Empirical Software Engineering (2021) 26: 104
https://doi.org/10.1007/s10664-021-09999-9

Industry practices and challenges for the evolvability assurance of microservices

An interview study and systematic grey literature review

Justus Bogner[1] · **Jonas Fritzsch**[1] · **Stefan Wagner**[1] · **Alfred Zimmermann**[2]

Accepted: 8 June 2021 / Published online: 22 July 2021

Abstract

Context Microservices as a lightweight and decentralized architectural style with fine-grained services promise several beneficial characteristics for sustainable long-term software evolution. Success stories from early adopters like Netflix, Amazon, or Spotify have demonstrated that it is possible to achieve a high degree of flexibility and evolvability with these systems. However, the described advantageous characteristics offer no concrete guidance and little is known about evolvability assurance processes for microservices in industry as well as challenges in this area. Insights into the current state of practice are a very important prerequisite for relevant research in this field.

Objective We therefore wanted to explore how practitioners structure the evolvability assurance processes for microservices, what tools, metrics, and patterns they use, and what challenges they perceive for the evolvability of their systems.

Method We first conducted 17 semi-structured interviews and discussed 14 different microservice-based systems and their assurance processes with software professionals from 10 companies. Afterwards, we performed a systematic grey literature review (GLR) and used the created interview coding system to analyze 295 practitioner online resources.

Results The combined analysis revealed the importance of finding a sensible balance between decentralization and standardization. Guidelines like architectural principles were seen as valuable to ensure a base consistency for evolvability and specialized test automation was a prevalent theme. Source code quality was the primary target for the usage of tools and metrics for our interview participants, while testing tools and productivity metrics were the focus of our GLR resources. In both studies, practitioners did not mention architectural or service-oriented tools and metrics, even though the most crucial challenges like *Service Cutting* or *Microservices Integration* were of an architectural nature.

Communicated by: Arpad Beszedes and Miryung Kim

This article belongs to the Topical Collection: *Software Maintenance and Evolution (ICSME)*

✉ Justus Bogner
justus.bogner@iste.uni-stuttgart.de

Extended author information available on the last page of the article.

Conclusions Practitioners relied on guidelines, standardization, or patterns like *Event-Driven Messaging* to partially address some reported evolvability challenges. However, specialized techniques, tools, and metrics are needed to support industry with the continuous evaluation of service granularity and dependencies. Future microservices research in the areas of maintenance, evolution, and technical debt should take our findings and the reported industry sentiments into account.

Keywords Microservices · Evolvability · Assurance · Industry · Interviews · Grey literature review

1 Introduction

Fast moving markets and the age of digitalization require that software can be quickly adapted or extended with new features. If change implementations frequently happen under time pressure, the sustainable evolution of a long-living software system can be significantly hindered by the intentional or unintentional accrual of technical debt (Lehman 1980; Avgeriou et al. 2016). The quality attribute associated with software evolution is referred to as *evolvability* (Rowe et al. 1998): the degree of effectiveness and efficiency with which a system can be adapted or extended. Software evolution can therefore be seen as a subset of maintenance since the latter also includes changes where the requirements are stable, like the fixing of bugs. For Rajlich (2018), evolvability is therefore more demanding and *requires* maintainability, i.e. an evolvable system is always maintainable but not vice versa. In this sense, it is impossible to evolve unmaintainable software, yet maintaining software that can no longer be evolved may still be possible. Evolvability is especially important for software with frequently changing requirements, e.g. internet-based systems.

To provide sufficient confidence that such a system can be sustainably evolved, software professionals apply a set of numerous activities that we refer to as *evolvability assurance*. These activities are usually either of an *analytical* or *constructive* nature (Wagner 2013). The goal of analytical activities is to identify evolvability-related issues in the system, i.e. to evaluate or quantify evolvability. This includes manual techniques like code review or scenario-based analysis but also tool-supported static or dynamic analysis with e.g. metrics. The goal of constructive activities, on the other hand, is the remediation of identified issues or systematic evolvability construction for some part of the system, i.e. to improve evolvability. The primary constructive activity is code-level or architectural refactoring. However, adhering to evolvability-related principles and guidelines or using evolvability-related design patterns during software evolution is also a constructive – and proactive – form of evolvability assurance. Lastly, some practices like conscious technical debt management cover both areas: the identification and documentation of technical debt items is analytical, while the removal of prioritized items is constructive. For larger systems, all these activities often form a communicated assurance process and are an important part of the development workflow with integration into the continuous integration and delivery (CI/CD) pipeline.

Microservices constitute an important architectural style that prioritizes evolvability (Newman 2015). A key idea here is that fine-grained and loosely coupled services that are independently deployable should be easy to change and to replace. Consequently, one of the postulated microservices characteristics is *evolutionary design* (Fowler 2019). While these properties provide a beneficial theoretical basis for evolvable systems, they offer no concrete and universally applicable solutions. As with each architectural style, the implementation of a concrete microservice-based system can be of arbitrary quality. Especially

the "service cutting" activity has been described as challenging and several approaches have been proposed by academia to support it (Fritzsch et al. 2019b). Apart from this, very little scientific research has covered the areas of maintenance, evolution, or technical debt for microservices. Examples include the tracking and management of microservices dependencies (Esparrachiari et al. 2018), antipatterns for microservices (Taibi et al. 2020), and the applicability of service-based maintainability metrics for microservices (Bogner et al. 2017).

In addition to this sparse scientific state of the art, there are also very few empirical studies on the industry state of practice. Little is known about what evolvability assurance processes and techniques companies use for microservices or if these are different compared to other architectural styles. In the general area of service-based systems, Schermann et al. (2016) even describe a mismatch between what academia assumes and what industry actually does. An analysis of industry practices in this regard could identify common challenges, showcase successful processes, and highlight gaps and deficiencies. This would also provide insights into how industry perceives academic approaches specifically designed for service orientation, e.g. service-oriented maintainability metrics. Results of such a study could help to design new and more suited evolvability assurance processes or techniques.

We therefore conducted an analysis of the industry state of practice based on two qualitative studies. First, we interviewed 17 Germany-based software professionals from 10 different companies (see Section 4). They described 14 different systems with various microservices characteristics and their concrete evolvability assurance process including tool, metric, and pattern usage. We also talked with them about the evolution qualities of microservices, how microservices influence the assurance process, and their perceived challenges for evolvability.

Using these identified concepts, we subsequently conducted a systematic grey literature review (GLR) to confirm and expand our interview results based on a large number of practitioner blog posts, conference presentations, white papers, and Q&A forum posts (see Section 5). In total, we analyzed 295 relevant internet resources written by software professionals using our existing interview coding system, which we extended with newly identified labels.

In this paper, we present the results of these two qualitative studies and discuss their combined implications (see Section 6). The interview results related to the evolvability assurance of microservices have already been discussed on their own in (Bogner et al. 2019a). Since the extensive interviews also covered additional topics, there are also publications on used technologies, the adherence to microservice characteristics, and overall software quality (Bogner et al. 2019b) as well as on intentions, strategies, and challenges for migrating to microservices (Fritzsch et al. 2019a). This article therefore extends our previous work (Bogner et al. 2019a) with a grey literature review and the joint interpretation of interview and GLR results.

2 Related Work

Several empirical studies that report challenges for microservices adoption have been published so far. Baškarada et al. (2018) conducted 19 in-depth interviews with experienced architects. They discussed opportunities and challenges associated with the adoption and implementation of microservices. Four types of corporate systems with different levels of suitability for microservices were identified. Especially large corporate systems of record

like enterprise resource planning (ERP) systems were not seen as appropriate targets. In this context, organizational challenges such as DevOps methodologies would be less serious for information and communication technology enterprises than for traditional organizations, where IT was perceived as a "necessary evil" (Baškarada et al. 2018).

Ghofrani and Lübke (2018) conducted a similar empirical survey among 25 practitioners that were mainly developers and architects. Their objective was to find perceived challenges in designing, developing, and maintaining microservice-based systems. The results reveal a lack of notations, methods, and frameworks for architecting microservices. Several participants named the distributed architecture as responsible for the challenging development and debugging of the system. Participants generally prioritized optimizations in security, response time, and performance over aspects like resilience, reliability, and fault tolerance.

In their interviews with 10 microservices experts from industry, Haselböck et al. (2018) focused on design areas of microservices and associated challenges. The study identified 20 design areas and their importance as rated by the participants. Design principles and common challenges from earlier mapping studies could be confirmed by the authors. Similar to our qualitative study, interviewees' rationales are discussed as well. Microservices design is a fundamental aspect of their evolvability later on. As such, the study can be seen as a valuable contribution to the topic at hand.

Some studies also focus on evolvability-related areas like technical debt and antipatterns. Carrasco et al. (2018) conducted a literature review to gather migration and architecture smells. The authors derived best practices, success stories, and pitfalls. By digesting 58 different sources from academia and grey literature, they presented nine common bad smells with proposed solutions.

Similarly, Taibi et al. (2020) synthesized a taxonomy of 20 microservices antipatterns via an extensive mixed-method study over several years, combining an industrial survey, a literature review, and interviews. Their most recent replication relied on interviews with 27 experienced developers, who shared bad microservices practices that they encountered and their applied solutions. The taxonomy contains both organizational and technical antipatterns.

To sum up the frequently studied area of microservices antipatterns, Neri et al. (2019) conducted a multivocal review on the refactoring of antipatterns which violate microservices principles. In addition to scientific publications, they also included grey literature from practitioners, such as blog posts, industrial white papers, or books. As results, they present 16 refactorings for seven architectural microservices smells.

In a previous study (Bogner et al. 2018), we surveyed 60 software professionals via an online questionnaire to assess maintainability assurance practices in industry as well as notable differences with both service- and microservice-based systems. We asked questions related to the used processes, tools, and metrics to learn about treatments specific to such systems. Very few participants reported the usage of techniques to address existing issues related to architecture-level evolvability. Since 67% of participants neglected service-oriented particularities in the assurance, the study revealed a weak spot in industry practice that may impair the lifespan of service-based systems.

A pure grey literature review in the area of microservices was conducted by Soldani et al. (2018). To distill commonly perceived technical and operational advantages ("gains") and drawbacks ("pains") of microservices, they selected and analyzed 51 practitioner resources obtained via search engines like Google, Bing, or Duck Duck Go. The analyzed white papers, blog posts, and videos were published between 2014 and 2017 and were mapped to the general areas of microservices design, development, and operations.

Bandeira et al. (2019) investigated how microservices were discussed on StackOverflow. They argue that Q&A websites can serve as representative samples of the community. With StackOverflow being the biggest platform for software development, they refer to several other studies that already leveraged this source of accumulated knowledge. Unlike our approach, they did not use keyword search, but only retrieved questions for which the author used the *microservices* tag. In total, they applied mining techniques and topic modelling to 1,043 discussions and extracted technical and conceptual subjects. While there is a slight overlap with codes we used in our GLR, evolvability is not a thematic priority in their very general classification scheme.

Lastly, Lenarduzzi and Taibi (2018) investigated technical debt interest by means of a long-term case study where they monitored the migration of a monolithic legacy system to microservices. The study aimed to characterize technical debt and its growth comparatively in both architectural styles. As a preliminary result, they found that the total amount of technical debt grew much faster in the microservice-based system.

In summary, the majority of microservices industry studies focuses on general challenges or antipatterns and not on a broader set of applied evolvability assurance activities. Our survey (Bogner et al. 2018) is one of the few to do so, but we did not focus exclusively on microservices and could not report much on developers' rationales due to the limitations of the quantitative questionnaire. To address this gap, we therefore conducted two qualitative microservices studies to analyze industry evolvability assurance processes, techniques, and challenges.

3 Research Design

We generally followed the five-step case study process as described by Runeson and Höst (2009) to structure our research, which includes the consecutive steps of *study design*, *preparation for data collection*, *evidence collection*, *data analysis*, and *reporting*. As a first step, we defined a research objective and related research questions. The primary goal of this study can be formulated in the following way:

*Analyze the applied **evolvability assurance***
for the purpose of knowledge generation
with respect to common practices and challenges
from the viewpoint of software professionals
*in the context of **microservices** in industry*

We formulated three research questions to set more fine-grained directions for our study and to limit its scope:

RQ1: What processes do software professionals follow for the evolvability assurance of microservices and for what reasons?

This RQ covers the process aspects of evolvability assurance, i.e. what concrete activities are used and how are they structured and combined.

RQ2: What tools, metrics, and patterns do software professionals use for assuring the evolvability of microservices and with what rationales?

This RQ is concerned with three concrete concepts which are frequently used within evolvability assurance activities, namely tools and metrics to quantify evolvability or to identify evolvability-related issues as well as design and architecture patterns for systematic evolvability construction.

RQ3: How do software professionals perceive the quality of their microservices and assurance processes and what parts do they see as challenging?

This RQ analyzes software professionals' perceptions with respect to the quality characteristics of their systems (e.g. what evolvability characteristics are they satisfied with) and their assurance activities (e.g. do they think that their activities are effective and efficient). Special emphasis is put on perceived challenges for the evolvability of microservices.

Since quantitative survey research with questionnaires would not be in-depth enough to cover practitioners' rationales – an issue we also experienced during the analysis of our survey data (Bogner et al. 2018) – we selected a predominantly qualitative research approach. Qualitative methods analyze relationships between concepts and directly deal with identified complexity (Seaman 2008). Results from such methods are therefore very rich and informative and may provide insights into the thought process behind the analyzed information. As concrete methods for surveying the state of practice, we chose **semi-structured interviews** (Seaman 2008; Hove and Anda 2005) and a **systematic grey literature review** (GLR) (Garousi et al. 2019), i.e. we collected and analyzed a large number of practitioner online resources related to our research objective. The detailed research designs for each method are described in the following two subsections.

