduce the dependencies and, hence, the change impact.

Li et al. (2013) discovered that REST APIs are more change-prone than Java APIs after conducting an empirical study. Furthermore, they identified six additional challenges for REST API evolution compared to source code evolution, e.g., deleted methods are not recoverable since previous versions are not accessible after shutdown and REST APIs require authorization over the network. Lamothe et al. (2021) concluded their literature survey with the need for web API evolution research. Aksakalli et al. (2021) identified synchronous communication with REST and event-driven publish–subscribe communication as the most preferred communication patterns in MSA. They concluded that both approaches are oftentimes combined: REST APIs publish explicit interfaces to the outside world, while publish–subscribe communication loosely couples the internal service architecture for state-changing operations.

Zimmermann et al. (2020) introduced the microservice API pattern (MAP) framework for API design and evolution. They proposed five evolution patterns: running multiple versions in parallel, limited and unlimited lifetime guarantees for backward compatibility, experimental previews without promising stability, and aggressive obsolescence, i.e., shutting down previous versions with a fixed date (Lübke et al., 2019). Espinha et al. (2015) interviewed 6 developers who criticized that early API versions are unstable and change regularly without notice. They formulated basic recommendations, such as staying backward-compatible, exposing some stability status information, and monitoring the system to identify the consumers per feature. Similarly, Wu et al. (2016) recommended Java API providers should publish their API stability expectations. Sohan et al. (2015) investigated REST API evolution strategies and found both strategies of running single and multiple concurrent versions in practice. They identified the disconnected source code, documentation, and change log artifacts as challenging because developers must manually link all artifacts to identify the changes and their impact. The authors recommended generating customized change logs per consumer. Neumann et al. (2021) analyzed 500 REST APIs and found that almost half of them automatically generate documentation, e.g., with Swagger UI. The other half provides textual information in varying granularity, complicating automated change identification and impact analysis.

Baškarada et al. (2020) investigated microservice opportunities and challenges by interviewing 19 practitioners. Amongst other organizational challenges, they reported that API changes require intensive communication and coordination. Furthermore, they found that very few practitioners had experience with event-driven communication. Chen et al. (2021) conducted a grey literature review for microservice API technologies and concerns. They identified three concerns regarding API gateway design, API versioning, and API testing and test case generation. Wu et al. (2022) analyzed microservice-related StackOverflow questions and identified technical communication as the major challenge in the service construction phase. In the governance phase, they identified further challenges, including defining API standards and API gateway design. The proposed solution strategies for these challenges included utilizing event-driven communication and creating GitHub examples for API usage scenarios. de Toledo et al. (2021) interviewed 25 practitioners about architectural technical debts in microservice systems. Interview partners acknowledged that poor REST API design results in API instability, regular breaking changes, and increased difficulty of maintaining backward compatibility. The authors proposed the API-first approach as a solution, which also reduces the tight coupling between services. Zhang et al. (2019) interviewed 13 companies and found that inappropriate service boundaries lead to tight coupling and regular changes. The multiple parallel versions then increased the debugging complexity. Similarly, Bushong et al. (2021) reported that the system design has a direct impact on microservice evolution and proposed detecting antipatterns in source code and identifying technical debt as future research directions. Koçi et al. (2023) analyzed web APIs based on the consumers' behavioral patterns and proposed corresponding API changes, such as merging two endpoints usually called consecutively into a single one.

In summary, related works reported on the strategies for backward compatibility and versioning to support outdated consumers and recommended the API-first approach. Regarding the challenges, they reported on communication and coordination overheads when migrating API changes, the importance of the API design's quality, and increased code complexity and technical debt when supporting multiple versions. However, they did not present and discuss the underlying reasons for the challenges or how to solve them sustainably. In this work, we formulated a comprehensive list of strategies and challenges for microservice API evolution actively used in practice. We discovered that close communication and collaboration between teams is not only a well-known challenge but an actively performed and expected strategy in MSA. We further identified that the collaboration strategy results from the manual change impact analysis challenge and leads to the problem of tight organizational coupling. Moreover, we discovered that the established strategies for compatibility and versioning create a new problem of consumer lock-in. In turn, the consumer lock-in degrades the API design and increases technical debt without any resolution strategy. To the best of our knowledge, we are the first to formulate such a comprehensive list, to define the problems of tight organizational coupling and consumer lock-in including their underlying challenges, and to propose open research directions addressing them.

