From Code Refactoring to API Refactoring: Agile Service Design and Evolution

Mirko Stocker and Olaf Zimmermann

University of Applied Sciences of Eastern Switzerland (OST), Oberseestrasse 10, 8640 Rapperswil, Switzerland
{mirko.stocker,olaf.zimmermann}@ost.ch

Abstract. Refactoring is an essential agile practice; microservices are a currently trending implementation approach for service-oriented architectures. While program-internal code refactoring is well established, refactoring components on the architectural level has been researched but not adopted widely in practice yet. Hence, refactoring service Application Programming Interfaces (APIs) is not understood well to date. As a consequence, practitioners struggle with the evolution of APIs exposed by microservices. To overcome this problem, we propose to switch the refactoring perspective from implementation to integration and study how refactorings can be applied to the problem domain of agile service API design and evolution. We start with an empirical analysis and assessment of the state of the art and the practice. The contributions of this paper then are: 1) presentation of results from a practitioner survey on API change and evolution, 2) definitions for a future practice of API refactoring and 3) a candidate catalog of such API refactorings. We discuss these contributions and propose a research action plan as well.

Keywords: Agile practices · Application programming interfaces · Design patterns · Microservice architectures · Refactoring · Service evolution.

1 Introduction

Message-based remote Application Programming Interfaces (APIs) have become an important feature of modern distributed software systems [32]. Software functionality is increasingly provided not just through end-user facing applications, but also via APIs. These allow mobile clients, Web applications, and third parties to integrate API capabilities into their own applications and to combine different APIs to address new use cases. Such API-based integration approaches impact the software architectures and how these are developed and deployed: a growing number of distributed services must work together and communicate [8,18]. Independent of the technologies and protocols used, messages travel through one or several APIs, placing high demands on quality aspects of the API implementation – in many application scenarios, API implementations have to be highly available, reliable, responsive and scalable.

J. Barzen (Ed.). SummerSOC 2021, CCIS 1429. Springer Nature, 2021, pp. 174–193, 2021.
The final publication is available at Springer via https://doi.org/10.1007/978-3-030-87568-8_11.

During the first phases of developing a new API, especially in agile development, the focus is on implementing features (or even just a Minimum Viable Product[1], Walking Skeleton [7] or proof of concept). Questions about reliability, performance and scalability might not have a high priority yet; information on how the API will be used by clients is missing but required to make informed decisions. One could just guess and try to anticipate how potential clients will use the API, but that would not be prudent and violate agile values. For instance, one related lean principle is to make decisions at the most responsible moment[2]. Even if quality aspects of an API were to be highly ranked in the development team's priorities, the API implementation is likely to be created before any real client is using it; so even if the developers wanted to, there would still be no way to measure the actual usage and resulting quality characteristics.

Once the API is in production and receives real-world request traffic, quality issues start to surface. An API that has been published and is used by clients should not be changed ad hoc without carefully considering the positive and/or negative implications. Different strategies exist to mitigate these risks; many of these have been documented as design patterns [15]. For example, an API can use *Semantic Versioning*, so clients can compare versions of an API. The *Two in Production* pattern shows how multiple versions of an API can be provided so that clients can gradually switch to a new version.

Refactoring is a technique for improving the structure of a software system without changing its external/observable behavior [6] – typical concerns are maintainability and readability of the source code. The purpose of a refactoring can also be the alignment of the software with a design pattern [10]. Architectural refactorings [21] aim at improving the future evolvability of the software on the architecture level. Such coarser-grained refactorings "improve at least one quality attribute without changing the scope and functionality of the system" [28].

In this context, this paper presents the results of a practitioner survey that aims at understanding the reasons that force software architects and API developers to change an API and also the consequences of such changes. The survey addresses the following knowledge questions:

- *Q1: What causes API changes?*
- *Q2: How often are quality issues the cause of an API change?*
- *Q3: How do architects and developers mitigate the found issues?*

Taking the survey results into account, our second contribution is to propose a new form of refactoring – *API Refactoring* – to evolve APIs towards patterns:

Improve quality aspects of message-based remote APIs by offering refactoring and continuous API evolution practices, including actionable step-by-step instructions to help API architects and developers mitigate quality issues and evolve their APIs rapidly and reliably.

[1] http://www.syncdev.com/minimum-viable-product/

[2] http://wirfs-brock.com/blog/2011/01/18/agile-architecture-myths-2-architecture-decisions-should-be-made-at-the-last-responsible-moment/

The remainder of the paper is structured as follows. Section 2 introduces fundamental concepts and discusses the state of the art and the practice in API refactoring. Section 3 describes the survey design, Section 4 its results. Section 5 then proposes a definition of API refactoring and outlines a catalog of candidate refactorings, targeting the survey participants' wants and needs. Section 6 critically reviews our approach and presents an action plan for further research. Section 7 summarizes the paper and gives an outlook to future work.

2 Background/Context and Related Work

Microservice and API Domain Terminology. In a *message-based remote API*, a message is sent to a receiver, which must then serve it (e.g., dispatch to an object running inside it, or pass the message on to another receiver). The dispatch/delegation policy and its implementation is hidden from the sender, who can not make any assumptions about the receiver-side programming model and service instance lifecycles. In contrast, remote procedure calls do not only model remote service invocations, but also bind server-side subprogram runs to the service instances they are invoked on; such instances (e.g., remote objects) are visible across the network. While we deal with remote calls, we do not assume that these remote calls are remote *procedure* calls. Message-based remoting services can be realized in multiple technologies and platforms including RESTful HTTP, Web services (WSDL/SOAP), WebSockets, and even gRPC (despite its name). API calls come as HTTP *methods* operating on *resources* identified by URIs and appearing in messages as resource *representations*. A service is a component with a remote interface according to M. Fowler[3]; it exposes one or more API endpoints (for instance, resources in RESTful HTTP APIs) which bundle one or more operations (for instance, HTTP POST and GET operations) [32].

Gray Literature on API, Cloud Application and Service Design. Platform-specific design heuristics for designing highly available and reliable applications include the Amazon Web Services (AWS) Well-Architected Framework[4] or the Microsoft Azure Well-Architected Framework[5]. Such heuristics and guidelines focus on architectural aspects that lead to applications that are well suited to run in the respective clouds of these providers, but not on API implementation-design aspects. They also do not cover the refactoring of existing applications.