3.1 Practitioner Interviews

Semi-structured interviews left us with a basic agenda, but also allowed us to dynamically adapt our questions based on responses. For our interview participants, we defined the following requirements:

- Significant professional experience (minimum of five years) and solid knowledge of service orientation
- Technical role (e.g. developer or architect) that at least sometimes writes code
- Recent participation in the development of a system with microservices characteristics

We recruited participants via personal industry contacts of the research group and by attending industry meet-up groups on microservices, where we approached companies from different domains and of different size. To ensure a base degree of heterogeneity within our population, we only allowed a maximum of three participants per company and if two participants worked on the same system, they needed to have different roles.

3.1.1 Preparation for Data Collection

Before conducting the interviews, we created several documents. We prepared an *interview preamble* (Runeson and Höst 2009) that explained the interview process and relevant topics. To make participants familiar with the study, they received this document beforehand. The preamble also outlined ethical considerations like confidentiality, requested consent for audio recordings, and guaranteed that recordings and transcripts would not be published. As a second document, we created an *interview guide* (Seaman 2008) that contained the most important questions grouped in thematic blocks. This guide helped us to scope and organize the semi-structured interviews and was used as a loose structure during the sessions. We did not share it with interviewees beforehand. Furthermore, we created a slide set with additional interview artifacts for certain topics or questions, such as an exemplary list of assurance tools. These slides were also not shared with participants before the interviews. For the analysis, we created a preliminary set of *coding labels* and a *case characterization matrix* (Seaman 2008) containing the most important case attributes.

3.1.2 Evidence Collection

In total, we conducted 17 individual interviews (no group interviews). Six of these were performed face to face and 11 via remote communication software with screen sharing. All interviews except for a single English one were conducted in German. Each participant agreed to a recording of the interviews, which took between 45 and 75 minutes. We loosely followed the structure of the interview guide and adapted based on how a participant reacted. As an initial ice breaker, participants were asked to describe their role and the system they worked on. Later on, the topic shifted to the evolvability of the system and potential symptoms of technical debt.

The next thematic block was the concrete evolvability assurance process for the system. We presented a custom maturity model with four levels that ranged from *implicit and basic* to *explicit and systematic* (see Fig. 1). We created this simplistic model inspired by existing frameworks like CMMI (Software Engineering Institute 2010) or the maintenance maturity model from April et al. (2005). Since its only purpose was to act as an initial conversation opener about evolvability assurance, our model has not been evaluated in any way. To get participants to reflect on their own assurance, we explained the different levels and examples of associated practices. Participants were then asked to place themselves on the level that corresponded the most to their current assurance activities and to give some rationales for their choice. From there, we discussed the details of their processes and concrete techniques, tools, and metrics. Even though we discussed specific level placements in some cases, it was neither the intention to reach perfectly consistent maturity levels across all cases nor to rigorously assess the maturity of the different companies. The model just served as an icebreaker to talk about evolvability assurance. Additionally, the level placements give an indication for how elaborate and systematic the interviewees perceived their own practices.

Lastly, we asked questions about challenges and participants' satisfaction with the current process. The satisfaction and reflection questions relied on a five point scale from -2 (very negative) to +2 (very positive) with 0 being the neutral center. After the interviews, we manually transcribed each audio recording to create a textual document. We then sent these documents to participants for review and final approval. During this review, interviewees were able to delete sensitive paragraphs or change statements of unclear or unintended meaning. The approved transcripts were then used for detailed qualitative content analysis.

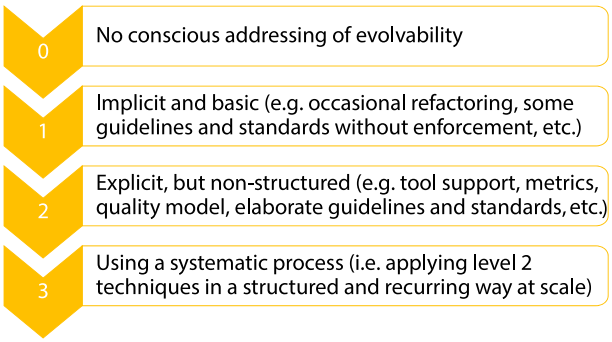

Fig. 1 Evolvability Assurance Maturity Levels

3.1.3 Data Analysis

As a first step for analyzing the interview data, we performed the coding of each transcript. Using the created preliminary set of codes, we assigned labels to relevant paragraphs. During this process, several new labels were created and already finished transcripts were revisited. Labels were also renamed, split, or merged as we acquired a more holistic understanding of the cases. These coding activities followed the *constant comparison method* that is based on grounded theory (Seaman 2008). After the coding of all transcripts, we analyzed the details and code relationships of each individual transcript. This activity resulted in a textual description for every case[1].

In the second step, we applied *cross-case analysis* (Seaman 2008) to identify important generalizations and summaries between the cases. We used the coding system and the created case characterization matrix as well as *tabulation* (Runeson and Höst 2009). For each research question, important findings were extracted from the transcript and documented. During this process, we also refined the case characterization matrix. General trends and deviations were documented and later aggregated into results and take-aways. To increase transparency and reproducibility, we published all interview documents and artifacts (except for the full transcripts) as well as the results of the analysis on both GitHub[2] and Zenodo[3].

3.2 Grey Literature Review (GLR)

Since software engineering is a very practitioner-oriented field and microservices still have limited scientific publication coverage in a lot of areas, grey literature may hold valuable insights that academic literature simply cannot provide yet (Garousi et al. 2016). Therefore, multivocal and grey literature reviews in software engineering experienced a rise in popularity over the last years, even though the usage of such methods is still at an early stage and clear guidelines are only starting to emerge (Neto et al. 2019). In our research design, we used the identified concepts and results from our interview as the basis for planning our GLR, i.e. the goal of the GLR was to confirm (or reevaluate) the interview results on the foundation of a large sample size of documents.

3.2.1 Preparation for Data Collection

We developed a detailed review protocol based on many of the guidelines proposed in (Garousi et al. 2019) to support us during the process (see also Fig. 2). With respect to data sources (see also Fig. 3), we decided to include the search engines Google and Bing, as they are the two most popular ones[4] and have been used in most GLRs in the field of software engineering. Additionally, we included the Q&A platform StackOverflow and the three specialized StackExchange communities Software Engineering, Software Quality Assurance & Testing, and DevOps. StackExchange communities are popular with practitioners and especially valuable to identify frequently experienced issues and challenges.

Based on our experiences with the interviews, we defined a set of seven search strings for Google and Bing (see Fig. 4). Each included the term `microservices` combined with

[1] https://github.com/xJREB/research-microservices-evolvability-interviews/tree/master/case-descriptions
[2] https://github.com/xJREB/research-microservices-evolvability-interviews
[3] https://doi.org/10.5281/zenodo.2586916
[4] https://www.reliablesoft.net/top-10-search-engines-in-the-world

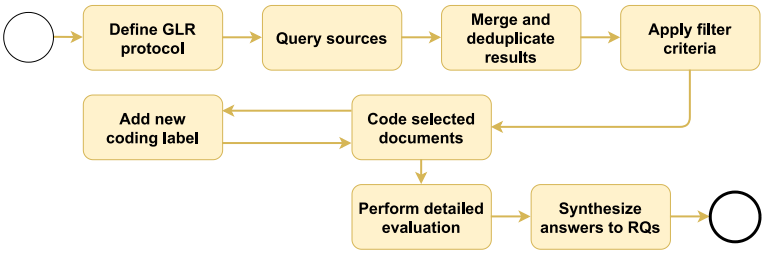

Fig. 2 General GLR Process

various relevant concepts like quality attributes (e.g. `evolvability`) or means of assurance (e.g. `metrics` and `tools`). Additionally, we excluded domains that only produced unwanted academic results like `researchgate.net` or `scholar.google.com`[5]. For StackOverflow and the StackExchange communities, we constructed a detailed query for the offered SQL interfaces[6]. This SQL query[7] relied on the same search terms, but only included posts created in 2014 or later that had a score of at least +2. To limit the long runtime and large number of results, the required score was raised to +5 for the StackOverflow query.

3.2.2 Evidence Collection

We manually entered all seven search strings into both Google and Bing and extracted the first 100 URLs per search string, i.e. a total of 1,400 URLs ($7 * 100 * 2$). We used an anonymous browsing session and set the search location to the United States. As Google did not reliably respect domain exclusions of its own domains and Bing did not offer a feature for this at all, we identified any excluded domains via regular expression search and filled up the list with additional search hits to guarantee 100 results per search string. Concerning the four Q&A platforms, we simply extracted all posts returned by the respective queries. We then merged the results (1,730 resources) and eliminated duplicates (485 resources, 28%), which left us with a total of 1,245 URLs that we needed to manually assess for inclusion.

This assessment was based on the following criteria. First, we only included textual practitioner online resources in English. Most frequently, these were resources like blog posts, news or wiki articles, Q&A posts, tutorials (in written form), company white papers, or presentation slides. Resources were excluded if they were not in English or if they were scientific papers, books, videos, job offerings, or announcements of conferences, seminars, and trainings. On top of these basic criteria, filtering out resources not relevant for our research questions was most important. To be included, a resource needed to contain a description or reflection of some form of evolvability assurance or some form of systematic evolvability construction. This could be the usage of tools, metrics, or patterns, but also descriptions of guidelines, best practices, or lessons learned as well as experienced challenges. A last possibility for exclusion was a perceived suboptimal quality of the resource, e.g. if the author's or company's experience or authority was questionable. It was important that the author had

[5] https://github.com/xJREB/research-microservices-evolvability-glr/blob/master/protocol.md#google-and-bing-search

[6] https://data.stackexchange.com/stackoverflow/query/new

[7] https://github.com/xJREB/research-microservices-evolvability-glr/blob/master/protocol.md#stackexchange-sql-query

- Google: `https://www.google.com/advanced_search`
- Bing: `https://www.bing.com`
- StackOverflow: `https://stackoverflow.com`
- Software Engineering: `https://softwareengineering.stackexchange.com`
- Software Quality Assurance & Testing: `https://sqa.stackexchange.com`
- DevOps: `https://devops.stackexchange.com`

Fig. 3 Used Search Engines and Q&A Platforms

applied or experienced the described assurance techniques in the real world and that they were not simply a hypothetical suggestion the author thought about. For Q&A posts, we did not follow this as sternly, since we also wanted to record the experienced challenges.

To be able to split the manual work of filtering over 1,200 resources without significantly reducing consistency, the first two authors performed several rounds of inter-rater reliability calibration by both filtering the same set of resources and comparing the results. We did this for four consecutive rounds, each with 100 mixed resources from all sources. After each round, differences were discussed and resolved, which gradually led to more consensus. For the last 100 resources, the percentage agreement was 87% and Cohen's Kappa (Cohen 1960) was 0.602, which Landis and Koch (1977) categorize as right at the end of "moderate" and the beginning of "substantial" agreement. At this point, we split the remaining 845 resources between us and filtered independently. If one rater was unsure about the inclusion of a resource, he assigned a review of his preliminary decision to the other rater. Differences of opinion were discussed until a consensus was reached. Using this process, we finally ended up with 295 included resources that needed to be analyzed in detail (see also Fig. 5).

3.2.3 Data Analysis

To analyze the selected GLR resources, we relied on roughly the same coding process as with the interviews. We used the existing coding system as the basis and iteratively extended it with new labels we discovered during the process. A difference to the interview transcripts was that we did not code concrete text passages, but assigned labels to the complete resource. For longer resources, we also documented the occurrence of the label to support data extraction later on, e.g. a page number or associated heading. Similar to the filtering stage, we also wanted to split the work between two coders and therefore performed an initial calibration round where both coders analyzed the first 20 resources and discussed any differences. Due to the very elaborate coding system (over 130 unique labels in six categories), it did not make sense to calculate agreement measures like Cohen's Kappa since

```
- microservices ∧ (maintainability ∨ evolvability ∨ modifiability)
- microservices ∧ (maintenance ∨ evolution)
- microservices ∧ "quality assurance"
- microservices ∧ metrics ∧ (maintainability ∨ evolvability ∨
  modifiability)
- microservices ∧ patterns ∧ (maintainability ∨ evolvability ∨
  modifiability)
- microservices ∧ tools ∧ (maintainability ∨ evolvability ∨
  modifiability)
- microservices ∧ (challenges ∨ guidelines ∨ "best practices")
```

Fig. 4 Used Search Terms for Google and Bing

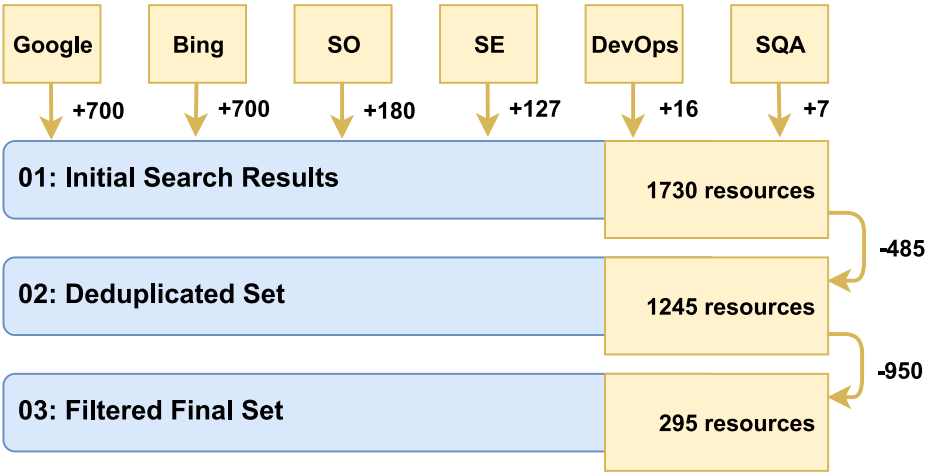

Fig. 5 GLR Stages with # of Resources at Each Stage (SO: StackOverflow, SE: Software Engineering, DevOps: DevOps, SQA: Software Quality Assurance & Testing)

high values are unlikely to be reached, even if the majority of labels will be the same per resource. After the calibration discussion, however, we felt that we had been sufficiently consistent to nonetheless split up the remaining resources. Additionally, if a coder was unsure about one of his resources, he could assign it to the other coder for double-checking. After coding was completed, we synthesized answers to our research questions by analyzing frequently occurring labels and their related text passages. Similar to the interview analysis, we looked for general tendencies and documented interesting quotes. Later on, these extractions were aggregated into results and take-aways. Likewise, all GLR artifacts and results are published on both GitHub[8] and Zenodo[9].