9. Conclusion

The microservice API evolution process suffers from the loose coupling between services and leads to communication overheads and backward compatibility necessity. In this work, we conducted semi-structured interviews with 17 developers, architects, and managers in 11 companies and reported their strategies and challenges for API evolution.

In summary, we discovered six strategies and six challenges for REST and event-driven communication techniques. The strategies mainly focus on API backward compatibility, versioning, and close collaboration between teams when introducing breaking changes. The challenges illuminate the manual change impact analysis efforts, ineffective communication of changes, and consumer reliance on outdated API versions. From our findings, we formulated relationships between the strategies and challenges and discovered two underlying problems in the microservice API evolution process. Tight organizational coupling undermines the loose technical coupling of services by regularly requiring communication and collaboration between development teams. Consumer lock-in increases technical debt and degrades the API design over time by enforcing continuous support for outdated API versions without a clear resolution strategy. We proposed two relevant research directions to mitigate these two problems.

Future work includes studying the evolution of event-driven communication in particular, which many participants disregarded during our study. Furthermore, based on our insights, we propose a new study that investigates the two underlying problems in microservice API evolution and evaluates approaches to mitigate them.

CRediT authorship contribution statement

Alexander Lercher: Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Validation, Visualization, Writing – original draft, Writing – review & editing, Conceptualization. **Johann Glock:** Formal analysis, Validation, Writing – original draft. **Christian Macho:** Formal analysis, Funding acquisition, Validation, Writing – original draft, Writing – review & editing. **Martin Pinzger:** Funding acquisition, Methodology, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The interview transcripts are confidential, but the interview guide and codebook are available as a replication package.

References

Adams, W.C., 2015. Conducting semi-structured interviews. In: Handbook of Practical Program Evaluation. John Wiley & Sons, Ltd, pp. 492–505, (Chapter 19).

Adolph, S., Hall, W., Kruchten, P., 2011. Using grounded theory to study the experience of software development. Empir. Softw. Eng. 16 (4), 487–513. http://dx.doi.org/10.1007/s10664-010-9152-6.

Akbulut, A., Perros, H.G., 2019. Performance analysis of microservice design patterns. IEEE Internet Comput. 23 (6), 19–27. http://dx.doi.org/10.1109/MIC.2019.2951094.

Aksakalli, I.K., Çelik, T., Can, A.B., Tekinerdoğan, B., 2021. Deployment and communication patterns in microservice architectures: A systematic literature review. J. Syst. Softw. 180, 111014. http://dx.doi.org/10.1016/j.jss.2021.111014, URL: https://www.sciencedirect.com/science/article/pii/S0164121221001114.

Alshuqayran, N., Ali, N., Evans, R., 2016. A systematic mapping study in microservice architecture. In: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications. SOCA, pp. 44–51.

Assunção, W.K., Krüger, J., Mosser, S., Selaoui, S., 2023. How do microservices evolve? An empirical analysis of changes in open-source microservice repositories. J. Syst. Softw. 204, 111788. http://dx.doi.org/10.1016/j.jss.2023.111788, URL: https://www.sciencedirect.com/science/article/pii/S0164121223001838.

Baltes, S., Ralph, P., 2022. Sampling in software engineering research: a critical review and guidelines. Empir. Softw. Eng. 27 (4), 94. http://dx.doi.org/10.1007/s10664-021-10072-8.

Baškarada, S., Nguyen, V., Koronios, A., 2020. Architecting microservices: Practical opportunities and challenges. J. Comput. Inf. Syst. 60 (5), 428–436. http://dx.doi.org/10.1080/08874417.2018.1520056, arXiv:https://doi.org/10.1080/08874417.2018.1520056.