Combining refactorings and patterns is discussed in [10], which defines *refactoring to patterns* as "[..] the marriage of refactoring [..] with patterns, the classic solutions to recurring design problems. [..] We improve designs with patterns by applying sequences of low-level design transformations, known as refactorings." In our context, APIs can be refactored to align more closely with specific patterns, or to switch between pattern alternatives that have similar forces but different consequences.

[3] https://martinfowler.com/articles/injection.html
[4] https://aws.amazon.com/architecture/well-architected
[5] https://docs.microsoft.com/en-us/azure/architecture/framework

Academic Publications on API Design and Evolution. API design and evolution concern rather different stakeholders; therefore, we find related work in different research communities, including enterprise computing, software architecture, software evolution and maintenance, and service-oriented computing.

It has been researched how Web service API evolution affects clients; the challenges to be overcome include dealing with newly added rate limits [22] or changes in authentication and authorization rules [14]. This study classifies change patterns as refactorings and non-refactorings and that "web API evolves in limited patterns" and concludes that "a tool addressing all these patterns could potentially automate the migration of clients". Similarly, [24] identified and categorized API changes and how developers react to these changes by analyzing discussions on StackOverflow[6], a Q&A website for developers.

The refactoring of local, program-internal APIs and the role of such refactorings during software evolution and maintenance has been well researched empirically [4,3,5,12]; to the best of our knowledge, the refactoring and evolution of remote APIs, however, has not been investigated much yet. Design principles, architectural smells and refactorings for microservices are covered in a multifocal review [17]. Another publication show how applications can be split into services and how local interfaces can be transferred into remote ones [13].

Quality of service aspects of services has been the focus of several studies [16,20,26]. Other research has focused on operational qualities [1]. For instance, we presented a decision model for guiding architectural decisions on quality aspects in microservice APIs in our previous work [25].

3 Research Method

We initiated our empirical research with a survey among software engineering professionals involved in the design, implementation and maintenance of APIs to understand the reasons for API changes and the approaches chosen during API evolution. In this section, we summarize the survey design; our findings will then be discussed in Section 4.

Figure 1 shows the workflow during survey design. We chose Microsoft Forms as our survey tool because it provides suitable types of questions (multiple-choice with choices in random order; free-text; Likert scale) and privacy policy[7].

The questionnaire comprises 17 questions, organized into two parts. The first part contains questions about the context of the API and the technologies used; we asked participants that are involved in several projects to consider their overall experience when answering the questions. The second part of the survey focuses on the forces that drive API changes. We use different types of questions: single- and multiple-choice questions, with a free-form *other* option for additional answers where it makes sense. For questions where there was no natural ordering of the answers, the answers appeared in a random order. For

[6] https://stackoverflow.com/

[7] https://www.microsoft.com/en-us/servicesagreement/default.aspx

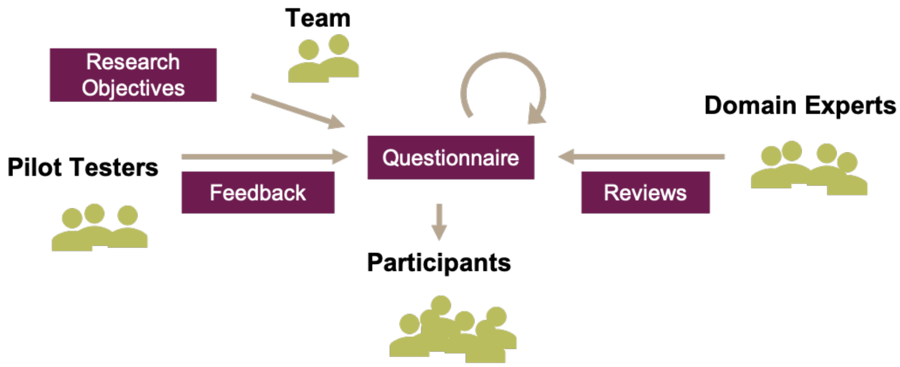

Fig. 1. We designed the questionnaire iteratively: a team-internal brainstorming led to an initial set of questions, which we discussed and refined internally. The first draft of the questionnaire was then sent to several domain experts acting as external reviewers, which provided feedback on the wording of the questions and answering options. A pilot among selected members of the target audience provided further feedback.

rating questions we used a five-point Likert scale (Never, Rarely, Sometimes, Often, Very Often).

Figure 2 lists the survey questions. We distributed the questionnaire through social media (Twitter, LinkedIn) and via e-mail to our personal networks of professional contacts, asking these contacts to fill out the survey themselves, but also to further distribute the questionnaire within their own networks.

4 Findings

At the time of this writing, we have received 64 completed questionnaires. In this section, we will first look at the usage context and used technologies of the APIs and then begin to answer our research questions with the responses to the second part of the survey.

API Usage Context and Technologies. The first part of the questionnaire aims at understanding the context in which the respondents' APIs are employed and the technologies used. While these responses do not directly contribute to answering our research questions, we can incorporate the articulated programming- or specification languages and other preferences when crafting solutions, examples, etc., as discussed in Section 6. Of our respondents, 64% have both internal and external clients, 31% have only internal and a mere 4% have just external clients. The numbers of users accessing the API are surprisingly large, with 54% of respondents stating that the API has more than 100 individual end users. The respondents' code bases vary between less than 10k lines of source code up to more than a million lines, with the majority in the 10k to 100k range. Asked

Part 1	Questions about the context and the technologies used:
Q1	Are the users of your API internal in your organization or do external third-parties use the API?
Q2	How many individual end users (e.g., in other teams or companies) access your API via client applications?
Q3	What is the relationship between the API provider (you) and the API client developers?
Q4	How large is the total provider-side code base in terms of lines of code approximately?
Q5	Do you develop the API specification first (e.g. using OpenAPI or Swagger), or do you write the code first and then let tools or frameworks create the specification?
Q6	Which specification or interface definition languages, if any, are you using?
Q7	Which of the following message exchange technologies do you use?
Part 2	**Questions about the forces that drive API changes:**
Q8	What were causes for API changes on your current/past projects?
Q9	Were there other causes for API changes in addition to those listed above? Please also indicate how often these occurred (Never, Rarely, Sometimes, Often, Very Often).
Q10	How often were quality issues the reason for an API change, as opposed to functional changes?
Q11	Which quality attributes were lacking and led to an API change?
Q12	Were there other quality-related problems that led to an API change? Please also indicate how often these occurred (Never, Rarely, Sometimes, Often, Very Often).
Q13	Have you ever decided against changing an API even though there were quality issues? If so, why?
Q14	Which of the following activities do you perform when making changes to an API?
Q15	How do you handle backwards compatibility?
Q16	If you version your API, where do you put the version identifier?
Q17	What do you find the most difficult challenge in API design and evolution?