4 Interview Results

Our interviewees were from 10 different companies (C1–C10) of different sizes and domains (see Table 1). Half of these were software & IT services companies that mostly developed systems for external customers. The companies from other domains always had an internal system owner. Every participant was located in Germany, even though some companies had sites in several European countries or even globally. From our 17 participants (P1–P17), 11 stated architect as their role while four were developers. The remaining two roles were data engineer and DevOps engineer. All participants possessed a minimum of five years of professional experience, with a median of 12 and a mean of 14.7 years. Altogether, we discussed 14 systems (S1–S14) and their evolvability assurance processes, where in three cases, two participants talked about the same system (S5, S9, S11).

For the sake of brevity, this publication only contains the aggregated interview results. We provide a detailed description of every case in our online repositories[2,3]. The descriptions include general information about the system, the details of the evolvability assurance

[8] https://github.com/xJREB/research-microservices-evolvability-glr
[9] https://doi.org/10.5281/zenodo.3731259

Table 1 Interview Demographics: Companies and Participants (CID: Company ID, SID: System ID, PID: Participant ID, Exp.: Professional Experience in Years

CID	Domain	Employees	SID	PID	Role	Exp.
C1	Financial Services	1–25	S1	P1	Developer	6
C2	Software & IT Services	>100,000	S2	P2	Lead Architect	30
			S3	P3	Architect	24
			S4	P4	Architect	30
C3	Software & IT Services	26–100	S5	P5	Architect	20
				P6	Lead Developer	8
C4	Software & IT Services	101–1,000	S6	P7	Architect	9
			S7	P8	Architect	17
C5	Software & IT Services	>100,000	S8	P9	Lead Developer	7
C6	Tourism & Travel	1,001–5,000	S9	P10	Developer	9
				P11	Data Engineer	7
			S10	P12	Architect	12
C7	Logistics & Public Transport	101–1,000	S11	P13	DevOps Engineer	5
			S11a	P14	Architect	17
C8	Retail	5,001–10,000	S12	P15	Lead Architect	9
C9	Software & IT Services	101–1,000	S13	P16	Architect	18
C10	Retail	1,001–5,000	S14	P17	Architect	22

process, and lastly reflections and challenges per system. Table 2 provides some basic system information and the self-assessed assurance maturity levels while Table 3 lists the usage of tools, metrics, and patterns to analyze and improve evolvability. By analyzing and comparing the individual cases, we identified several trends or common relationships. These generalizations, summaries, or notable deviations from common assumptions are presented in the following subsections that correspond to our three research questions.

4.1 Assurance Processes (RQ1)

The intention of RQ1 was to find out what general activities participants employed to assure the evolvability of microservices, how systematically they organized these activities, and what participants' rationales for these decisions were. Since one microservices characteristic is the decentralization of control and management, we also wanted to analyze how much central governance was applied.

4.1.1 Decentralization vs. Governance

In general, every analyzed assurance process had some degree of explicit and conscious addressing of evolvability, even though the sophistication and extensiveness of the applied techniques varied greatly. When looking at the larger systems, there were **two different approaches** for assuring evolvability: very decentralized with very autonomous teams (e.g. S9, S10, S12, S14) vs. centralized governance for macroarchitecture, technologies, and assurance combined with a varying degree of team autonomy for microarchitecture (e.g. S2, S3, S4, S7, S13). The latter kind was usually applied for systems that were built for external customers and that exhibited some project characteristics.

In the decentralized variant, the internal system was managed in a continuous product development mode, which created quality awareness by making people responsible and simultaneously empowering them. This variant is more in line with the microservices and

Table 2 Interview Systems: General Characteristics

ID	System Purpose	Age in Years	# of Services	Assurance Maturity Level (0-3)
S1	Derivatives management system (banking)	1.5	9	2
S2	Freeway toll management system	2	10	3
S3	Automotive problem management system	1	10	1.5
S4	Public transport sales system	2	∼100	1.5
S5	Business analytics & data integration system	1.5	6	P5: 1
				P6: 1.5
S6	Automotive configuration management system	1	60	3
S7	Retail online shop	2.5	∼250	2.5
S8	IT service monitoring platform	3	9	2
S9	Hotel search engine	2	∼10	P10: 1.5
				P11: 3
S10	Hotel management suite	1.5	20	2
S11	Public transport management suite			
	(S11a: human resource management part)	2.5	10 products	P13: 1.5
				P14: 2.5
S12	Retail online shop	2	∼45	2.5
S13	Automotive end-user services mgmt. system	2	7	2
S14	Retail online shop	6	∼175	2

DevOps principle "you build it, you run it". Techniques in this variant also were by no means basic or implicit. Even though teams were allowed to choose their own assurance activities, they usually created a more or less structured processes that did not depend on external governance. This was hard to replicate for IT service providers that often did not operate the systems themselves (S2, S3, S4, S7, S13) and had to coordinate with external customers or even other contractors (S4, S7). Therefore, they relied more on central governance. Architect P4 described it as follows: "In our case, the main challenge is to convince 300 people to move in the same direction. For that, we created a very large number of guidelines and rules for service creation."

Such **guidelines, principles, or standardizations** were nonetheless seen as important parts of the process in both variants. These coding labels were among the most frequent ones. Nearly all participants reported their usage in various areas such as architectural principles, rules for service communication, skeleton projects, style guides, cross-cutting concerns like logging or authentication, candidate technologies, or Docker images. The degree of enforcement varied between companies and was usually higher within the centralized variant. In the decentralized variant, pragmatism was often more important than

Table 3 Interview System Assurance Processes: Tools, Metrics, and Patterns

ID	Tools	Metrics	Patterns
S1	IDE linting	–	Event-Driven Messaging, Service Registry
S2	SonarQube (FindBugs, Checkstyle, PMD)	Test coverage, cyclomatic complexity, clone coverage, # of defects per service, # of failed tests, # of code smells, # of endangered requirements	Event-Driven Messaging
S3	SonarQube (FindBugs, Checkstyle, PMD)	Test coverage, clone coverage, defect resolution time	Event-Driven Messaging, Strangler
S4	SonarQube (FindBugs), VersionEye	# of code smells, test coverage, # of outdated dependencies	Event-Driven Messaging, Backends for Frontends, Consumer-Driven Contracts, Tolerant Reader
S5	SonarQube (FindBugs, Checkstyle, PMD)	Test coverage	Event-Driven Messaging
S6	SonarQube (FindBugs, Checkstyle)	Test coverage, # of failed tests, # of code smells, cyclomatic complexity, clone coverage, LOC	Event-Driven Messaging
S7	SonarQube (FindBugs, PMD, Checkstyle), Cobertura, IDE linting, custom static analyzer for coding conventions	Test coverage, cyclomatic complexity, clone coverage, # of rule violations, velocity	API Gateway
S8	SonarQube (FindBugs), IDE linting	Cognitive complexity, # of code smells, test coverage	Event-Driven Messaging
S9	P10: Checkstyle, IDE linting P11: SonarQube (FindBugs, Checkstyle)	P10: Defect resolution time, burndown P11: # of code smells, # of rule violations	Event-Driven Messaging, Request-Reaction
S10	SonarQube, IDE linting	Test coverage, # of endangered usage scenarios	Self-Contained Systems, Backends for Frontends
S11	P13: SonarQube (FindBugs, Checkstyle) P14: FindBugs, PMD, Cobertura, custom tool for architectural conformance	P13: # of code smells P14: # of architectural violations, # of rule violations	–
S12	SonarQube (FindBugs, Checkstyle, PMD), IDE linting, Codecov, Cobertura	Test coverage, cyclomatic complexity, # of code smells, # of rule violations	Event-Driven Messaging, Consumer-Driven Contracts
S13	SonarQube (FindBugs, Checkstyle, PMD)	Test coverage, # of code smells, # of rule violations	Event-Driven Messaging
S14	SonarQube (FindBugs, Checkstyle, PMD), Structure101, Codecov, Cobertura, Codacy	Test coverage, CI/CD pipeline duration, LOC	Event-Driven Messaging, Self-Contained Systems, Event Sourcing

the strict adherence to rules and several participants (P5, P8, P10, P12, P15, P17) reported simplicity as a key principle ("KISS" ⇒ "keep it simple, stupid").

4.1.2 Automated and Manual Activities

To make assurance activities more efficient and objective, all participants saw **automation and tool support** as useful, albeit with varying enthusiasm. Several participants reported the integration of quality analysis tools into the CI/CD pipeline (P2, P3, P8, P9, P11, P14, P15, P17). This was often combined with quality gates, i.e. automated source code checks that could prevent merging or deployment. For architect P17, the pipeline's execution time was very important: only tools that were absolutely necessary should therefore be integrated. Additionally, several participants advocated for a sensible usage of quality gates. Data engineer P11 was frustrated with how difficult it would be to get a passing merge request due to the strict rules. Similarly, developer P10 mentioned that strict quality gates could hinder the deployment of important production bug fixes. Architect P7's team did not use any quality gates because of continuous experimentation and prototyping. Lastly, lead architect P2's team circumvented some of these issues by applying quality gates only for releases and not for merge or pull requests.

Nearly all participants agreed that test automation was an important part for the assurance process of microservices. While unit tests were very common, several participants also reported automated end-to-end tests for the integration of microservices and stressed their importance (P2, P3, P7, P12, P16). Some teams also had more elaborate strategies that linked tests to requirements (S2) or usage scenarios (S10). Participants with only unit tests (P5, P6, P9) or barely any tests (P10) also saw the importance to bring their test automation to a higher level.

Despite the reported importance of automation and tool support, several participants also highlighted the usefulness of **manual assurance activities**. Code reviews were seen as an important practice to increase code quality and to share knowledge within the team (P1, P3, P4, P5, P7, P8, P10, P11). Pair programming was used for the same reasons by two participants and the downside of additional man-hours was willingly accepted (P8, P15). Lastly, refactoring was highly valued and some participants also explicitly mentioned the use of the "boy scout" rule during feature implementations (P11, P15), i.e. the principle to always leave changed code cleaner than before. Activities like these would efficiently increase code quality over time.

4.1.3 Documentation Practices

Even though some participants were proponents of concise documentation or architectural decision records within the source code repository (P10, P11), several systems relied on more elaborate **architecture and service documentation** in a system like Confluence or SharePoint (S1, S4, S5, S6, S8). Common types of documentation were system architecture, service dependencies and contracts between teams, service functionality and API descriptions, reference architectures and service blueprints, design rationales, or architectural principles and guidelines. For IT service providers, parts of this documentation was also used to communicate with the customer. Lastly, only P7 and P14 reported the conscious tracking of identified technical debt items for later debt management. Architect P7's teams held an explicit meeting every two weeks, where the most important technical debt items were discussed and their prioritization was decided.

4.2 Tools, Metrics, and Patterns (RQ2)

Our second research question targeted the application of and rationale for tools, metrics, and design patterns. Automation and tool support is an often cited microservices characteristic and seen as necessary to manage a large number of small components. We were also interested if participants used tools and metrics specifically designed for service orientation. Lastly, we wanted to explore the usage of design patterns for evolvability construction, since there is a large body of patterns for service-oriented architecture (SOA) and more recently also for microservices.

4.2.1 Tools Related to Evolvability Assurance

While the usage of over a dozen different **tools for evolvability assurance** was reported, 14 of 17 participants named SonarQube as a central tool that was usually integrated into the CI/CD pipeline. Since P1 and P10 planned to introduce it soon, architect P14 remained the only participant that would not use SonarQube in the foreseeable future. Reported reasons for its popularity were the OpenSource license, the easy installation, plugin availability, and configurability. In Java-focused systems, SonarQube was often extended with tools like FindBugs, Checkstyle, and PMD. Additionally, specialized tools for test coverage like Cobertura (P8, P14, P15, P17), Codecov (P15, P17), or Codacy (P17) were used. For a basic degree of local and immediate quality assurance, IDE linting via e.g. TSLint, ESLint, and PHPLint was reported by some participants (P1, P8, P10, P12, P15).

4.2.2 Evolvability-Related Metrics

With respect to **metrics**, 10 of the 17 participants reported the usage of test coverage, even though some perceived this metrics as less important than others and were very aware of possible quality differences with automated tests. Architect P12 termed it as follows: “Even I could fake the coverage for two classes you give me in like five minutes. You can write a test that brings coverage to about 60%, but actually it covers like 2%.” Some participants also focused on additional metrics for testing and functional correctness like # of failed tests over time (P2, P7), # of defects per service (P2), or # of endangered requirements (P2) or usage scenarios (P12). Most SonarQube users also payed attention to standard findings like code smells, code duplication, and cognitive or cyclomatic complexity. Participants with rule-based tools like FindBugs, Checkstyle, or other linters used the number of rule violations as a simple metric that had to be zero.