Beaulieu, N., Dascalu, S.M., Hand, E., 2022. API-first design: A survey of the state of academia and industry. In: Latifi, S. (Ed.), ITNG 2022 19th International Conference on Information Technology-New Generations. Springer International Publishing, Cham, pp. 73–79.

Biernacki, P., Waldorf, D., 1981. Snowball sampling: Problems and techniques of chain referral sampling. Sociol. Methods Res. 10 (2), 141–163. http://dx.doi.org/10.1177/004912418101000205, arXiv:https://doi.org/10.1177/004912418101000205.

Biswas, S., Mall, R., Satpathy, M., Sukumaran, S., 2011. Regression test selection techniques: A survey. Informatica (Ljubljana) 35 (3).

Bloch, J., 2006. How to design a good API and why it matters. In: Companion To the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA '06, Association for Computing Machinery, New York, NY, USA, pp. 506–507. http://dx.doi.org/10.1145/1176617.1176622.

Bogart, C., Kästner, C., Herbsleb, J., Thung, F., 2021. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. ACM Trans. Softw. Eng. Methodol. 30 (4), http://dx.doi.org/10.1145/3447245.

Brito, G., Mombach, T., Valente, M.T., 2019. Migrating to GraphQL: A practical assessment. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 140–150. http://dx.doi.org/10.1109/SANER.2019.8667986.

Brito, G., Valente, M.T., 2020. REST vs GraphQL: A controlled experiment. In: 2020 IEEE International Conference on Software Architecture. ICSA, pp. 81–91. http://dx.doi.org/10.1109/ICSA47634.2020.00016.

Brito, A., Valente, M.T., Xavier, L., Hora, A., 2020. You broke my code: understanding the motivations for breaking changes in APIs. Empir. Softw. Eng. 25 (2), 1458–1492. http://dx.doi.org/10.1007/s10664-019-09756-z.

Bushong, V., Abdelfattah, A.S., Maruf, A.A., Das, D., Lehman, A., Jaroszewski, E., Coffey, M., Cerny, T., Frajtak, K., Tisnovsky, P., Bures, M., 2021. On microservice analysis and architecture evolution: A systematic mapping study. Appl. Sci. 11 (17), http://dx.doi.org/10.3390/app11177856, URL: https://www.mdpi.com/2076-3417/11/17/7856.

Cai, H., Fu, X., 2022. D^{2}abs: A framework for dynamic dependence analysis of distributed programs. IEEE Trans. Softw. Eng. 48 (12), 4733–4761. http://dx.doi.org/10.1109/TSE.2021.3124795.

Cai, H., Thain, D., 2016. DistIA: A cost-effective dynamic impact analysis for distributed programs. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 344–355.

Campbell, J.L., Quincy, C., Osserman, J., Pedersen, O.K., 2013. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. Sociol. Methods Res. 42 (3), 294–320. http://dx.doi.org/10.1177/0049124113500475.

Cerny, T., Donahoo, M.J., Pechanec, J., 2017. Disambiguation and comparison of SOA, microservices and self-contained systems. In: Proceedings of the International Conference on Research in Adaptive and Convergent Systems. RACS '17, Association for Computing Machinery, New York, NY, USA, pp. 228–235. http://dx.doi.org/10.1145/3129676.3129682.

Cerny, T., Donahoo, M.J., Trnka, M., 2018. Contextual understanding of microservice architecture: Current and future directions. SIGAPP Appl. Comput. Rev. 17 (4), 29–45.

Chaturvedi, A., Binkley, D., 2021. Web service slicing: Intra and inter-operational analysis to test changes. IEEE Trans. Serv. Comput. 14 (3), 930–943. http://dx.doi.org/10.1109/TSC.2018.2821157.

Chen, F., Zhang, L., Lian, X., 2021. A systematic gray literature review: The technologies and concerns of microservice application programming interfaces. Softw. - Pract. Exp. 51 (7), 1483–1508. http://dx.doi.org/10.1002/spe.2967, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2967.