Fig. 2. The survey questionnaire comprises 17 questions, organized into two parts. See https://forms.office.com/r/R0UjvPHALE for the complete questionnaire.

about the relationship between the API provider and the API client developers, 57% work together with the clients when changing the API.

We also asked participants about their usage of specification (interface definition) languages and message exchange technologies. The most used specification language is *OpenAPI / Swagger* (see Figure 3). OpenAPI specification can either be written first and code then generated from it; alternatively, the specification can be generated from annotated program code. Both approaches are common among our respondents, as Figure 4 shows. The last question in this part of the questionnaire asked about the technologies that are used to exchange messages. As can be seen in Figure 5, both RESTful and plain HTTP are in the lead, which fits the top response for specification languages.

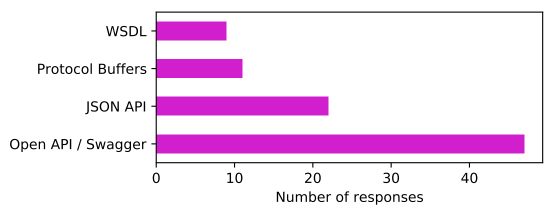

Fig. 3. Which specification or interface definition languages, if any, are you using? – Individually mentioned were Apollo, AsyncAPI, GraphQL, RAML, Apache Avro, XMI/UML, Markdown and "plain text documents without a structured format". Not surprisingly, *OpenAPI / Swagger* came out on top, but older languages such as *WSDL* are still in use as well.

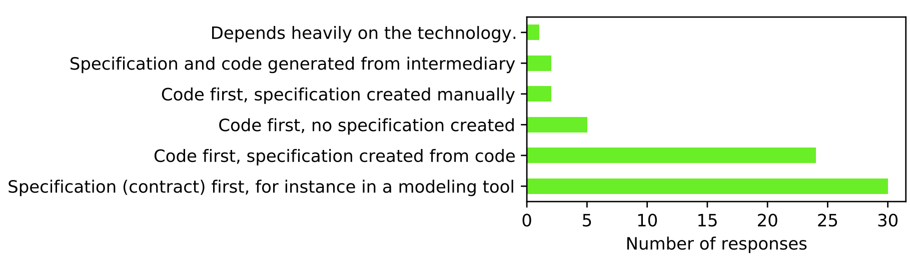

Fig. 4. Do you develop the API specification first, or do you write the code first and then let tools or frameworks create the specification? Overall, the two approaches appear to be equally common.

API Change Drivers. The second part of the questionnaire focuses on the forces that drive API changes and the mitigation tactics applied. To answer our

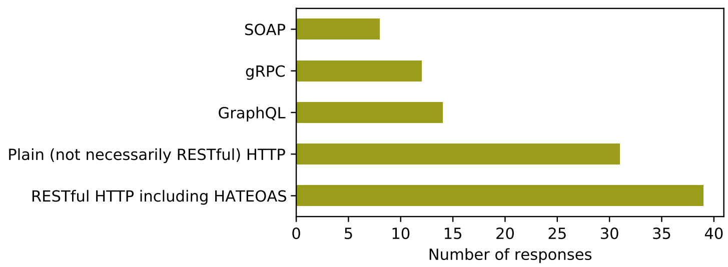

Fig. 5. Which of the following message exchange technologies do you use? – Singular responses were RabbitMQ, Server-Sent Events, Kafka and Java RMI. We were surprised by the high number of "RESTful HTTP including HATEOAS" responses, which seem to contradict earlier empirical observations [19].

first research question – *What causes API changes?* – we asked participants about the causes leading to an API change. As shown in Figure 6, modifications to APIs are most often caused by changed functional requirements, driven by the client or the provider (represented by the first two entries in the figure), whereas non-functional issues were occasionally the cause for a change as well.

The answers to our next question confirms this, as can be seen in Figure 7. Although functional changes are the main driver behind API changes, quality issues do occur and also result in API changes. Asked about which quality attributes were lacking (Figure 8), two of them stand out: *usability* and *maintainability*, followed by *performance* and *scalability*.

Change and Mitigation Tactics. When asked to select all actions they perform when changing an API, the top two answers were – unsurprisingly – "update specification or code" and "update API documentation" (see Figure 9). 59% of the respondents also said that they adjust the API version number, and almost half (43%) also adjust the API clients. These are two important findings for our next steps in designing API refactorings. Backwards compatibility seems to be an important consideration, as can be seen in the answers to our question about the handling of backwards compatibility shown in Figure 10.

As we saw above, quality issues in APIs do lead to changes, but lack of resources and other priorities also cause developers not to go through with a change. Figure 11 shows the main reasons for deciding against fixing quality issues. With the results from our survey, we can now attempt to find an answer to our initial questions:

- *Q1: What causes API changes?* The main reason for API changes is the introduction of new features, i.e., the provider drives the change. Client-driven changes are also common, but slightly less so.
- *Q2: How often are quality issues the cause of an API change?* Quality issues are common, but not the main driver of API changes. *Usability*, *performance*, and *maintainability* are the most common causes that led to API changes.