Overall, most applied metrics were focused on source code quality, even though their effectiveness for the whole system was seen as controversial by a few participants (P8, P17). Architect P17 described it as follows: “Most of these metrics relate to a single project, which is very useful when I have a monolith with a million LOC. However, if I have a service with 1000 LOC which code base is separated from all other 150 microservices, most of these metrics lose their importance.” With respect to productivity metrics, some interviewees reported the usage of defect resolution time (P3, P10), velocity (P8), sprint burndown (P10), or deployment duration (P17). These were important for them to control and manage software evolution.

While architecture-related topics like microservices dependencies were very prevalent during our interviews, participants generally did not apply **architecture-level tools and metrics**. Architect P14’s team used a custom tool for architectural conformance checking in the monolithic code base for a sub product of S11 and architect P17 reported the intermittent

usage of Structure101 for a larger subsystem that also consisted of one code base. Apart from that, tools or metrics were exclusively focused on code quality with a local view for a single service. No automatic or semi-automatic efforts were mentioned to evaluate the architecture of a microservice-based system.

Likewise, no participant reported the usage of a tool or metric specifically designed for service orientation. When we explicitly asked about service-oriented metrics like the coupling of a service, the cohesion of a service interface, or the number of operations in a service interface, several participants indicated that these sounded interesting and useful (P1, P5, P6, P7, P8, P15). Some interviewees also noted that the underlying principles of these metrics were important guidelines in the architecture and design phase (P7, P8, P10, P15, P17). They tried to manually respect these principles during e.g. service cutting, even though they currently had no concrete measurements in place to validate them.

Another common theme in this area was the **healthy and non-patronizing usage of tools and metrics**, which should be respected when developing microservices in decentralized and autonomous teams. As already mentioned, several participants voiced reservations against test coverage (P10, P12, P15). Architect P14 also warned that a strict metric focus would pose the danger that people optimized for measurements instead of fixing the underlying problems. Moreover, lead architect P15 perceived it as difficult to interpret measurements of a single service without a point of reference or a system-wide average. Architect P17 advocated for a sparse usage of tools, because too many metrics could not be analyzed by developers and their collection could slow down the deployment pipeline. Only tools that would support the analysis of current problems should be kept. Lastly, architects P8 and P14 highlighted the agile principle of "individuals and interactions over processes and tools": the usage of tools and metrics should support developers in their daily work and not be a frustrating and alienating experience for them.

4.2.3 Service-Based Patterns for Evolvability

We also analyzed the usage of **service-oriented design patterns** as conscious means to increase evolvability. In general, we did not find a widespread usage of them. Most common was *Event-Driven Messaging* that was partially applied in 11 of the 14 cases. While several participants stated that the pattern was used to decouple services, another intention was to implement reliable asynchronous and long-running communication. The pattern was sometimes paired with *Request-Reaction* (P10, P16). Apart from messaging, most participants applied activity patterns like *Service Refactoring*, *Service Decomposition*, and *Service Normalization*. In line with the philosophy of evolutionary design, microservices were frequently split and merged.

Other patterns were used sporadically. P12 and P17 applied the *Self-Contained System* paradigm to achieve vertical isolation between subsystems. In a migration context, P3 and P16 reported the usage of the *Strangler* pattern to extend an existing monolith with new microservices until its final replacement. To place an intermediary between service consumers and producers, P4 and P12 implemented the *Backends for Frontends* pattern that would also prevent too many concurrent long-running HTTP requests. Similarly, P8 chose the *API Gateway* pattern which also brought benefits for security. The patterns *Consumer-Driven Contracts* (P4, P15) and *Tolerant Reader* (P4) were applied to make service interface evolution more robust and to prepare consumers for future changes. Lastly, developer P1 was the only participant to explicitly report the usage of the *Service Registry*

pattern for dynamic service discovery, even though some participants may have used similar functionality via Kubernetes.

4.3 Evolvability Reflections and Challenges (RQ3)

With RQ3, we wanted to analyze participants' perception of the general evolution qualities of their microservice-based systems as well as their satisfaction with their current assurance processes (see Fig. 6). We also tried to summarize what participants experienced as the most important challenges for the evolvability of their microservices (see Fig. 7).

4.3.1 Perceived System Evolvability

In general, our interviewees perceived the **evolvability of their microservices** as positive (mean: +0.88, median: +1), especially in cases with a migration context where a monolith had been rewritten. Only two participants chose a negative rating (-1). Architect P4 saw the high degree of technological heterogeneity and the very different service granularity as threatening for the large project, especially once S4 would be handed over to the smaller maintenance team. Data engineer P11 described the chosen service cuts as inefficient and politically motivated and worried about significant issues with the consistency of the data model as well as the inaccessibility of code due to distributed repositories.

As for more specific quality attributes, the analyzability of individual services would be much improved (P1, P8, P10, P16, P17), even though grasping and understanding the whole system would be difficult (P7, P8, P11, P17). When compared to most monoliths, the modularity of microservices would make it very convenient to change or add functionality (P1, P3, P6, P9, P12, P17) and would also allow to efficiently scale-out the development with multiple teams (P7, P16). Even though reuse is usually a theme more common in SOA, several participants reported a positive reusability of their microservices (P1, P3, P7, P8, P10, P17). To reduce coupling, some participants avoided the sharing of non-open source libraries between services via duplication (P7, P10, P15, P17). Others tried to consciously increase reuse via shared libraries (P3, P9) or by slightly generalizing service interfaces (P1). Lastly, participants reported that individual services would be easy to test (P3, P7, P10, P13, P15) and to replace (P2, P15, P17).

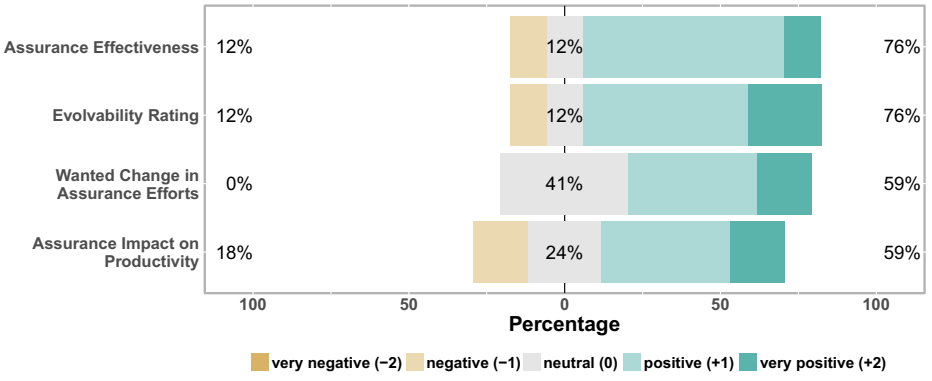

Fig. 6 Aggregated Evolvability Assurance Reflections of 17 Participants

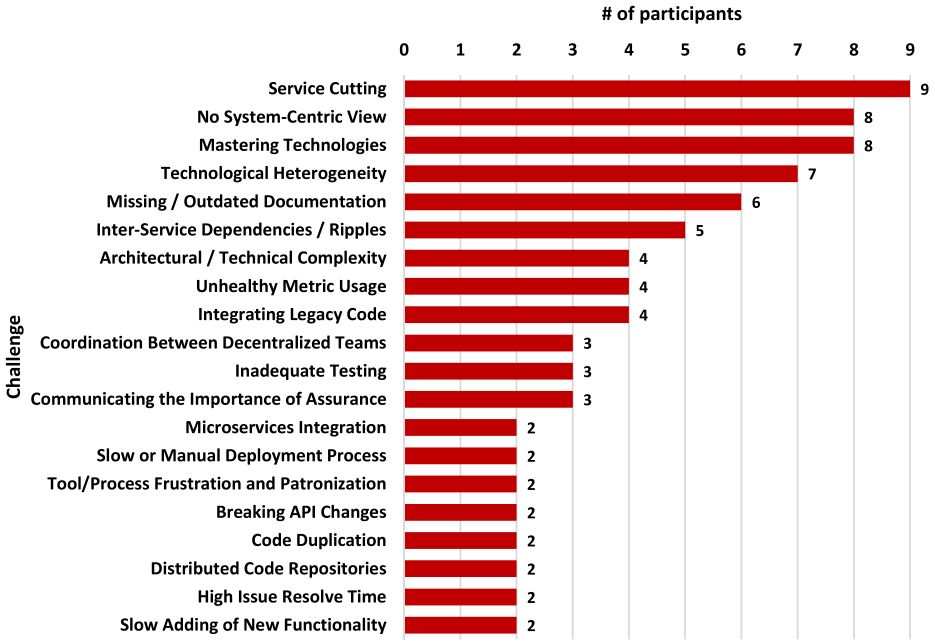

Fig. 7 Evolvability Challenges With At Least 2 Mentions from 17 Participants

4.3.2 Evolvability Challenges

Since most systems were fairly young or even still in the process of being migrated, individual services were usually of a good quality. Basic symptoms of technical debt or bad code quality were rarely seen as an issue, especially since a single service would be easy to replace. However, **problems related to architecture and the data model** were reported as serious threats for long-term evolvability (P3, P7, P11, P13, P15). This was sometimes exacerbated because coordination between autonomous teams would be difficult (P4, P10, P11, P15). Moreover, finding the appropriate service granularity was a prevalent theme and service cutting was by far named as the most challenging activity that was also associated with frequent refactoring (P2, P3, P4, P6, P7, P9, P11, P12, P15). Harmful inter-service dependencies sometimes led to ripple effects on changes (P3, P5, P9, P11, P15), which made adding or changing functionality slower and more error-prone. Breaking API changes caused similar effects for service consumers (P1) or automated tests (P2). Participants did not use any tools to support service decomposition or metrics to evaluate the quality of the chosen cuts, e.g. via coupling or cohesion. Lead architect P2 described it as follows: "In my opinion, there are no useful tools to split up a monolith. It's always a very difficult manual activity. You can use something like Domain-Driven Design, but that's just a methodology which doesn't give you a concrete solution."

Participants were divided when it came to **technological heterogeneity**. In very decentralized environments, it was generally perceived as overall beneficial (P10, P15, P17), as it would allow choosing the best solution for problems at hand, broaden developers' experience and skills, and make a company a more attractive employer. Other participants perceived it as potentially dangerous and wished for a more sensible handling of technology hypes (P3, P4, P12, P16). Similarly, the mix of legacy and modern service technology would

sometimes pose additional problems (P2, P11, P13, P14), like in the case of S9 where additional tooling was necessary to integrate legacy PHP components. Most participants also noted that significant efforts had to be spent on mastering new microservices and DevOps technologies and it would be problematic to find skilled developers (P1, P3, P5, P6, P9, P10, P12, P13, P16). Overall, participants were very aware of the human factors of evolvability and sometimes even saw them as more challenging as technical ones. Knowledge exchange between teams was therefore a high priority for some interviewees (P10, P13, P15).

Concerning participants' **reflection of their assurance processes** (see also Fig. 6), most saw the effectiveness of their assurance activities (mean: +0.76, median: +1) as well as overall impact on productivity (mean: +0.59, median: +1) as positive. Only three interviewees (P9, P11, P12) reported that activities would hinder development efficiency (-1) and would sometimes slow down feature development. Moreover, participants generally wanted to invest more effort for the assurance (mean: +0.76, median: +1) and try out new techniques or metrics. No one reported the wish to reduce efforts.

4.3.3 Influence of Microservices on the Assurance Process

However, the influence of microservices on the assurance process was seen as controversial. While testing a single service would be easy, integration testing would be more complex because of an additional layer (P2, P3, P13). This would be especially critical if microservices were developed in independence for a long time and integrated at a later stage. Furthermore, root cause analysis of issues would be more complex in such a highly distributed system (P3, P11). A very commonly named concern was that keeping a system-centric quality view and assessing the macroarchitecture would be much more difficult (P4, P6, P7, P8, P10, P11, P15, P17), which architect P8 described as follows: "I'd say we are pretty good when it comes to assuring the evolvability of single services. However, we have a lot of catching up to do for everything that crosses product or service boundaries." Distributed code repositories and autonomous teams would make the access to code as well as static analysis more complicated. It would also be hard to compare metrics between services and relate them to system-wide averages (P8, P11, P17).

Nonetheless, participants also named positive factors. Small services would not only be easy to replace, people would also be much more motivated to fix a small number of issues for a project (P15, P17), which lead architect P15 described as follows: "In a monolith with 100.000 FindBugs warnings, you are completely demotivated to even fix a single one of those. In a microservice with 100 warnings, you just get to work and remove them." If adopted correctly, microservices would also bring a cultural change with respect to quality awareness and responsibility (P10, P15, P17). Architect P17 highlighted the importance of continuous product development in this regard: "If you work in a project mode, evolvability assurance usually annoys you, because you have short-term goals and want to finish the project. In a product mode, the team knows that they sabotage their system's evolvability in the long run, if they take too many short-cuts." Lastly, lead architect P15 noted that while they were relatively satisfied with their current evolvability assurance activities, they did not really invest much efforts into researching and designing a fitting evolvability assurance strategy for the future. Finding out which approaches, tools, and metrics worked best for their microservices could be a vital advantage in the long-term.