Conway, M.E., 1968. How do committees invent? Datamation 14 (4), 28–31.

Corbin, J.M., Strauss, A., 1990. Grounded theory research: Procedures, canons, and evaluative criteria. Qual. Sociol. 13 (1), 3–21.

Cremaschi, M., De Paoli, F., 2017. Toward automatic semantic API descriptions to support services composition. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (Eds.), Service-Oriented and Cloud Computing. Springer International Publishing, Cham, pp. 159–167.

Daigneau, R., 2012. Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. In: Addison-Wesley Signature Series (Fowler), Pearson Education, URL: https://books.google.at/books?id=wljZbEO8ZQC.

de Toledo, S.S., Martini, A., Sjøberg, D.I., 2021. Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. J. Syst. Softw. 177, 110968. http://dx.doi.org/10.1016/j.jss.2021.110968, URL: https://www.sciencedirect.com/science/article/pii/S0164121221000650.

Demircioğlu, E.D., Kalipsiz, O., 2022. API message-driven regression testing framework. Electronics 11 (17), http://dx.doi.org/10.3390/electronics11172671, URL: https://www.mdpi.com/2079-9292/11/17/2671.

Dig, D., Johnson, R., 2006. How do APIs evolve? A story of refactoring. J. Softw. Maint. Evol. Res. Pract. 18 (2), 83–107. http://dx.doi.org/10.1002/smr.328, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.328.

Ed-Douibi, H., Daniel, G., Cabot, J., 2020. OpenAPI Bot: A chatbot to help you understand REST APIs. In: Bielikova, M., Mikkonen, T., Pautasso, C. (Eds.), Web Engineering. Springer International Publishing, Cham, pp. 538–542.

Espinha, T., Zaidman, A., Gross, H.-G., 2015. Web API growing pains: Loosely coupled yet strongly tied. J. Syst. Softw. 100, 27–43. http://dx.doi.org/10.1016/j.jss.2014.10.014, URL: https://www.sciencedirect.com/science/article/pii/S0164121214002180.

Fielding, R.T., 2000. REST: Architectural Styles and the Design of Network-Based Software Architectures (Doctoral Dissertation). University of California.

Fokaefs, M., Stroulia, E., 2014. WSDarwin: Studying the evolution of web service systems. In: Bouguettaya, A., Sheng, Q.Z., Daniel, F. (Eds.), Advanced Web Services. Springer New York, New York, NY, pp. 199–223. http://dx.doi.org/10.1007/978-1-4614-7535-4_9.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA.

García, S., Strüber, D., Brugali, D., Berger, T., Pelliccione, P., 2020. Robotics software engineering: A perspective from the service robotics domain. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, pp. 593–604. http://dx.doi.org/10.1145/3368089.3409743.

Glaser, B., Strauss, A., 1967. The Discovery of Grounded Theory: Strategies for Qualitative Research. Observations (Chicago, Ill.), Aldine.

Godefroid, P., Lehmann, D., Polishchuk, M., 2020. Differential regression testing for REST APIs. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2020, Association for Computing Machinery, New York, NY, USA, pp. 312–323. http://dx.doi.org/10.1145/3395363.3397374.

Gómez, A., Iglesias-Urkia, M., Urbieta, A., Cabot, J., 2020. A model-based approach for developing event-driven architectures with AsyncAPI. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS '20, Association for Computing Machinery, New York, NY, USA, pp. 121–131. http://dx.doi.org/10.1145/3365438.3410948.

Goodrick, D., Rogers, P.J., 2015. Qualitative data analysis. In: Handbook of Practical Program Evaluation. John Wiley & Sons, Ltd, pp. 561–595. http://dx.doi.org/10.1002/9781119171386.ch22, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119171386.ch22.

Gos, K., Zabierowski, W., 2020. The comparison of microservice and monolithic architecture. In: 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design. MEMSTECH, pp. 150–153.