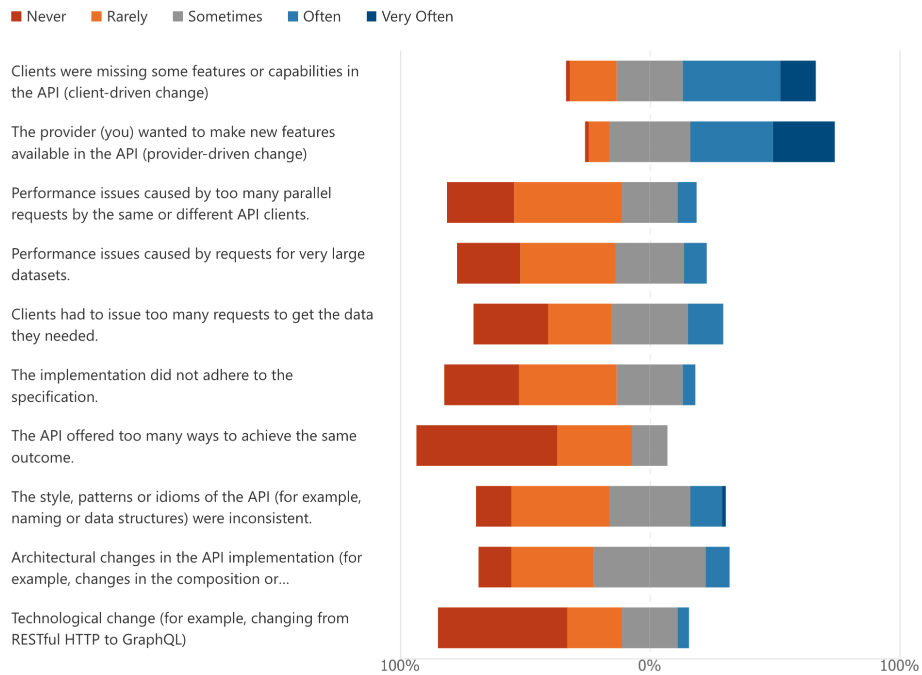

Fig. 6. What were causes for API changes on your current/past projects? The first two scales show that changed functional requirements are most often the cause that lead to API changes.

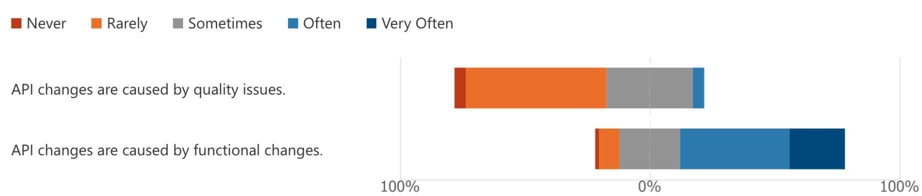

Fig. 7. How often were quality issues the reason for an API change, as opposed to functional changes? The answers reveal a very similar picture to the previous question reported in Figure 6.

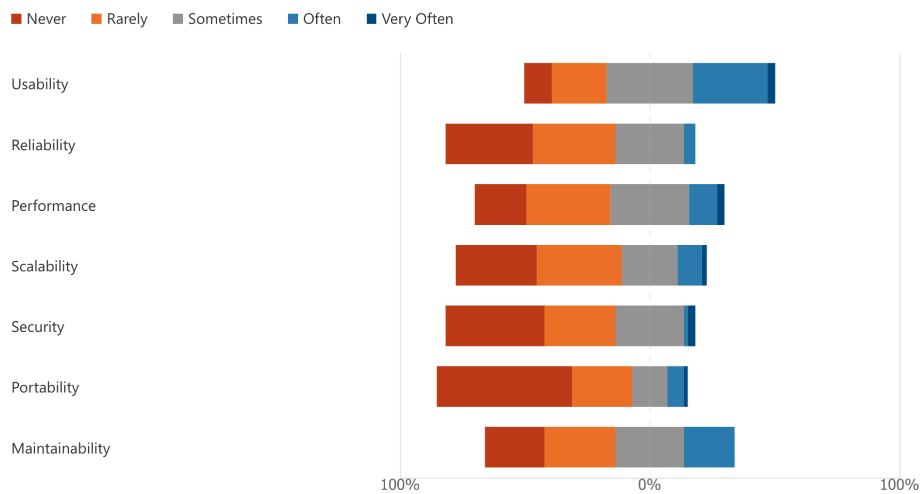

Fig. 8. Which quality attributes were lacking and led to an API change? *Usability* and *maintainability* stand out, followed by *performance* and *scalability*.

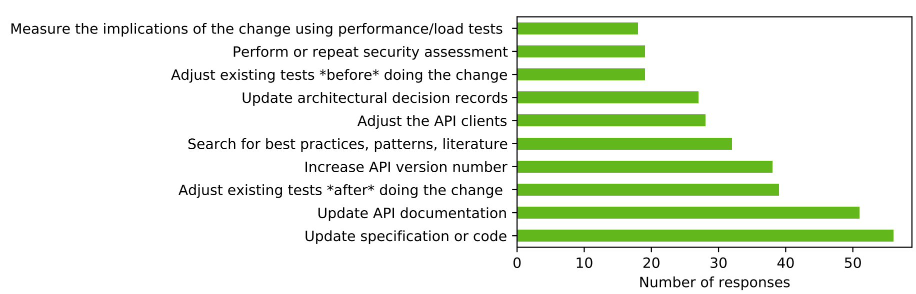

Fig. 9. Which of the following activities do you perform when making changes to an API? The approach to testing seems to differ among respondents: most update their tests after performing a change, suggesting that test-driven development is not used.

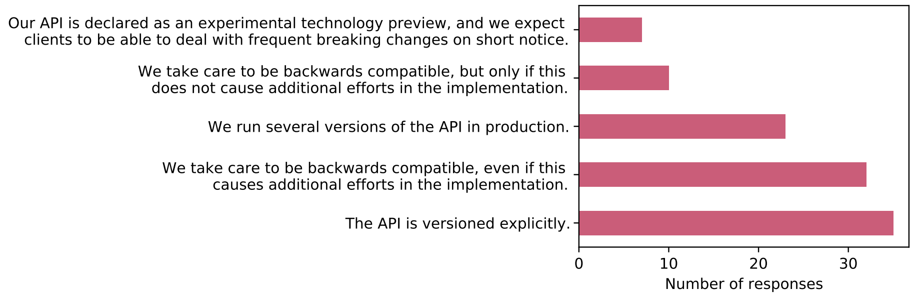

Fig. 10. How do you handle backwards compatibility? Most respondents version their API and try to keep backwards compatibility, accepting additional implementation efforts. Individual responses mentioned working with internal clients on the migration, using technologies that support deprecation of parameters or not supporting backwards compatibility at all.