5 GLR Results

From our 295 included resources, 96 were from Bing, 78 from Google, 30 were discovered by both Bing and Google, 2 by both Google and StackExchange, and 89 were exclusively from the StackExchange communities (SO: 32, SE: 51, SQA: 4, DevOps: 2). Therefore, roughly one third of our resources were Q&A posts. From the six categories of our coding system[10], the most frequently used one was *Challenges*: 218 of 295 resources (74%) included at least one label from this category. Second and third most popular categories were *Process* (66%) and *Patterns* (48%). 78 resources were assigned at least one label from *Influence on Assurance* (26%), while 16% of resources mentioned *Tools* and only 6% *Metrics*. Similar as with the interviews, we also present the GLR results in three subsections that correspond to our research questions.

5.1 Assurance Processes (RQ1)

Our first RQ was concerned with the general processes for evolvability assurance and the applied activities.

5.1.1 Test Automation

The most frequently mentioned process activity was **test automation** (94 resources, 32%), which was described as an essential prerequisite for sustainable microservice development and evolution. In general, unit tests were still seen as necessary but not sufficient for microservices. Instead, the majority of resources called for an extension of the classical test pyramid: "The test pyramid was conceived during the era of the monolith and makes a lot of sense when we think about testing such applications. For testing distributed systems, I find this approach to be not just antiquated but also insufficient."[11] Practitioners therefore advocated for an extensive usage of integration, contract, and end-to-end tests, or as André Schaffer from Spotify put it: "A more fitting way of structuring our tests for Microservices would be the Testing Honeycomb. That means we should focus on Integration Tests, have a few Implementation Detail Tests and even fewer Integrated Tests (ideally none)."[12] These integration or end-2-end tests were often enabled by partly mocking or spawning involved components. As an alternative to this, some resources mentioned *QA in Production* (8), i.e. running tests within the production environment to have the most realistic conditions. Lastly, a few resources also described the usage of chaos and load testing or applying practices like test-driven development (TDD) or behavior-driven development (BDD).

5.1.2 Decentralization vs. Governance

Similar as with the interviews, we also found the two antagonistic forces of **decentralization & empowerment** and **governance & standardization**, which both had an influence on the assurance process. In general, there was more tendency towards decentralization (37 resources with positive mentions) than towards standardization (28) since this was seen as important to guarantee independent and fast service evolution. Vinay Sahni, the founder of

[10] https://github.com/xJREB/research-microservices-evolvability-glr/blob/master/coding-labels.md
[11] https://medium.com/@copyconstruct/testing-microservices-the-sane-way-9bb31d158c16
[12] https://labs.spotify.com/2018/01/11/testing-of-microservices

Enchant, advocated to maximize the autonomy of teams so that they would have to coordinate less and would be more productive.[13] WSO2 described a similar philosophy in their microservices white paper: "With respect to development lifecycle management, microservices are built as fully independent and decoupled services with a variety of technologies and platforms."[14] Lastly, Armağan Amcalar, head of software engineering at unu GmbH, also remarked in an interview with InfoQ's Ben Linders that this decentralization was not only beneficial for service evolution but also desired by developers: "Teams want autonomy and ownership. They don't want to be bound by the decisions of others, and they don't want to feel obliged to justify themselves and their decisions towards others."[15]

On the other hand, many resources recommended basic standardization and governance to avoid chaos, especially for infrastructure, cross-cutting concerns, or service communication, but not for domain-related functionality. Among these were also several proponents of decentralization like Vinay Sahni ("Provide flexibility without compromising consistency: Give teams the freedom to do what's right for their services, but have a set of standardized building blocks to keep things sane in the long run."[13]) and WSO2 ("Run-time governance aspects, such as SLAs, throttling, monitoring, common security requirements, and service discovery, are not implemented at each microservice level. Rather, they are realized at a dedicated component, often at the API-gateway level."[14]). This sensible balance between decentralization and autonomy on the one hand and governance and standardization on the other hand was also described as "*Goldilocks Governance*" by Neal Ford, a director at ThoughtWorks[16]. In general, a lot of practitioners were very aware of this trade-off: "Choose wisely what you leave out of your macro-architecture. For every choice you allow the individual development teams to make, you must be willing to live with differing decisions, implementations, and operational behaviors."[17]

5.1.3 Principles and Guidelines

A very important part of the mentioned standardization were **principles or guidelines** (66 resources, 22%), which were mostly defined for service design and inter-service communication. The most popular principle (20) was to limit or completely avoid the sharing of databases or database tables between microservices. Advocated reasons were to encapsulate information and to avoid coupling: clear data ownership should guarantee independence between services. A similar "don't share!" principle was related to domain-specific libraries (18). Having the same shared library within several microservices would lead to upgrade and deployment dependencies. Most authors saw redundancy as preferable to coupling in such cases ("lesser evil"). A softer version of this principle we encountered was to allow sharing libraries between services managed by the same team. In nearly all cases, the shared usage of open source libraries for non-domain related functionality was not seen as harmful. Another popular guideline was to reduce or avoid direct synchronous (RESTful) calls between services (14), since this would lead to harmful coupling. Some authors also wrote that true service independence would only be possible with *Event-Driven Messaging*. Other mentioned principles were favoring pragmatism and simplicity (12) over complex solutions,

[13] https://www.vinaysahni.com/best-practices-for-building-a-microservice-architecture
[14] https://wso2.com/whitepapers/microservices-in-practice-key-architectural-concepts-of-an-msa
[15] https://www.infoq.com/news/2018/11/human-side-microservices
[16] http://nealford.com/downloads/Evolutionary_Architecture_Keynote_by_Neal_Ford.pdf
[17] https://www.freecodecamp.org/news/microservices-from-idea-to-starting-line-ae5317a6ff02

avoiding breaking API changes (8), using (semantic) versioning (4) in the spirit of evolvable providers and adaptable consumers, the SOLID[18] design principles (8), and the 12-Factor App Guidelines (4).

5.1.4 Other Activities

Lastly, the following process-related activities and techniques were mentioned by a few resources: the deliberate refactoring of services (merging and splitting) to improve evolvability (16); architecture documentation (15), which was seen as especially important for service interfaces; code reviews (6) to improve quality and share knowledge; and feature toggles (5) to enable a more flexible service evolution within continuous integration practices.

5.2 Tools, Metrics, and Patterns (RQ2)

Our second RQ targeted the usage of concrete tools, metrics, and patterns related to evolvability assurance.

5.2.1 Tools Related to Evolvability Assurance

We identified 46 resources (16%) that mentioned the usage of **tools for the evolvability assurance process** (see Fig. 8). The overwhelming majority were related to automated testing (38 of 47 of unique reported tools). Among these, tools for integration and contract testing were especially popular. Most frequently mentioned tools here were Pact (15), SoapUI (5), Spring Cloud Contract (5), and Postman (4). In the important area of consumer-driven contract testing, the company Testsigma advocated the usage of two tools: "There are several tools for contract testing, but the most reliable and effective ones are Spring Cloud Contract and Pact."[19] For the mocking and stubbing of services, WireMock (7) was the most popular tool, even though a lot of other tools were mentioned as well, e.g. Mountebank (2), Restito (2), Hoverfly (1), or REST-driver (1). Another area of interest was UI and end-to-end testing. Here, Selenium (8) was the most trusted tool: "Selenium is considered an industry standard in automating web applications for testing purposes. Thanks to Selenoid, Selenium Hub successor, multiple Selenium servers can be run with many different browser versions encapsulated in Docker containers."[20] An alternative framework was Cypress (2), which was e.g. chosen by Zalando for its fast performance and avoidance of non-deterministic tests.[21] Less represented were tools to capture and replay HTTP traffic for realistic and repeatable tests like VCR (4), GoReplay (2), or Betamax (1); resilience and chaos testing like Netflix's Simian Army (5) or the Java tool Byteman (1); or load testing with e.g. JMeter (1).

Besides this variety of testing tools, there was just a small number of tools for other assurance-related activities. Only three resources reported the usage of SonarQube with integration into a CI/CD pipeline with quality gates. Other used static analysis tools were Kiuwan (1), X-Ray (1), NDepend (1), JDepend (1), Code City (1), and Source Monitor (1).

[18] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[19] https://testsigma.com/blog/testing-microservices-challenges-and-strategies-testsigma

[20] https://codilime.com/quality-assurance-trends-for-2020

[21] https://jobs.zalando.com/en/tech/blog/end-to-end-microservices

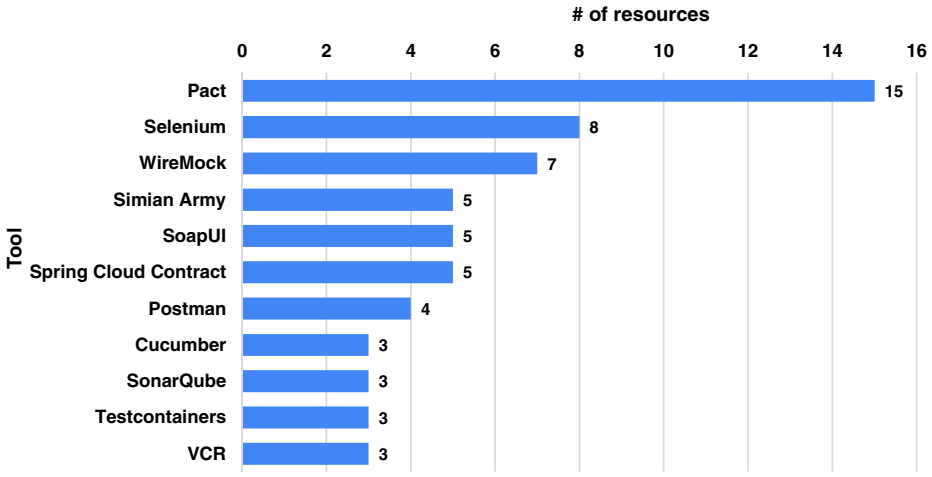

Fig. 8 Evolvability Tools With At Least 3 Mentions in 295 GLR Resources

Overall, we found very little explicit reports of such tools in our resources (only 9 mentions in 295 resources).

5.2.2 Evolvability-Related Metrics

With respect to **evolvability-related metrics** (see Fig. 9), we identified 27 unique metrics in only 19 resources (6%). This makes metrics the least represented label category. Even though 38 of 47 identified tools were related to automated testing, only seven resources reported the usage of test coverage as a metric. While this still makes it the most mentioned metric, some authors like Chris Richardson saw deficits with it, even though he still recommended its usage: "While test coverage is not the best metric, it should also be enforced by the deployment pipeline." [22] Other metrics related to correctness and reliability were deployment success rate (2), # of defects per service (1), or # of failed tests (1).

Similar to the small number of tools for static analysis, we also identified just a few metrics for architecture and source code quality (12 metrics with 13 mentions). Apart from lines of code (2), each of these metrics were only reported once, e.g. # of classes, clone coverage, cognitive complexity, or cyclomatic complexity. Among these 12 metrics, even less were related to architecture or service orientation. Rare examples were # of dependencies, component entanglement, or static coupling.

Lastly, the most frequently used metrics were related to productivity, often in the context of the CI/CD pipeline (19 mentions of 8 metrics). Popular examples were cycle time (5), # of deploys to production (4), deployment duration (3), or mean time to repair (MTTR) (3). For Chris Richardson, such metrics were a very important instrument to track and improve the practices for "*rapid, frequent and reliable software delivery*".[23] Proponents of such metrics advocated that teams should have some measure of productivity or stability in place to

[22] https://chrisrichardson.net/post/antipatterns/2019/04/09/antipattern-flying-before-walking.html
[23] https://www.slideshare.net/chris.e.richardson/melbourne-jan-2019-microservices-adoption-antipatterns-obstacles-to-decomposing-for-testability-and-deployability

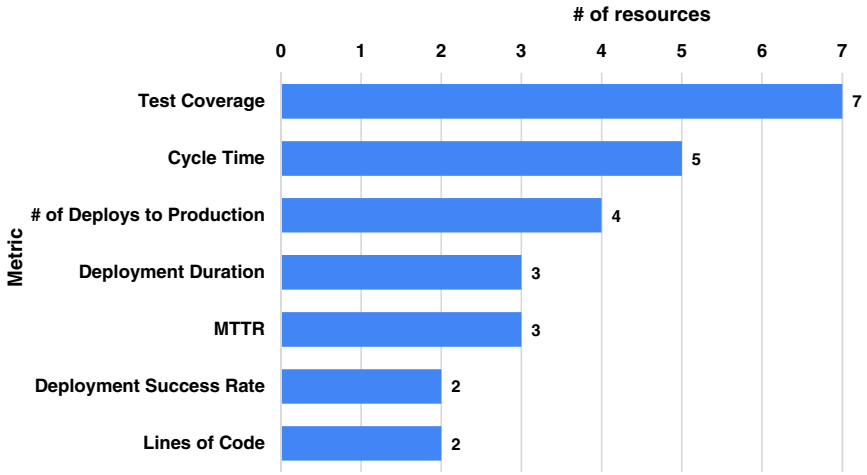

Fig. 9 Evolvability Metrics With At Least 2 Mentions in 295 GLR Resources

identify changes in their service evolution speed, preferably with drill-downs to identify the parts of the life cycle that took longer than usual.