Gudkova, S., 2018. Interviewing in qualitative research. In: Ciesielska, M., Jemielniak, D. (Eds.), Qualitative Methodologies in Organization Studies: Volume II: Methods and Possibilities. Springer International Publishing, Cham, pp. 75–96. http://dx.doi.org/10.1007/978-3-319-65442-3_4.

Guo, X., Peng, X., Wang, H., Li, W., Jiang, H., Ding, D., Xie, T., Su, L., 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, pp. 1387–1397. http://dx.doi.org/10.1145/3368089.3417066.

Hanam, Q., Mesbah, A., Holmes, R., 2019. Aiding code change understanding with semantic change impact analysis. In: 2019 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 202–212. http://dx.doi.org/10.1109/ICSME.2019.00031.

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. Softw. Qual. J. 26 (1), 161–191. http://dx.doi.org/10.1007/s11219-016-9344-4.

Karlsson, S., Čaušević, A., Sundmark, D., 2020. Quickrest: Property-based test generation of openapI-Described restful APIs. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification. ICST, pp. 131–141. http://dx.doi.org/10.1109/ICST46399.2020.00023.

Knoche, H., Hasselbring, W., 2021. Continuous API evolution in heterogenous enterprise software systems. In: 2021 IEEE 18th International Conference on Software Architecture. ICSA, pp. 58–68. http://dx.doi.org/10.1109/ICSA51549.2021.00014.

Koçi, R., Franch, X., Jovanovic, P., Abelló, A., 2019. Classification of changes in API evolution. In: 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference. EDOC, pp. 243–249. http://dx.doi.org/10.1109/EDOC.2019.00037.

Koçi, R., Franch, X., Jovanovic, P., Abelló, A., 2023. Web API evolution patterns: A usage-driven approach. J. Syst. Softw. 198, 111609. http://dx.doi.org/10.1016/j.jss.2023.111609, URL: https://www.sciencedirect.com/science/article/pii/S0164121223000043.

Kopecký, J., Fremantle, P., Boakes, R., 2014. A history and future of web APIs. it - Inf. Technol. 56 (3), 90–97. http://dx.doi.org/10.1515/itit-2013-1035.

Koren, I., Klamma, R., 2018. The exploitation of OpenAPI documentation for the generation of web frontends. In: Companion Proceedings of the the Web Conference 2018. WWW '18, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, pp. 781–787. http://dx.doi.org/10.1145/3184558.3188740.

Krafzig, D., Banke, K., Slama, D., 2006. Enterprise SOA: Service-Oriented Architecture Best Practices, fifth print ed. Prentice Hall PTR, USA.

Kratzke, N., Quint, P.-C., 2017. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. J. Syst. Softw. 126, 1–16.

Lamothe, M., Guéhéneuc, Y.-G., Shang, W., 2021. A systematic review of API evolution literature. ACM Comput. Surv. 54 (8), http://dx.doi.org/10.1145/3470133.

Laverdière, M.-A., Berger, B.J., Merloz, E., 2015. Taint analysis of manual service compositions using cross-application call graphs. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering. SANER, pp. 585–589. http://dx.doi.org/10.1109/SANER.2015.7081882.

Lehman, M., 1979. On understanding laws, evolution, and conservation in the large-program life cycle. J. Syst. Softw. 1, 213–221.

Lercher, A., Glock, J., Macho, C., Pinzger, M., 2023. Microservice API evolution in practice: A study on strategies and challenges - replication package. http://dx.doi.org/10.5281/zenodo.8275798.

Leung, H., White, L., 1989. Insights into regression testing (software testing). In: Proceedings. Conference on Software Maintenance - 1989. pp. 60–69. http://dx.doi.org/10.1109/ICSM.1989.65194.

Li, J., Xiong, Y., Liu, X., Zhang, L., 2013. How does web service API evolution affect clients? In: 2013 IEEE 20th International Conference on Web Services. pp. 300–307. http://dx.doi.org/10.1109/ICWS.2013.48.