– *Q3: How do architects and developers mitigate the found issues?* We gained some insights into the steps taken, especially on backwards compatibility, where versioning of the API is common. Apart from that, we were not able to answer this question in detail. Follow-up research will be required (e.g., case studies).

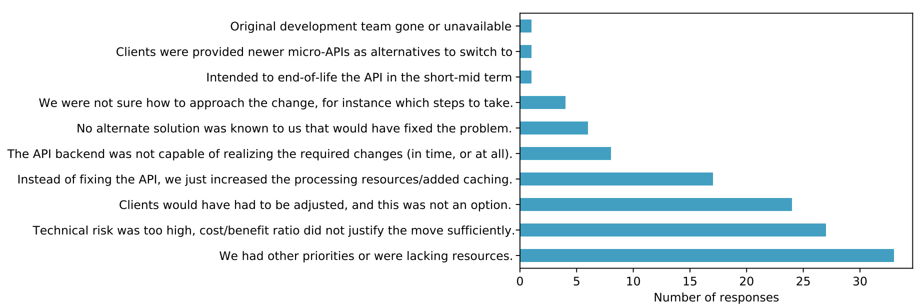

Fig. 11. Have you ever decided against changing an API even though there were quality issues? If so, why? Differing priorities and lack of resources were mentioned as common reasons for not changing an API, followed by the inability to also adjust clients.

The answers to Question 3 deliver candidate refactorings, the answers to Questions 1 and 2 the corresponding smells. Grounding our upcoming research in the survey results – also regarding the respondents' usage of specification languages and message exchange technologies – will increase the chances of getting accepted by developers in practice.

5 Towards API Refactoring

Our survey shows that quality issues in APIs do occur and lead to API changes. One technique to improve non-functional aspects of a software system is refactoring. In this section, we will first review existing types of refactoring and then propose a definition of API refactoring. A draft catalog of candidate API refactorings with scope and smell-resolution pairs derived from the survey results as well as additional input concludes the section.

Definition of Code Refactoring. Fowler defines refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [6]. An example of an everyday refactoring is the renaming of a local variable to improve the readability of the code. Not all code cleanup, such as formatting, falls under the umbrella of refactoring, and neither do changes to the software that add new features or fix bugs.

A refactoring is performed as a series of small steps that always keeps the software in a functioning state, and should be accompanied by a comprehensive suite of tests. For example, when moving a method between classes, a forwarding method that delegates calls from the original to the new class can be kept so that callers of the method can be migrated individually.

Refactoring is an essential agile practice: it is one of the extreme programming [2] practices, but also essential in test-driven development as part of the Red-Green-Refactor[8] cycle of development.

Definition of Architecture/Architectural Refactoring. The principles of refactoring can not only be applied to code-level software changes, but also on an architectural level. For instance, Stal defines software architecture refactoring that "improves the architectural structure of a system without changing its intended semantics" [21]. By only ruling out changes to intended semantics, and not the more strict observable behavior, an architectural refactoring allows more radical changes to a software. Zimmermann views Architecture Refactoring as "a coordinated set of deliberate architectural activities that remove a particular architectural smell and improve at least one quality attribute without changing the system's scope and functionality" [27].

An example of an architectural refactoring is the splitting of a monolithic system into several (micro-) services [29]. For example, an e-commerce software could be split into distinct product discovery and order checkout systems. The intended semantics of the overall software stays the same, but the new architecture might allow the individual development teams greater velocity.[9].

[8] http://www.jamesshore.com/v2/blog/2005/red-green-refactor

[9] One might argue that this violates the original definition of refactoring because the observable behavior is clearly and deliberately changed. In the refactoring example given earlier, a local variable is renamed, making it unlikely that the observable behavior of the software changes. But refactorings involving multiple classes, for example moving a method from one class to another, change the interfaces of these classes – possibly including public ones – in a backwards-incompatible man-

Proposed Definition of API Refactoring. Due to its hybrid character, we can define API refactoring starting from the above two definitions:

An API refactoring evolves the remote interface of a system without changing its feature set and semantics to improve at least one quality attribute.

API refactoring can be seen as a variant of architectural refactoring – interfaces and their implementations in components are architectural elements – with a focus on controlled evolution. In contrast to code refactorings, an API refactoring can affect the API behavior as observed by clients. The semantics of a specific operation might be changed in a refactoring, but not the overall feature set and semantics of the API. Operating on the boundaries of a system, API providers require an explicit evolution strategy to communicate API changes to clients and manage their expectations. Finally, the goal of an API refactoring is the improvement of at least one quality attribute (e.g., as mentioned in the survey: usability, performance, maintainability) and not to be able to introduce new features or fix bugs more efficiently. Figure 12 compares the different refactoring styles.

	Code Refactoring	API Refactoring	Architectural Refactoring
Improves	Internal structure, maintainability	Interface, many quality attributes	Overall structure, many quality attributes
Target	Source code	Specification, source code, API implementation components	Components, (sub)-systems, documentation, architectural decisions
Impact	Internal only	Externally visible, API clients	Potentially large group of stakeholders (3rd parties, other teams, operations), often system-wide or cross-cutting
Drivers	Code smells, test driven development	Changed client- and provider-side quality requirements, API smells	Changed requirements and constraints, architectural smells
Evolution	Not key, observable behavior unchanged	Important, compatibility with existing clients	Depends on visibility of changes
Examples	Rename, Extract Method, Move Class	Split Endpoint, Inline/Extract Information Holder	Migrate from SQL to NoSQL («De-SQL»), Split Subsystem

Fig. 12. Types of refactorings by scope and stakeholder concerns, according to [6,28] and our own analysis.