5.2.3 Service-Based Patterns for Evolvability

We also analyzed the usage or recommendation of **service-based patterns to improve evolvability** (see Fig. 10). Nearly half of our resources reported at least one such pattern (140 of 295, 48%). Out of the 15 unique patterns, the most frequently mentioned one was *Event-Driven Messaging* (73 resources). Practitioners used it to decouple services and to allow for a more flexible service evolution. Moreover, they sometimes combined it with the related patterns *Command Query Responsibility Segregation (CQRS)* (22) or *Event Sourcing* (21) for an even greater effect. Most resources described this kind of communication architecture as more evolvable as RESTful HTTP: "The request-response pattern creates point-to-point connections that couple both sender to receiver and receiver to sender, making it hard to change one component without impacting others. Due to this, many architects use middleware as a backbone for microservice communication to create decoupled, scalable, and highly available systems."[24] In an Nginx blog post, Chris Richardson explained that messaging would decouple client and service via a shared channel so that clients could be completely unaware of the final message receivers.[25]

A lot of resources also mentioned the usage of patterns that acted as a shielding intermediary like *API Gateway* (43), *Backends for Frontends (BFF)* (12), or *Service Façade* (8). This would also reduce the efforts of service interface changes, since only the intermediary needed to be changed instead of all clients. IBM Cloud Learn Hub described the *API Gateway* as follows in their microservices guide: "While it's true that clients and services can communicate with one another directly, API gateways are often a useful intermediary layer, especially as the number of services in an application grows over time. An API gateway acts as a reverse proxy for clients by routing requests, fanning out requests across multiple

[24] https://www.confluent.io/blog/microservices-apache-kafka-domain-driven-design
[25] https://www.nginx.com/blog/building-microservices-inter-process-communication

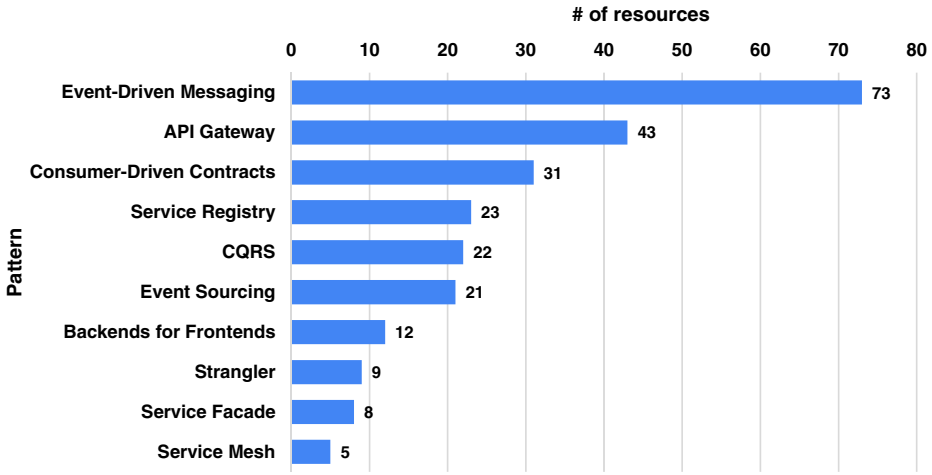

Fig. 10 Top 10 Evolvability Patterns Mentioned in 295 GLR Resources

services, and providing additional security and authentication." [26] Similarly, Microsoft Azure's Mike Wasson highlighted the advantages of the BFF pattern, namely that backend microservices did not need to respect specialized needs of different clients, which simplified and shielded services by shifting client-specific requirements to a BFF.[27]

To facilitate dynamic communication via service discovery, the *Service Registry* pattern (23) was recommended: "In an environment where service instances come and go, hard coding IP addresses isn't going to work. You will need a discovery mechanism that services can use to find each other." [13] Another means to cope with a dynamic microservices environment were patterns to manage API evolution like *Consumer-Driven Contracts* (31): "These tests should be part of the regular deployment pipeline. Their failure would allow the consumers to become aware that a change on the producer side has occurred, and that changes are required to achieve consistency again."[28] An alternative on the client-side was the *Tolerant Reader* pattern (5). Chris Richardson recommended to prepare both client and services for changes: "It makes sense to design clients and services so that they observe the robustness principle. Clients that use an older API should continue to work with the new version of the service. The service provides default values for the missing request attributes and the clients ignore any extra response attributes."[25] Lastly, a few resources described the usage of patterns for sharing common infrastructure-related needs like *Service Mesh* (5) or *Sidecar* (4). WSO2 justified this with the argument that a lot of cross-cutting concerns were shared between services and could be offloaded to infrastructure components to keep the services themselves analyzable and focused on business logic.[14]

5.3 Evolvability Reflections and Challenges (RQ3)

Our last RQ for the GLR primarily targeted **evolvability challenges** for microservices as well as their influence on the assurance process.

[26] https://www.ibm.com/cloud/learn/microservices
[27] https://azure.microsoft.com/en-us/blog/design-patterns-for-microservices
[28] https://phoenixnap.com/blog/microservices-continuous-testing

5.3.1 Evolvability Challenges

Since 74% of resources (219) mentioned at least one of the 21 identified challenges (see Fig. 11), this was the most frequent label category. The challenge we encountered most often was *Microservices Integration* (86 resources, 29%) and its specialization *Aggregating Data from Several Services* (39 resources, 13%): "The biggest challenge is aggregation of all the individual products or services and their integration with one another. As Sam Newman points out, 'Getting integration right is the single most important aspect of the technology associated with microservices in my opinion. Do it well, and your microservices retain their autonomy, allowing you to change and release them independent of the whole. Get it wrong, and disaster awaits.'"[29] Even though many experienced companies like Spotify pointed out the importance and difficulty of this problem ("The biggest complexity in a Microservice is not within the service itself, but in how it interacts with others, and that deserves special attention." [12]), they provided very little concrete guidance on how to approach it. Many questions on StackOverflow and the StackExchange communities circled around these topics and showed practitioners' uncertainty, e.g. "Microservices : aggregate data : is there some good patterns?"[30] or "Communication between two microservices"[31].

Most of these integration issues were arguably related to other prevalent challenges like *Service Cutting* (69 resources, 23%), i.e. finding the right service granularity and encapsulating related functionality in the same microservice, and its consequences *Inter-Service Dependencies / Ripples* (40 resources, 14%) and *Breaking API Changes* (30 resources, 10%). Many practitioners ended up with far too many services with lots of dependencies: "In our attempt to decompose our services we have made them very small (e.g. responsible for handling a few data attributes). This easily creates the challenge of the individual services to need to talk to each other to accomplish their own task. It is as if they're jealous of each other's data and functionality."[32] Others reported that too large services would also be problematic and common: "On their initial foray into microservices, many people are concerned that they'll overpartition their functionality and end up with too many tiny microservices. In my experience, overpartitioning is rarely the issue; it's more common to stuff too much into each service."[33] Additionally, many developers were often unsure if they should add new functionality to an existing service or create a new one. With respect to harmful coupling between services, Mark Richards highlighted the importance to control dependencies: services should be isolated as much as possible, without a lot of direct communication with other services.[34] One frequently encountered advice was to simply merge two highly coupled services: "If two services are constantly calling back to one another, then that's a strong indication of coupling and a signal that they might be better off combined into one service."[35]

Another common challenge was the increased *Architectural / Technical Complexity* (67 resources, 23%), which would make it more difficult to understand, extend, and operate the

[29] https://www.cigniti.com/blog/testing-microservices-architecture-strategy
[30] https://stackoverflow.com/questions/57788014
[31] https://stackoverflow.com/questions/36701111
[32] https://www.tigerteam.dk/2014/micro-services-its-not-only-the-size-that-matters-its-also-how-you-use-them-part-1
[33] https://techbeacon.com/app-dev-testing/5-fundamentals-successful-microservice-design
[34] https://www.oreilly.com/content/microservices-antipatterns-and-pitfalls
[35] https://opensource.com/article/18/4/guide-design-microservices

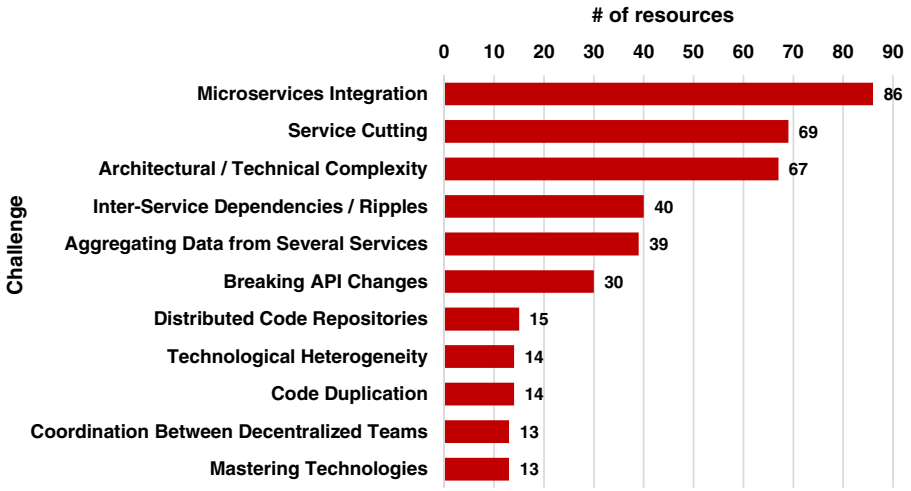

Fig. 11 Evolvability Challenges With At Least 10 Mentions in 295 GLR Resources

system: "Things can get a lot harder for developers. In the case where a developer wants to work on a journey, or feature which might span many services, that developer has to run them all on their machine, or connect to them. This is often more complex than simply running a single program."[36] Many resources described this as a gradually increasing problem, especially if the necessary automation and infrastructure was not yet present. Thorben Janssen wrote in a blog post on Stackify that implementing a single microservice would be straightforward, but – due to the technical complexity of distributed systems – this would quickly change when developing several inter-connected services.[37] In some cases, this complexity became so unmanageable that teams even moved back to more coarse-grained or monolithic structures, as for example described by the company Segment: "It seemed as if we were falling from the microservices tree, hitting every branch on the way down. Instead of enabling us to move faster, the small team found themselves mired in exploding complexity. Essential benefits of this architecture became burdens."[38] As specialized challenges of this complexity, practitioners also reported *Distributed Code Repositories* (15), *Code Duplication* (13), and having *No System-Centric View* (8). While one repository per service would have the benefit of isolation, it also would come with a management overhead: "As soon as you start reaching 30+ microservices, managing source repositories/versioning and setting CI/CD hooks is going to become an increasingly important but also a very tedious process, and there is no way around it. Managing repositories will add complexity and costs for you."[39] Lastly, keeping an overview of all microservices and their interactions was also described as near impossible: "Your management tools no longer work as well, as they may not have ways of visualizing complex microservices views. [...] Microservices architectures without some amount of structure are difficult to rationalize and reason with, as there is no obvious way to categorize and visualize the purpose of each microservice."[40]

[36] https://dwmkerr.com/the-death-of-microservice-madness-in-2018
[37] https://stackify.com/communication-microservices-avoid-common-problems
[38] https://segment.com/blog/goodbye-microservices
[39] https://blog.usejournal.com/microservices-have-you-thought-this-through-44fc2d829fe3
[40] https://www.slideshare.net/AbhishekSood10/the-top-6-microservices-patterns-83814065

In addition to these many technical and architectural challenges, we also found a few organizational or human-related challenges. In the context of governance, some practitioners warned against a too high degree of *Technological Heterogeneity* (14), as this would also come with the burden of *Mastering Technologies* (13): "I wouldn't recommend mixing too many programming languages because hiring people gets more difficult. Also, the context switches for your programmers would slow down development."[41] An explosion in programming languages could easily increase maintenance and evolution efforts. According to Susan Fowler's "Production-Ready Microservices" guide, it would therefore be more sensible to decide on a small set of languages, frameworks, and libraries to avoid the burden of supporting a multitude of technologies.[42] Finally, practitioners reported the *Coordination Between Decentralized Teams* (13) as another challenge in this area: "One obvious source of complexity is coordination and consensus between teams. In order to control this complexity, this necessitates activities like a deeper understanding of the common libraries and dependencies used within the microservices."[43]

5.3.2 Influence of Microservices on the Assurance Process

As a last focus of analysis, we wanted to determine how practitioners described the **influence of microservices on the assurance process**. The 78 resources (26%) in this area predominantly reported a negative influence (81 of 101 mentions). The most prominently used label was *Increased Testing Complexity* (60 resources, 20%). Many practitioners pointed out the need for new approaches in this area: "Microservices demand a new approach to QA. In contrast to monolithic applications, where every part of an application can be tested at the same time, microservices make QA much more complicated because each microservice may be developed and delivered according to its own schedule."[44] Even in comparatively small microservice-based systems, testing was described as much more difficult, especially if each service would have a specialized technology stack and its own code base, dependencies, feature branches, and database technology.[45] Additionally, several questions on StackExchange like "*How to test a cluster of microservices?*"[46] or "*What is the role of QA in testing an application having MicroServices architecture?*"[47] circled around this topic. Other described negative influences were related to the *Difficult Root Cause Analysis* (11), *Difficult Macroarchitecture Assessment* (6), or *Difficult Quality Analysis* (4) in general. Michael Kutz from REWE digital reported the following experiences: "While the Microservice architectural style has a lot of benefits, it makes certain QA practices impractical: there is no big release candidate that can be tested before put to production, no single log file to look into for root cause analysis and no single team to assign found bugs to. Instead there are deployments happening during test runs, as many log files as there

[41] https://codingsans.com/blog/microservice-architecture-best-practices
[42] https://www.oreilly.com/library/view/production-ready-microservices/9781491965962/ch01.html
[43] https://www.capitalone.com/tech/software-engineering/analyzing-polyglot-microservices
[44] https://blog.runscope.com/posts/qa-in-a-microservices-world
[45] https://www.freecodecamp.org/news/these-are-the-most-effective-microservice-testing-strategies-according-to-the-experts-6fb584f2edde
[46] https://devops.stackexchange.com/questions/731
[47] https://sqa.stackexchange.com/questions/33998

are microservices and many teams to mess with the product."[48] A similar story is told by Mark Balbes and Zachary Becker from WWT Asynchrony Labs: "Microservices pose a new challenge for QA. While maintaining the quality of a given microservice is simplified, the complexity of a monolithic application doesn't disappear. It is simply transferred from the software to the interconnectedness of the systems running the many microservices. So QA has to consider not just the quality of the individual microservice but of the entire web of microservices and systems that support them."[49]

On the other hand, some practitioners also described three different positive influences (20 mentions). Developers would have more motivation to improve a small service (8) and new services would be of good quality (7). Moreover, the organizational and cultural changes would lead to more quality awareness (5) in general: "There are smaller decentralized teams of 5-7 people that have their own set of KPIs and success metrics to achieve. This allows them to take ownership of 'their' product and it gives them better clarity on the progress. It also gives them the freedom to innovate, which boosts their morale. [...] As you can see, using DevOps and microservices architecture together will not only boost the productivity of the team, but it will also enable them to develop a more innovative and better quality product at a faster pace."[50] Sachin Dhamane from Majesco reported similar positive cultural changes: "Organizations need to cultivate a culture of sharing responsibility, ownership and complete accountability in microservices teams. [...] They take full ownership for their services, often beyond the scope of their roles or titles, by thinking about the end customer's needs and how they can contribute to solving those needs."[51] Lastly, REWE's Michael Kutz also highlighted the positive influence of the "you build it, you run it" DevOps principle. Empowering teams while simultaneously demanding responsibility would lead to an increase in service quality: "Teams are made responsible for their services during office hours, but also after that, even at 3:14 am on a Saturday! Since most people working for us like sleeping, we expect this to be a good motivation to improve the services' reliability a lot. You simply pay more attention to technical debt, when you pay it in sleep!"[52]

6 Discussion

When interpreting the combined results of the interviews and the GLR, we found a lot of commonalities, but also some differences. In the following paragraphs, we discuss these findings per RQ-related topic. Finally, Table 4 provides a summarized overview to compare the results of both studies.