Liu, H., Zhang, J., Shan, H., Li, M., Chen, Y., He, X., Li, X., 2019. JCallGraph: Tracing microservices in very large scale container cloud platforms. In: Da Silva, D., Wang, Q., Zhang, L.-J. (Eds.), Cloud Computing – CLOUD 2019. Springer International Publishing, Cham, pp. 287–302.

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, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3361149.3361164.

Ma, S.-P., Fan, C.-Y., Chuang, Y., Liu, I.-H., Lan, C.-W., 2019. Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Gener. Comput. Syst. 100, 724–735. http://dx.doi.org/10.1016/j.future.2019.05.048, URL: https://www.sciencedirect.com/science/article/pii/S0167739X19302614.

Mendonca, N.C., Aderaldo, C.M., Camara, J., Garlan, D., 2020. Model-based analysis of microservice resiliency patterns. In: 2020 IEEE International Conference on Software Architecture. ICSA, pp. 114–124. http://dx.doi.org/10.1109/ICSA47634.2020.00019.

Neumann, A., Laranjeiro, N., Bernardino, J., 2021. An analysis of public REST web service APIs. IEEE Trans. Serv. Comput. 14 (4), 957–970. http://dx.doi.org/10.1109/TSC.2018.2847344.

Newman, S., 2021. Building Microservices: Designing Fine-Grained Systems, second ed. O'Reilly Media, p. 616.

Nguyen, H.A., Nguyen, T.N., Dig, D., Nguyen, S., Tran, H., Hilton, M., 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, pp. 819–830. http://dx.doi.org/10.1109/ICSE.2019.00089.

O'Connor, C., Joffe, H., 2020. Intercoder reliability in qualitative research: Debates and practical guidelines. Int. J. Qual. Methods 19, 1609406919899220. http://dx.doi.org/10.1177/1609406919899220, arXiv:https://doi.org/10.1177/1609406919899220.

Pautasso, C., Wilde, E., 2009. Why is the web loosely coupled? A multi-faceted metric for service design. In: Proceedings of the 18th International Conference on World Wide Web. WWW '09, Association for Computing Machinery, New York, NY, USA, pp. 911–920. http://dx.doi.org/10.1145/1526709.1526832.

Peng, C., Goswami, P., Bai, G., 2018. Fuzzy matching of OpenAPI described REST services. Procedia Comput. Sci. 126, 1313–1322. http://dx.doi.org/10.1016/j.procs.2018.08.081, URL: https://www.sciencedirect.com/science/article/pii/S1877050918313590. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 22nd International Conference, KES-2018, Belgrade, Serbia.

Popescu, D., 2010. Helios: impact analysis for event-based components and systems. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering. Vol. 2, pp. 531–532. http://dx.doi.org/10.1145/1810295.1810466.

Popescu, D., Garcia, J., Bierhoff, K., Medvidovic, N., 2012. Impact analysis for distributed event-based systems. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems. DEBS '12, Association for Computing Machinery, New York, NY, USA, pp. 241–251. http://dx.doi.org/10.1145/2335484.2335511.

Quiña-Mera, A., Fernandez, P., García, J.M., Ruiz-Cortés, A., 2023. GraphQL: A systematic mapping study. ACM Comput. Surv. 55 (10), http://dx.doi.org/10.1145/3561818.

Richardson, C., 2018. Microservices Patterns: With examples in Java. Manning.

Rubin, J., Rinard, M., 2016. The Challenges of Staying Together While Moving Fast: An Exploratory Study. In: 2016 IEEE/ACM 38th International Conference on Software Engineering. ICSE, pp. 982–993.

Runeson, P., Höst, M., Rainer, A., Regnell, B., 2012. Data analysis and interpretation. In: Case Study Research in Software Engineering. John Wiley & Sons, Ltd, pp. 61–76. http://dx.doi.org/10.1002/9781118181034.ch5, (Chapter 5).

Safwan, K.A., Servant, F., 2019. Decomposing the rationale of code commits: The software developer's perspective. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 397–408. http://dx.doi.org/10.1145/3338906.3338979.

Sato, D., 2014. Parallel change. https://martinfowler.com/bliki/ParallelChange.html. (Accessed 04 July 2023).