Let us consider a concrete application of an API refactoring in the following scenario (capitalized pattern names are from the Microservices API Patterns language[10]): You are a developer at Lakeside Mutual, a fictitious insurance company, responsible for the *Policy-Management-Backend* microservice. The service

ner. Whether this constitutes observable behavior depends on the viewpoint and expectations of the observer: an API client developer might notice and be directly affected by the change, but not the end user of the software. It has been shown that behavior-changing modifications still qualify as refactorings[11]

[10] https://www.microservice-api-patterns.org, also see [33,22,15,30,31]

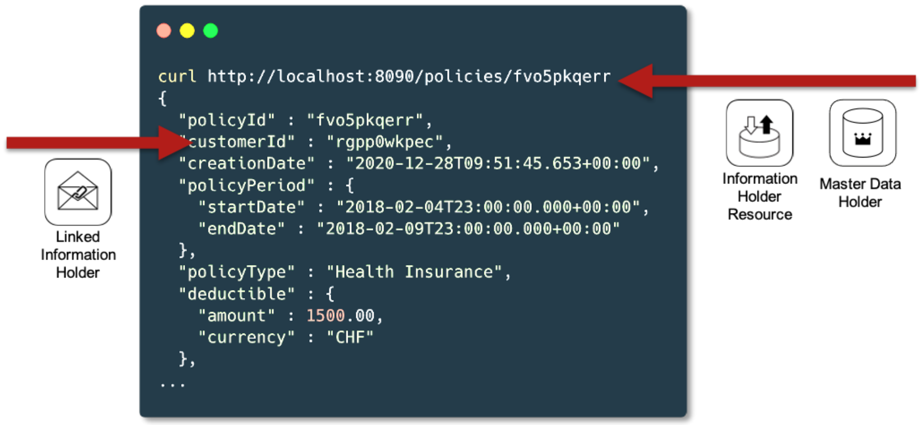

Fig. 13. The endpoint returns a representation of the requested policy. The response contains a reference to a `customerId` Linked Information Holder. See www.microservice-api-patterns.org for more information on the patterns and icons.

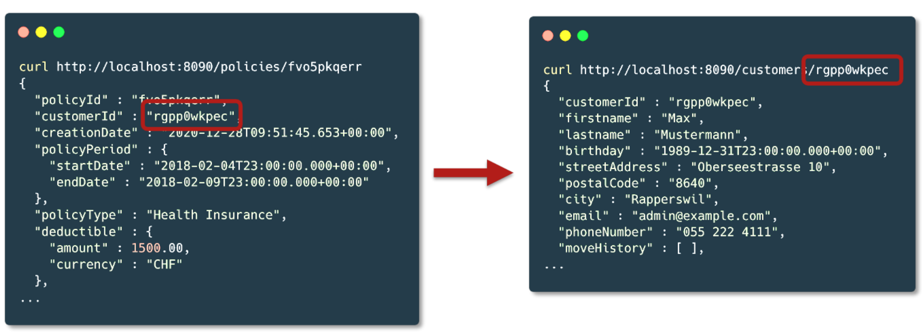

Fig. 14. The customer data can be retrieved by using the `customerId` in a follow-up request to its Master Data Holder endpoint.

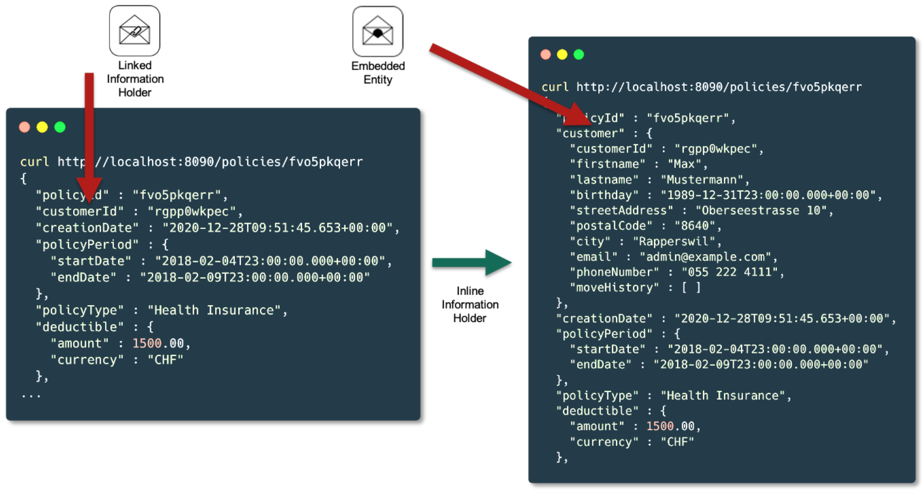

Fig. 15. Applying the Inline Information Holder refactoring, the Linked Information Holder can be replaced with an Embedded Entity.

offers an HTTP endpoint to read insurance policy data, as shown in Figure 13. After the API has been put into production, a performance analysis makes evident that the majority of clients first requests data from the policy endpoint, but then also retrieves the corresponding master data from the customer endpoint (see Figure 14). By inlining the Information Holder Resource and refactoring to the Embedded Entity pattern, the linked data is now available inside the initial response message, saving clients the additional request and avoiding underfetching. Figure 15 shows the resulting response structure. While the consequences of applying this refactoring are beneficial for some clients, those not requiring the additional data now face increased message sizes and transfer times. The processing and database retrieval effort in the endpoint increases as well. Alternative patterns that the API can be refactored to are Wish List and Wish Template [32]. These two patterns let the client inform the provider about its data interests.

Draft Catalog of Candidate Refactorings. The survey results unveil a few quality concerns that can be addressed with API refactorings, for instance in the area of performance. Also taking our previous work [17] into account and reflecting on our own industry projects and action research, we have collected the following list of candidate refactorings for further elaboration[11]:

- *Inline Information Holder:* Reduce indirection by directly embedding referenced information holder resources. Figure 16 details this refactoring in a table format adapted from [27].

[11] At https://interface-refactoring.github.io we have started to publish these refactorings; names and content of the refactoring are subject to change.

- *Extract Information Holder:* Decrease message size by replacing embedded entities with linked information holders.
- *Introduce Payload Feature Toggle:* Lessen the mismatch between client expectations and provider capabilities by letting the client decide on the response message content.
- *Introduce Pagination:* Reduce response data set size by dividing large data sets into chunks.
- *Collapse Operations:* Enhance discoverability by reducing the number of distinct operations.
- *Move Operation:* Improve cohesion by shifting an operation from one endpoint to another.
- *Rename Endpoint:* Increase developer usability by making the name of an endpoint more expressive and accurate.
- *Split Endpoint:* Move one or more operations to a new endpoint so that coupling between the operations in an endpoint is reduced.
- *Merge Endpoints:* Collapse two endpoints that are tightly coupled.