With respect to perceived challenges for the evolvability of microservices, *Service Cutting* was reported as one of the most important issues in both studies. Many practitioners were struggling with finding an adequate service decomposition. As a result, related challenges like *Microservices Integration*, *Inter-Service Dependencies / Ripples*, *Aggregating Data from Several Services*, or *Breaking API Changes* were common as well, even though they were much more prevalent in the GLR resources. The GLR results were therefore more focused on architectural and technical challenges, while the interviews also had a

[48] https://agiletestingdays.com/2019/session/team-driven-microservice-quality-assurance
[49] https://adtmag.com/articles/2018/04/11/agile-and-tranforming-qa.aspx
[50] https://www.thinksys.com/microservices/microservices-comes-together-brilliantly-with-devops
[51] https://www.majesco.com/innovating-insurance-microservices-part-3
[52] https://slides.com/mkutz/team-driven-microservicequality-assurance

Table 4 Comparison of Interview Results (N=17) and GLR Results (N=295); the percentages refer to the fraction of total participants/resources that mentioned the label

Category		Interviews	GLR
Process	Mentions	26 labels, 120 times	20 labels, 373 times
	Top 3	Test Automation (13; 76%)	Test Automation (94; 32%)
		Governance (12; 71%)	Principle/Guideline (66; 22%)
		Standardization (10; 59%)	Decentralization & Empowerment (37; 13%)
	Focus	**Decentralization/governance**	**Decentralization/governance**
Tools	Mentions	11 labels, 59 times	47 labels, 106 times
	Top 3	SonarQube (14; 82%)	Pact (15; 5%)
		FindBugs (12; 71%)	Selenium (8; 3%)
		Checkstyle (9; 53%)	WireMock (7; 2%)
	Focus	**Static code analysis**	**Test automation**
Metrics	Mentions	14 labels, 46 times	27 labels, 46 times
	Top 3	Test Coverage (10; 59%)	Test Coverage (7; 2%)
		# of Code Smells (4; 24%)	Cycle Time (5; 2%)
		Clone Coverage (4; 24%)	# of Deploys to Production (4; 1%)
	Focus	**Code quality**	**Productivity**
Patterns	Mentions	9 labels, 25 times	15 labels, 264 times
	Top 3	Event-Driven Messaging (12; 71%)	Event-Driven Messaging (73; 25%)
		Backends for Frontends (2; 12%)	API Gateway (43; 15%)
		Consumer-Driven Contracts (2; 12%)	Consumer-Driven Contracts (31; 11%)
	Focus	**Reducing coupling**	**Reducing coupling, evolution management**
Challenges	Mentions	24 labels, 84 times	21 labels, 430 times
	Top 3	Service Cutting (9; 53%)	Microservices Integration (86; 29%)
		No System-Centric View (8; 47%)	Service Cutting (69; 23%)
		Mastering Technologies (8; 47%)	Architectural/Technical Complexity (67; 23%)
	Focus	**Architecture, organization**	**Architecture, integration**

decent amount of organizational and human-related challenges like *Mastering Technologies* or *Technological Heterogeneity*. In general, the prevalence of most interview challenges was confirmed by the GLR results. We identified only two new labels in this category, namely *Aggregating Data from Several Services* and *Vendor / Ecosystem Lock-in*. Concerning the influence of microservices on the assurance process, no new labels were formed during the GLR analysis. Except for *Increased Testing Complexity* (60 GLR resources), this small category was rarely mentioned in both studies. While the analysis revealed negative as well as positive influences, the GLR results were generally more negative than the interview participants suggested. Besides the more complex testing, keeping a system-centric quality view of the architecture was seen as difficult. All positive mentions like the increased cultural quality awareness are predominantly helping the quality of single services in isolation.

The analysis of applied practices in both studies revealed that industry adopted ways to at least partially address some of the mentioned challenges. Concerning the general assurance process, we observed a similar theme of finding a balance between decentralization and standardization, even though the GLR resources generally perceived decentralization as more positive. Guidelines for foundational standardization and principles for service communication and architecture were also important in both studies. Finding a balanced "Goldilocks" governance was promoted as a means to partially address challenges like

Architectural / Technical Complexity, *Technological Heterogeneity*, or *Mastering Technologies*. Additionally, test automation was highly valued, but much more pronounced in the GLR resources, where a lot of practitioners described detailed suggestions for effective microservices testing. Unfortunately, we could not identify much information about the concrete structure of assurance processes within the GLR resources, especially concerning CI/CD pipeline integration, quality gates, review practices, or refactoring. In this area, the interview results were much richer and the GLR introduced only three new labels (*QA in Production*, *Feature Toggles*, and *Code Generation*).

In the area of evolvability tools and metrics, the two studies produced very different results. Our interview participants reported a variety of static analysis tools, while the vast majority of tools in the GLR were test automation tools and frameworks. The GLR introduced 46 new tool labels and only used one from the interviews (SonarQube). For metrics, test coverage was the most frequently mentioned metric in both studies, but while the interviews primarily mentioned source code quality metrics, the GLR resources mostly described productivity metrics and led to the creation of 19 new labels. In general, these two label categories were not well represented and the majority of tools and metrics were only mentioned once or twice. With the exception of test automation, our analysis suggests that most practitioners have not put much thought into these topics. Especially concerning is the absence of architectural or service-oriented tools and metrics, even though the most crucial challenges were of an architectural nature.

Lastly, our two studies revealed the usage of 15 service-oriented patterns to improve evolvability. While patterns were not strongly represented in the interviews, they were very prevalent in the GLR resources, which also mentioned six additional patterns not reported by interview participants. The majority of mentions focused on a small number of patterns, e.g. in the GLR, the top five patterns accounted for 73% of used pattern labels. The top patterns were similar in both studies and *Event-Driven Messaging* was by far the most frequently mentioned pattern. Recommended patterns were predominantly used to reduce coupling and to mitigate the risk of breaking API changes, i.e. they could partially address challenges like *Microservices Integration*, *Inter-Service Dependencies / Ripples*, and *Breaking API Changes*. However, many practitioners also warned that most of these patterns simultaneously introduced additional complexity in the system, which required considerable experience and made them difficult to apply for novice developers.

During our interviews, we collected several properties about the 14 discussed systems and analyzed their influence on the applied evolvability assurance. However, we did not identify many noteworthy relationships. While system size seems like an intuitive driver for different processes, tools, metrics, or patterns, we did not observe substantial differences between smaller and larger systems in this regard. It should be noted, though, that this observation was only based on the rough estimates for the number of services and the number of involved developers as reported by our participants. A thorough and explicit analysis of this relationship would require more extensive data about the systems. Similarly, we did not find a relationship between system age and assurance practices. One potential reason for this may be that most discussed systems were fairly young. With the exception of S14 (6 years), all systems were between 1 and 3 years old. The only major influence on the assurance process we observed was whether the system was built for an external customer. In this case, there was a tendency to rely much more on central governance and standardization, while systems based on internal product development exhibited more decentralized assurance processes with more autonomous teams.

In summary, finding a suitable service granularity without harmful dependencies, ripple effects, or chatty inter-service communication to fulfill basic operations is by far the top challenge. Due to the dynamic evolution of decentralized microservice-based systems, this is also not a problem that can be solved once and for all. With every new feature, the question will arise whether to add it to an existing service or to create a new one. Continuous architecture evaluation with respect to properties such as coupling, cohesion, or complexity is necessary to tackle these challenges in a microservice-based system. Practitioners recommended some guidelines, from which very few are actionable for inexperienced developers, and a number of patterns like *Event-Driven Messaging* to at least reduce the impact of these issues. Nonetheless, our results also indicate that very few specialized tools, metrics, or other structured evaluation approaches are used in this area. This does not necessarily mean that no evolvability assurance approaches for microservices exist. Especially early adopter companies like Netflix, Amazon, or Spotify are more advanced in this regard and rely for example on distributed tracing for architecture visualization and evaluation. However, such practices were not used in our sample.

7 Threats to Validity

During preparation, execution, and documentation of our studies, we aimed for rigor, consistency, and objectivity. However, qualitative methods can be especially prone to various subjective biases if not executed with great care. In this section, we therefore discuss potential limitations of our work according to the general validity dimensions of Wohlin et al. (2003) while respecting how these can materialize in qualitative studies (Easterbrook et al. 2008). We describe potential threats in the most common categories – construct, internal, conclusion, and external validity – and outline the steps we have taken to minimize or mitigate them.

Construct validity addresses the appropriateness of the chosen methods, the extent to which selected measurements are suitable to test a hypothesis or theory, and if the constructs under study are interpreted in a commonly used way. Threats in this category can be the appropriateness of the chosen research questions, the selected methods, or the categorization scheme for data collection and extraction. We carefully reviewed existing literature in the field to define a research objective and the subsequent questions. Likewise, we clearly defined our central constructs and study protocols before conducting the research. Lastly, we combined the two complementary methods of semi-structured interviews and grey literature review to obtain a rich and in-depth data set from a variety of industry sources.

Internal validity is concerned with the scientific rigor and consistency during study execution as well as with unknown factors that may influence the correctness of the obtained data. Concerning our interviews, it could be possible that participants were not completely honest with their answers, which is a common problem in survey- or interview-based studies. Nonetheless, we think that the risk for this study is rather low. Most interviewees seemed quite comfortable to describe negative aspects of their systems and the discussed topics were only of low to medium sensitivity. Furthermore, confidentiality and anonymity were provided. As a second threat, participants could have provided incorrect answers because they misunderstood questions or concepts. To limit this risk, important terms were defined before the corresponding topics. As an example, the non-validated maturity model and its levels could also have introduced bias or misunderstandings, even though it was only used as an icebreaker. However, we posed additional clarifying questions if participants used terms of

vague meaning. Interviewees also asked questions themselves, if something was not fully clear to them. In some cases, our analysis relied on the subjective opinion of participants, e.g. how they rated the evolvability of their system. Since participants could interpret such ratings very differently, detailed comparisons between systems may be difficult. Moreover, we only relied on one single opinion for most systems and people may have different opinions of quality, as is visible in the case of S9. To increase consistency and reduce researcher bias, every transcript was proofread by both moderators. Additionally, we strongly encouraged our participants to adjust wrong or unclear statements in the transcripts. Nevertheless, there still remains the small possibility that some of the more figurative paragraphs were misinterpreted.

For our GLR, we followed a systematic approach to ensure transparency and repeatability, thereby reducing common issues with multivocal and grey literature reviews. This involved a detailed protocol with the documentation of search engines, search terms, inclusion / exclusion criteria, and coding system. All those artifacts are publicly available in our repositories. One potential threat might arise from the difficulty of analyzing the authority and trustworthiness of sources. It was not always obvious if the assurance recommendations were opinions or actually based on substantial experience. Some practitioners may have simply repeated ideas or concepts they picked up elsewhere. Moreover, the selection of search engines and StackExchange communities as well as the stopping criteria were partly subjective. We tried to minimize this threat through the selection of the two most popular search providers, the inclusion of a large amount of initial search hits in comparison to existing studies, as well as a general methodological alignment with related studies in the field. Unfortunately, the reproducibility of our searches through Google and Bing is limited, as the underlying algorithms may change and it is not possible to select a time frame.

As the terminology for certain concepts may vary, GLR searches may miss important sources. Therefore, we used seven elaborate search strings and considered several synonyms to avoid narrowing down search results too much. However, the application of inclusion / exclusion criteria and the assignment of coding labels were based on the researchers' judgment and therefore inevitably involved some bias. To process the large amount of 1,245 sources, splitting among two researchers was necessary as well, which may have affected the consistency of the results. Both coders had several years of experience in the related industry field and thus were qualified to perform the important coding task. The initial inter-rater reliability calibration provided a tangible measurement of the consensus. Furthermore, all unclear cases were discussed between the two coders. Therefore, we are confident that following this procedure sufficiently reduced researcher bias and the incorrect inclusion / exclusion of sources. In the end, it is not necessary that 100% of the resources were selected and coded "correctly". Some researchers claim that such a notion of correctness does not even exist in qualitative and constructivist studies (Easterbrook et al. 2008). Instead, it is important that the reported general distributions and tendencies of our GLR resources and labels are correct, which we believe to be the case.