Schwichtenberg, S., Gerth, C., Engels, G., 2017. From open API to semantic specifications and code adapters. In: 2017 IEEE International Conference on Web Services. ICWS, pp. 484–491. http://dx.doi.org/10.1109/ICWS.2017.56.

Seaman, C.B., 2008. Qualitative methods. In: Shull, F., Singer, J., Sjøberg, D.I.K. (Eds.), Guide to Advanced Empirical Software Engineering. Springer London, London, pp. 35–62. http://dx.doi.org/10.1007/978-1-84800-044-5_2, (Chapter 2).

Serbout, S., Pautasso, C., 2023. An empirical study of web API versioning practices. In: Garrigós, I., Murillo Rodríguez, J.M., Wimmer, M. (Eds.), Web Engineering. Springer Nature Switzerland, Cham, pp. 303–318.

Shmerlin, Y., Hadar, I., Kliger, D., Makabee, H., 2015. To document or not to document? An exploratory study on developers' motivation to document code. In: Persson, A., Stirna, J. (Eds.), Advanced Information Systems Engineering Workshops. Springer International Publishing, Cham, pp. 100–106.

Sohan, S., Anslow, C., Maurer, F., 2015. A case study of web API evolution. In: 2015 IEEE World Congress on Services. pp. 245–252. http://dx.doi.org/10.1109/SERVICES.2015.43.

Sommer, P., Schellroth, F., Fischer, M., Schlechtendahl, J., 2018. Message-oriented middleware for industrial production systems. In: 2018 IEEE 14th International Conference on Automation Science and Engineering. CASE, pp. 1217–1223. http://dx.doi.org/10.1109/COASE.2018.8560493.

Sorgalla, J., Rademacher, F., Sachweh, S., Zündorf, A., 2018. On collaborative model-driven development of microservices. In: Mazzara, M., Ober, I., Salaün, G. (Eds.), Software Technologies: Applications and Foundations. Springer International Publishing, Cham, pp. 596–603.

Söylemez, M., Tekinerdogan, B., Kolukısa Tarhan, A., 2022. Challenges and solution directions of microservice architectures: A systematic literature review. Appl. Sci. 12 (11), http://dx.doi.org/10.3390/app12115507, URL: https://www.mdpi.com/2076-3417/12/11/5507.

Taibi, D., Lenarduzzi, V., 2018. On the definition of microservice bad smells. IEEE Softw. 35 (3), 56–62. http://dx.doi.org/10.1109/MS.2018.2141031.

Taibi, D., Lenarduzzi, V., Pahl, C., 2018. Architectural patterns for microservices: A systematic mapping study. In: Proceedings of the 8th International Conference on Cloud Computing and Services Science - CLOSER. SciTePress, INSTICC, pp. 221–232. http://dx.doi.org/10.5220/0006798302210232.

van Rijnsoever, F.J., 2017. (I can't get no) saturation: A simulation and guidelines for sample sizes in qualitative research. PLoS One 12 (7), 1–17. http://dx.doi.org/10.1371/journal.pone.0181689.

Wang, S., Keivanloo, I., Zou, Y., 2014. How do developers react to restful API evolution? In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (Eds.), Service-Oriented Computing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 245–259.

Wilde, E., Amundsen, M., 2019. The challenge of API management: API strategies for decentralized api landscapes. In: Companion Proceedings of the 2019 World Wide Web Conference. WWW '19, Association for Computing Machinery, New York, NY, USA, pp. 1327–1328. http://dx.doi.org/10.1145/3308560.3320089.

Wittern, E., Cha, A., Davis, J.C., Baudart, G., Mandel, L., 2019. An empirical study of graphql schemas. In: Yangui, S., Bouassida Rodriguez, I., Drira, K., Tari, Z. (Eds.), Service-Oriented Computing. Springer International Publishing, Cham, pp. 3–19.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer, Germany, http://dx.doi.org/10.1007/978-3-642-29044-2.