Note that we articulate the candidate refactorings in the API design vocabulary we established in previous work [32]. This wording differs substantially form that used in the survey reported on in the Sections 3 and 4 of this paper. We avoided our own pattern terminology as much as possible in the survey to avoid or reduce bias; more specifically, we did not want to steer participants towards desired responses.

When answering research Question 2 in the survey in Section 4, usability, performance, scalability and maintainability were reported as the most common causes that led to API changes (see Figure 8). Inline Information Holder and Rename Endpoint can improve usability, while Extract Information Holder aims at improving performance and scalability. Introduce Payload Feature Toggle and Introduce Pagination also aim at improving these two qualities, but can have a negative impact on maintainability. The remaining candidates address high cohesion and low coupling, general API design principles that did not come out of the survey (it is worth noting that we did not ask about them explicitly). See Figure 17 for a mapping of causes of API changes to suggested refactorings.

6 Discussion of Preliminary Results and Action Plan

Pros and Cons. API refactoring is a new (some might say yet another) practice. It picks up another, commonly applied one to reduce the learning effort. A related risk is that a different intuitive understanding, that of code refactoring, might cause undesired irritation. The community might argue about the criteria for inclusion/exclusion of refactorings in the catalog: does it claim to be complete? To mitigate this effect we started from the survey results and applied the "rule of three" from the patterns community[12]. Just like code refactoring

[12] Only include a pattern if there are at least three known uses. Not to be confused with the "rule of three" of refactoring [6] duplicated pieces of code, where it refers to the number of duplicates starting from which a refactoring is recommended.

Refactoring	Inline Information Holder
Motivation	Reduce indirection by directly embedding referenced information holder resources.
Smell	Clients have to issue follow-up requests to get the information they need.
Preconditions	The resource to be inlined should be part of the same service implementing the endpoint. Otherwise, the refactoring might introduce undesired dependencies between services.
Steps	1. Add additional attributes to the data transfer object (DTO; code first) or models (design/contract first). 2. From the implementation of the the linked information holder endpoint, copy the code to retrieve the additional entity or value. 3. Paste the code into the endpoint implementation and add the entity/value to the DTO. 4. Remove the link to the inlined resource. 5. Run tests to observe the changed responses; adjust tests to the new response structure. 6. Clean up the implementation code if necessary (observe the *Rule of Three* of refactoring [7]), for example, by moving duplicated code to a common implementation. 7. If under your control, adjust API clients to remove the unneeded API calls. 8. Adjust API description, sample code, tutorials, etc. where needed.
Evolution Strategy	If the refactoring only adds additional information to the response message, the refactoring is backwards compatible. Otherwise, consider applying a *MAP Evolution* pattern: *Experimental Preview*, *Semantic Versioning*, *Two in Production* or *Limited Lifetime Guarantee*.
Inverse	Extract Information Holder

Fig. 16. The Inline Information Holder refactoring reduces indirection by directly embedding an entity instead of indirectly referencing it with a link.

Causes for API Changes	Suggested API Refactoring
Clients were missing some features or capabilities in the API (client-driven change)	N/A
The provider (you) wanted to make new features available in the API (provider-driven change)	N/A
Performance issues caused by too many parallel requests by the same or different API clients.	Extract Information Holder
Performance issues caused by requests for very large datasets.	Introduce Payload Feature Toggle, Introduce Pagination
Clients had to issue too many requests to get the data they needed.	Inline Information Holder
The implementation did not adhere to the specification.	Move Operation, Rename Endpoint
The API offered too many ways to achieve the same outcome.	Merge Endpoint, Collapse Operations
The style, patterns or idioms of the API (for example, naming or data structures) were inconsistent.	Split, Merge or Rename Endpoint
Architectural changes in the API implementation (for example, changes in composition or responsibilities of backend services)	Split, Merge or Rename Endpoint and Operations
Technological change (for example, changing from RESTful HTTP to GraphQL)	N/A (Architectural Refactoring)

Fig. 17. Suggested refactorings for different causes of API changes. Causes for which refactoring is not the right technique – i.e., those where features are added – are marked with N/A. For technological changes, an architectural refactoring might be applicable.

requires unit testing, API refactoring requires automated API testing so that it can be practiced safely. Tests should ensure that an API refactoring does not accidentally change the request- or response message structures, endpoints or other API elements. Such tests likely have to cover multiple API calls that jointly deliver a particular integration functionality or API feature.

Threats to Validity (of Research Results). When designing the questionnaire, conducting the survey, and analyzing the results, we applied general guidelines for survey design [9]. Several threats to the validity of our results remain. The questionnaire was distributed by e-mail and social media, which is not representative of the overall population. While we did state in the invitation to the survey and also in the introduction of the questionnaire that we target "software architects, engineers and technical product owners that specify, implement and maintain message-based remote APIs", we cannot rule out that some participants did not fulfill these requirements. Moreover, targeting a particular industry or domain could lead to different responses, especially regarding technologies used. Nevertheless, the results show that an API refactoring practice would be valuable to practitioners.

Action Plan. The next step in this research are a) gather even more and deeper practitioner knowledge by continuing the analysis, b) describe all refactorings already identified and backed by the survey results in a template similar to that behind Figure 16, c) implement tool support for selected ones in editors for API description languages (motivated by the survey results in Figures 3 and 4), and d) validate the usability and usefulness of refactoring catalog and tools empirically (via action research, case studies, and controlled experiments).

The website interface-refactoring.github.io features our emerging API refactoring catalog (i.e., the results of action plan item b).

7 Summary and Outlook

Refactorings offer step-by-step instructions on how to evolve a software system systematically. Combining the agile practice of refactoring with API patterns can help to adapt (micro-)service APIs to changing client demands and quality requirements. Our practitioner survey on API change and evolution motivated the potential use of such a refactoring practice. Taking additional feedback into account, we plan to complete and extend our emerging refactoring catalog and have started to implement tool support for selected high-value refactorings.