Conclusion validity addresses the concern that the drawn conclusions may not be reasonably grounded in the gathered data or observed phenomena. This may in particular be the case for evolvability tools and metrics, which were generally identified in a rather low quantity compared to e.g. evolvability patterns or challenges. Thus, the perceived low importance of those categories is partly more interesting than the distribution of concepts within those categories. Overall, the largest threat for qualitative studies in this area is that summarizing and aggregating results depends heavily on the subjective interpretation of researchers. We

tried to limit this bias by relying on a precise coding system and by critically discussing all interpretations and implications between the two coders.

External validity is the degree to which the results and conclusions obtained through a limited sample can be transferred to the general population. From our 17 interviews, no distributions can be generalized, e.g. the industry usage of SonarQube for microservices. Because of its qualitative nature, the center of this research are the intentions and opinions of participants as well as the relationships between concepts. An additional threat for generalizability could be that our participants were exclusively based in Germany, even though we included European and international companies. We also tried to achieve diversity with respect to company domains and sizes, but had nine participants from software and IT services companies (52%). Lastly, possible selection bias could only be reduced in the case of C2, where we applied random sampling to pick three out of seven candidates.

In our GLR, we limited the scope to English resources only. We are confident that this provides a fair representation of literature in the studied area, as English is the most frequently used language by IT professionals. However, we need to be careful to generalize from a vocal minority of individuals and companies that distribute their viewpoints to the public via blog posts and articles. Even though a lot of practitioners may read and follow the published advice, it is most certainly not a complete representation of the target population. The inclusion of popular Q&A forums may have reduced this bias by broadening the selection of perceived challenges. Additionally, the differences between interview and GLR results in some areas make a complete generalization more difficult. As an example, the fact that our GLR results were generally more negative than the sentiments of our interview participants may have been influenced by the respective method. Nonetheless, we are confident that findings that were consistent within both studies have a solid foundation for generalization, e.g. the most frequent challenges as well as the lack of tool support, metrics, and structured approaches for architecture evaluation.

Lastly, it needs to be mentioned that some companies with more microservices experience like Netflix or Amazon apply more advanced practices than the ones identified in our sample, even though we did not find resources on this during our GLR. As a result, they may also experience many of the discussed challenges as less crucial. Several findings therefore may not generalize well to these early adopters. Instead, we believe that many results mostly apply for companies with less microservices experience, i.e. companies that could benefit the most from systematic approaches from industry or academia.

8 Conclusion

To analyze industry practices and challenges for the evolvability assurance of microservices, we conducted two qualitative studies. First, we interviewed 17 software professionals from 10 companies and talked with them about the evolvability assurance for 14 different microservice-based systems. Second, we performed a systematic grey literature review (GLR) and applied the interview coding system to 295 practitioner online resources to confirm the prevalence of the identified practices and challenges. The combined results suggest that the evolvability assurance of microservices requires new techniques to tackle the most critical reported challenges. The quality of individual services was not described as problematic, but the architecture of the system presented a focus point for issues. While practitioners were able to (partially) address several problems by using specialized test automation, evolvability patterns like *Event-Driven Messaging*, balanced standardization, or guidelines for service design and communication, many architectural and technical

challenges remain. Especially inexperienced teams may face problems related to *Service Cutting* and *Microservices Integration*, since no tool support and service-based metrics are employed to aid in the difficult macroarchitectural evaluation.

Future work that covers the areas of maintenance, evolution, and technical debt in the context of microservices should take these findings and industry sentiments into account. In particular, academia can support industry with methods, metrics, or tools that enable macroarchitectural assessment of microservices or provide a more system-centric view. We perceived tool support for service cutting activities and metrics to continuously evaluate service granularity, coupling, or cohesion as concrete gaps that could save industry substantial refactoring efforts. Finally, issues related to the dynamic evolution and decentralized nature of microservices like breaking API changes, technological heterogeneity, or the coordination and knowledge exchange between decentralized teams were described as important, since they would also require organizational solutions and cultural changes.

For transparency and to support future research and replications, we published all artifacts and results of both studies on GitHub (interviews[53], GLR[54]) and Zenodo (interviews[55], GLR[56]).

Funding Open Access funding enabled and organized by Projekt DEAL. This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the doctoral program *Services Computing* (https://www.services-computing.de/?lang=en).

Availability of data and material

- Interviews
  - GitHub: https://github.com/xJREB/research-microservices-evolvability-interviews
  - Zenodo: https://doi.org/10.5281/zenodo.2586916
- GLR
  - GitHub: https://github.com/xJREB/research-microservices-evolvability-glr
  - Zenodo: https://doi.org/10.5281/zenodo.3731259

References

April A, Huffman Hayes J, Abran A, Dumke R (2005) Software maintenance maturity model (SMmm): the software maintenance process model. J Softw Maint Evol Res Pract 17(3):197–223. https://doi.org/10.1002/smr.311

[53] https://github.com/xJREB/research-microservices-evolvability-interviews

[54] https://github.com/xJREB/research-microservices-evolvability-glr

[55] https://doi.org/10.5281/zenodo.2586916

[56] https://doi.org/10.5281/zenodo.3731259

Avgeriou P, Kruchten P, Ozkaya I, Seaman C (2016) Managing technical debt in software engineering. Dagstuhl Rep. 6(4):110–138. https://doi.org/10.4230/DagRep.6.4.110

Bandeira A, Medeiros CA, Paixao M, Maia PH (2019) We need to talk about microservices: an analysis from the discussions on stackoverflow. In: 2019 IEEE/ACM 16th international conference on mining software repositories, (MSR) IEEE, pp 255–259. https://doi.org/10.1109/MSR.2019.00051

Baškarada S, Nguyen V, Koronios A (2018) Architecting microservices: Practical opportunities and challenges. J Comput Inf Syst 00(00):1–9. https://doi.org/10.1080/08874417.2018.1520056

Bogner J, Wagner S, Zimmermann A (2017) Automatically measuring the maintainability of service- and microservice-based systems: a literature review. In: Proceedings of the 27th international workshop on software measurement and 12th international conference on software process and product measurement on - IWSM Mensura '17, ACM Press, New York, New York, USA, pp 107–115. https://doi.org/10.1145/3143434.3143443

Bogner J, Fritzsch J, Wagner S, Zimmermann A (2018) Limiting technical debt with maintainability assurance: An industry survey on used techniques and differences with service- and microservice-based systems. In: Proceedings of the 2018 international conference on technical debt - TechDebt '18, ACM Press, New York, New York, USA, pp 125–133. https://doi.org/10.1145/3194164.3194166

Bogner J, Fritzsch J, Wagner S, Zimmermann A (2019a) Assuring the evolvability of microservices: Insights into industry practices and challenges. In: 2019 IEEE International conference on software maintenance and evolution (ICSME), IEEE, Cleveland, Ohio, USA, pp 546–556. https://doi.org/10.1109/ICSME.2019.00089

Bogner J, Fritzsch J, Wagner S, Zimmermann A (2019b) Microservices in industry: Insights into technologies, characteristics, and software quality. In: 2019 IEEE international conference on software architecture companion (ICSA-C), IEEE, Hamburg, Germany, pp 187–195. https://doi.org/10.1109/ICSA-C.2019.00041

Carrasco A, van Bladel B, Demeyer S (2018) Migrating towards microservices: migration and architecture smells. In: Proceedings of the 2nd international workshop on refactoring - IWoR, 2018, ACM Press, New York, New York, USA, pp 1–6. https://doi.org/10.1145/3242163.3242164

Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Meas 20(1):37–46. https://doi.org/10.1177/001316446002000104

Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting empirical methods for software engineering research. In: Guide to advanced empirical software engineering, Springer London, London, pp 285–311 https://doi.org/10.1007/978-1-84800-044-5_11

Esparrachiari S, Reilly T, Rentz A (2018) Tracking and controlling microservice dependencies. Queue 16(4):44–65. https://doi.org/10.1145/3277539.3277541

Fowler M (2019) Microservices resource guide. http://martinfowler.com/microservices

Fritzsch J, Bogner J, Wagner S, Zimmermann A (2019a) Microservices migration in industry: Intentions, strategies, and challenges. In: 2019 IEEE International conference on software maintenance and evolution (ICSME), IEEE, Cleveland, Ohio, USA, pp 481–490. https://doi.org/10.1109/ICSME.2019.00081

Fritzsch J, Bogner J, Zimmermann A, Wagner S (2019b) From monolith to microservices: A classification of refactoring approaches. In: Bruel JM, Mazzara M, Meyer B (eds) Software engineering aspects of continuous development and new paradigms of software production and deployment. Springer, Toulouse, pp 128–141. https://doi.org/10.1007/978-3-030-06019-0_10

Garousi V, Felderer M, Mäntylä MV (2016) The need for multivocal literature reviews in software engineering. In: Proceedings of the 20th international conference on evaluation and assessment in software engineering - EASE '16, ACM Press, New York, New York, USA, vol 01-03-June, pp 1–6 https://doi.org/10.1145/2915970.2916008

Garousi V, Felderer M, Mäntylä MV (2019) Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Inf Softw Technol 106(September 2018):101–121. https://doi.org/10.1016/j.infsof.2018.09.006

Ghofrani J, Lübke D (2018) Challenges of microservices architecture: A survey on the state of the practice. In: 10th Central European workshop on services and their composition (ZEUS), CEUR-WS.org, Dresden, Germany, vol 10th

Haselböck S, Weinreich R, Buchgeher G (2018) An expert interview study on areas of microservice design

Hove S, Anda B (2005) Experiences from conducting semi-structured interviews in empirical software engineering research. In: 11th IEEE international software metrics symposium (METRICS'05), IEEE, Metrics, pp 23–23. https://doi.org/10.1109/METRICS.2005.24

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159. https://doi.org/10.2307/2529310

Lehman M (1980) Programs, life cycles, and laws of software evolution. Proc. IEEE 68(9):1060–1076. https://doi.org/10.1109/PROC.1980.11805

Lenarduzzi V, Taibi D (2018) Microservices, Continuous Architecture, and Technical Debt Interest: An Empirical Study. In: Euromicro SEAA Prague, Czech Republic, June

Neri D, Soldani J, Zimmermann O, Brogi A (2019) Design principles, architectural smells and refactorings for microservices: a multivocal review. SICS Software-Intensive Cyber-Physical Systems. https://doi.org/10.1007/s00450-019-00407-8, 1906.01553

Neto GTG, Santos WB, Endo PT, Fagundes RA (2019) Multivocal literature reviews in software engineering: Preliminary findings from a tertiary study. In: 2019 ACM/IEEE International symposium on empirical software engineering and measurement (ESEM), IEEE, vol 2019-Septe, pp 1–6 https://doi.org/10.1109/ESEM.2019.8870142

Newman S (2015) Building Microservices: Designing Fine-Grained Systems, 1st edn. O'Reilly Media Sebastopol, USA

Rajlich V (2018) Five recommendations for software evolvability. J Softw Evol Process 30(9):e1949. https://doi.org/10.1002/smr.1949

Rowe D, Leaney J, Lowe D (1998) Defining systems architecture evolvability - a taxonomy of change. In: International conference on the engineering of computer-based systems, IEEE, pp 45–52 https://doi.org/10.1109/ECBS.1998.10027

Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Emp Softw Eng 14(2):131–164. https://doi.org/10.1007/s10664-008-9102-8, 9809069v1

Schermann G, Cito J, Leitner P (2016) All the services large and micro: Revisiting industrial practice in services computing. In: Norta A, Gaaloul W, Gangadharan GR, Dam HK (eds) Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics), lecture notes in computer science, vol 9586. Springer, Berlin, pp 36–47. https://doi.org/10.1007/978-3-662-50539-7_4

Seaman CB (2008) Qualitative methods. In: Guide to advanced empirical software engineering. Springer, London, pp 35–62. https://doi.org/10.1007/978-1-84800-044-5_2

Software Engineering Institute (2010) CMMI® for Development, Version 1.3 (CMMI-DEV V1.3). Tech. rep. Software Engineering Institute

Soldani J, Tamburri DA, Van Den Heuvel WJ (2018) The pains and gains of microservices: A Systematic grey literature review. J Syst Softw 146(September):215–232. https://doi.org/10.1016/j.jss.2018.09.082

Taibi D, Lenarduzzi V, Pahl C (2020) Microservices anti-patterns: A taxonomy, Springer International Publishing, Cham. https://doi.org/10.1007/978-3-030-31646-4_5, 1908.04101

Wagner S (2013) Software Product Quality Control. Springer, Berlin. https://doi.org/10.1007/978-3-642-38571-1

Wohlin C, Höst M, Henningsson K (2003) Empirical research methods in software engineering. In: Esernet, vol 2765. Springer, Berlin, pp 7–23. https://doi.org/10.1007/978-3-540-45143-3_2

Affiliations

Justus Bogner[1] · Jonas Fritzsch[1] · Stefan Wagner[1] · Alfred Zimmermann[2]

Jonas Fritzsch
jonas.fritzsch@iste.uni-stuttgart.de

Stefan Wagner
stefan.wagner@iste.uni-stuttgart.de

Alfred Zimmermann
alfred.zimmermann@reutlingen-university.de

[1] Institute of Software Engineering, University of Stuttgart, Stuttgart, Germany

[2] Herman Hollerith Center, University of Applied Sciences Reutlingen, Reutlingen, Germany