Wong, W., Horgan, J., London, S., Agrawal, H., 1997. A study of effective regression testing in practice. In: Proceedings the Eighth International Symposium on Software Reliability Engineering. pp. 264–274. http://dx.doi.org/10.1109/ISSRE.1997.630875.

Wu, W., Khomh, F., Adams, B., Guéhéneuc, Y.-G., Antoniol, G., 2016. An exploratory study of api changes and usages based on apache and eclipse ecosystems. Empir. Softw. Eng. 21 (6), 2366–2412. http://dx.doi.org/10.1007/s10664-015-9411-7.

Wu, M., Zhang, Y., Liu, J., Wang, S., Zhang, Z., Xia, X., Mao, X., 2022. On the way to microservices: Exploring problems and solutions from online Q&A community. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 432–443. http://dx.doi.org/10.1109/SANER53432.2022.00058.

Xavier, L., Brito, A., Hora, A., Valente, M.T., 2017. Historical and impact analysis of API breaking changes: A large-scale study. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 138–147. http://dx.doi.org/10.1109/SANER.2017.7884616.

Yasmin, J., Tian, Y., Yang, J., 2020. A first look at the deprecation of restful APIs: An empirical study. In: 2020 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 151–161. http://dx.doi.org/10.1109/ICSME46990.2020.00024.

Yongguo, J., Qiang, L., Changshuai, Q., Jian, S., Qianqian, L., 2019. Message-oriented middleware: A review. In: 2019 5th International Conference on Big Data Computing and Communications. BIGCOM, pp. 88–97. http://dx.doi.org/10.1109/BIGCOM.2019.00023.

Zdun, U., Navarro, E., Leymann, F., 2017. Ensuring and assessing architecture conformance to microservice decomposition patterns. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (Eds.), Service-Oriented Computing. Springer International Publishing, Cham, pp. 411–429.

Zdun, U., Wittern, E., Leitner, P., 2020. Emerging trends, challenges, and experiences in DevOps and microservice APIs. IEEE Softw. 37 (1), 87–91. http://dx.doi.org/10.1109/MS.2019.2947982.

Zhang, H., Li, S., Jia, Z., Zhong, C., Zhang, C., 2019. Microservice architecture in reality: An industrial inquiry. In: 2019 IEEE International Conference on Software Architecture. ICSA, pp. 51–60. http://dx.doi.org/10.1109/ICSA.2019.00014.

Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., Zdun, U., 2020. Introduction to Microservice API Patterns (MAP). In: Cruz-Filipe, L., Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S. (Eds.), Joint Post-Proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019). In: OpenAccess Series in Informatics (OASIcs), vol. 78, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 4:1–4:17. http://dx.doi.org/10.4230/OASIcs.Microservices.2017-2019.4, URL: https://drops.dagstuhl.de/opus/volltexte/2020/11826.

Alexander Lercher is a Ph.D. student in the Software Engineering Research Group at the University of Klagenfurt, Austria. He received his Master's degree (Dipl.-Ing.) with a focus on distributed systems from the University of Klagenfurt in 2021 with distinction. His research interests include microservice architectures, software and API evolution, and program analysis.

Johann Glock is a Ph.D. student in the Software Engineering Research Group (SERG) at the University of Klagenfurt, Austria. He received his M.Sc. in software engineering at the University of Applied Sciences Upper Austria, Campus Hagenberg, Austria in 2020. His research interests are in program analysis, program comprehension, and formal methods.

Christian Macho is a PostDoc assistant (tenure track) in the Software Engineering Research Group (SERG) at the University of Klagenfurt. He received his M.Sc. from the Technical University Vienna in March 2015 and his Ph.D. from the University of Klagenfurt in 2019 both with distinction. His research interests include software evolution, mining software repositories, program analysis, build systems, continuous integration, automated repair, and empirical studies in software engineering.

Martin Pinzger is a full professor at the University of Klagenfurt, Austria where he is heading the Software Engineering Research Group. His research interests are in software evolution, mining software repositories, program analysis, software visualization, and automating software engineering tasks. He is a member of ACM and a senior member of IEEE.