A related open research problem is how architectural principles, both general ones such as loose coupling and interface-specific ones such as POINT[13] relate to refactoring. On the one hand, frequent refactoring of continuously evolving architectures can be seen as a principle in itself. On the other hand, violations of other principles may yield API smells suggesting refactorings as well, similar to code/design smells [23]; for instance, if two API operations violate the *I*solated principle in POINT, it might make sense to merge them into one.

[13] www.ozimmer.ch/practices/2021/03/05/POINTPrinciplesForAPIDesign.html

Another idea for future work is that of *smart proxies* that mediate between old and new versions of an API in case backward compatibility cannot be assured. Finally, the relation between API refactoring and API testing has to be studied and strengthened.

Acknowledgments

We would like to thank all reviewers and participants of the survey. The work of Mirko Stocker and Olaf Zimmermann is supported by the Hasler Foundation through grant nr. 21004.

References

1. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: Migration to a cloud-native architecture. IEEE Software **33**(3), 42–52 (2016)
2. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, Boston (2004)
3. Cossette, B.E., Walker, R.J.: Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. FSE '12, Association for Computing Machinery, New York, NY, USA (2012)
4. Dig, D., Johnson, R.: The role of refactorings in api evolution. In: 21st IEEE International Conference on Software Maintenance (ICSM'05). pp. 389–398 (2005)
5. Dig, D., Johnson, R.: How do apis evolve? a story of refactoring: Research articles. J. Softw. Maint. Evol. **18**(2), 83–107 (Mar 2006)
6. Fowler, M.: Refactoring. Addison-Wesley Signature Series (Fowler), Addison-Wesley, Boston, MA, 2 edn. (2018)
7. Hunt, A., Thomas, D.: The Pragmatic programmer : from journeyman to master. Addison-Wesley, Boston [etc.] (2000)
8. Jamshidi, P., Pahl, C., Mendonca, N.C., Lewis, J., Tilkov, S.: Microservices: The journey so far and challenges ahead. IEEE Software **35**(03), 24–35 (may 2018)
9. Kasunic, M.: Designing an Effective Survey. Software Engineering Institute (9 2005)
10. Kerievsky, J.: Refactoring to Patterns. Pearson Higher Education (2004)
11. Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. FSE '12, Association for Computing Machinery, New York, NY, USA (2012)
12. Kula, R.G., Ouni, A., German, D.M., Inoue, K.: An empirical study on the impact of refactoring activities on evolving client-used apis. Information and Software Technology **93**, 186–199 (2018)
13. Kwon, Y.W., Tilevich, E.: Cloud refactoring: Automated transitioning to cloud-based services. Automated Software Engg. **21**(3), 345–372 (Sep 2014)
14. Li, J., Xiong, Y., Liu, X., Zhang, L.: How does web service api evolution affect clients? In: 2013 IEEE 20th International Conference on Web Services. pp. 300–307 (2013)
15. Lübke, D., Zimmermann, O., Stocker, M., Pautasso, C., Zdun, U.: Interface evolution patterns - balancing compatibility and extensibility across service life cycles. In: Proceedings of the 24th EuroPLoP conference. EuroPLoP '19 (2019)

16. Menascé, D.A.: Qos issues in web services. IEEE internet computing **6**(6), 72–75 (2002)
17. Neri, D., Soldani, J., Zimmermann, O., Brogi, A.: Design principles, architectural smells and refactorings for microservices: a multivocal review. SICS Softw.-Intensive Cyber Phys. Syst. **35**(1), 3–15 (2020)
18. Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N.M.: Microservices in practice, part 2: Service integration and sustainability. IEEE Software **34**(2), 97–104 (2017)
19. Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J.C., Canali, L., Percannella, G.: Rest apis: A large-scale analysis of compliance with principles and best practices. In: Web Engineering. pp. 21–39. Springer International Publishing, Cham (2016)
20. Rosenberg, F., Celikovic, P., Michlmayr, A., Leitner, P., Dustdar, S.: An end-to-end approach for qos-aware service composition. In: IEEE Int. Conf. on Enterprise Distributed Object Computing Conference (EDOC'09). pp. 151–160. IEEE (2009)
21. Stal, M.: Agile Software Architecture. Morgan Kaufmann (12 2013)
22. Stocker, M., Zimmermann, O., Lübke, D., Zdun, U., Pautasso, C.: Interface quality patterns - communicating and improving the quality of microservices APIs. In: Proceedings of the 23nd EuroPLoP conference. EuroPLoP '18 (2018)
23. Suryanarayana, G., Sharma, T., Samarthyam, G.: Software process versus design quality: Tug of war? IEEE Software **32**(4), 7–11 (2015)
24. Wang, S., Keivanloo, I., Zou, Y.: How do developers react to restful api evolution? In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) Service-Oriented Computing. pp. 245–259. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
25. Zdun, U., Stocker, M., Zimmermann, O., Pautasso, C., Lübke, D.: Guiding architectural decision making on quality aspects in microservice APIs. In: 16th International Conference on Service-Oriented Computing ICSOC 2018. pp. 78–89 (November 2018)
26. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: Proceedings of the 12th international conference on World Wide Web. pp. 411–421. ACM (2003)
27. Zimmermann, O.: Architectural refactoring: A task-centric view on software evolution. IEEE Software **32**(2), 26–29 (Mar-Apr 2015)
28. Zimmermann, O.: Architectural refactoring for the cloud: Decision-centric view on cloud migration. Computing **99**(2), 129–145 (2017)
29. Zimmermann, O.: Microservices tenets. Comput. Sci. **32**(3-4), 301–310 (Jul 2017)
30. Zimmermann, O., Lübke, D., Zdun, U., Pautasso, C., Stocker, M.: Interface responsibility patterns: Processing resources and operation responsibilities. In: Proceedings of the 25th EuroPLoP conference. EuroPLoP '20 (2020)
31. Zimmermann, O., Pautasso, Cesare Lübke, D., Zdun, U., , Stocker, M.: Data-oriented interface responsibility patterns: Types of information holder resources. In: Proceedings of the 25th EuroPLoP conference. EuroPLoP '20 (2020)
32. Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., Zdun, U.: 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). vol. 78, pp. 4:1–4:17 (2020)
33. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U.: Interface representation patterns: Crafting and consuming message-based remote APIs. In: Proc. of the 22nd EuroPLop. pp. 27:1–27:36. EuroPLoP '17, ACM (2017)