Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality

Conference Paper · March 2019

DOI: 10.1109/ICSA-C.2019.00041

4 authors, including:

Justus Bogner
Vrije Universiteit Amsterdam
81 PUBLICATIONS **1,008** CITATIONS
SEE PROFILE

Jonas Fritzsch
Universität Stuttgart
29 PUBLICATIONS **358** CITATIONS
SEE PROFILE

Alfred Zimmermann
Reutlingen University - Herman Hollerith Center
166 PUBLICATIONS **1,694** CITATIONS
SEE PROFILE

Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality

Justus Bogner*†, Jonas Fritzsch†*, Stefan Wagner†, Alfred Zimmermann*
* University of Applied Sciences Reutlingen, Germany
† University of Stuttgart, Germany
{justus.bogner, alfred.zimmermann}@reutlingen-university.de
{jonas.fritzsch, stefan.wagner}@informatik.uni-stuttgart.de

***Abstract*—Microservices are a topic driven mainly by practitioners and academia is only starting to investigate them. Hence, there is no clear picture of the usage of Microservices in practice. In this paper, we contribute a qualitative study with insights into industry adoption and implementation of Microservices. Contrary to existing quantitative studies, we conducted interviews to gain a more in-depth understanding of the current state of practice. During 17 interviews with software professionals from 10 companies, we analyzed 14 service-based systems. The interviews focused on applied technologies, Microservices characteristics, and the perceived influence on software quality. We found that companies generally rely on well-established technologies for service implementation, communication, and deployment. Most systems, however, did not exhibit a high degree of technological diversity as commonly expected with Microservices. Decentralization and product character were different for systems built for external customers. Applied DevOps practices and automation were still on a mediocre level and only very few companies strictly followed the *you build it, you run it* principle. The impact of Microservices on software quality was mainly rated as positive. While maintainability received the most positive mentions, some major issues were associated with security. We present a description of each case and summarize the most important findings of companies across different domains and sizes. Researchers may build upon our findings and take them into account when designing industry-focused methods.**

***Index Terms*—Microservices, industry, interviews, service technology, software quality**

I. Introduction

Over the last 15 years, service-oriented computing [1] has steadily gained popularity in industry. The two service-based architectural styles Service-Oriented Architecture (SOA) [2] and Microservices [3], [4] are important foundations for enterprise applications with strong requirements for e.g. modifiability or scalability. While Microservices emerged from industry practice, they are currently heavily investigated in academia. Since most scientific publications focus on characteristics from a few early adopters like Amazon, Netflix, or ThoughtWorks [5], [6], academia needs to take care to get a complete picture of Microservices in industry. Scientific methods and techniques designed for "pure" Microservices may not be applicable to or relevant for systems that do not follow all postulated Microservices characteristics.

There are already a few empirical studies on industry adoption and implementation details [7], [8] that reveal a high degree of diversity in service-oriented industry practice (see section V). Schermann et al. [7] even report that several academic assumptions may be incorrect for typical service-based systems. While the majority of the empirical industry research on Microservices is quantitative and survey-based, important insights into the rationale for these differences could be gained via qualitative and interview-based methods. To address this gap and to provide additional empirical and industry-focused research, we conducted 17 in-depth interviews with software professionals based in Germany from 10 different companies. We talked with them about 14 different systems covering a number of topics including applied service technologies, adherence to Microservices characteristics, and the perceived influence on software quality.

II. Scope and Research Method

As a top-level guidance, we followed the five-step case study research process as described by Runeson and Höst [9].

Study Design: Our overarching research objective is to provide insights into industry adoption and implementation of Microservices as well as into rationales in this area. This objective is framed by the following three research questions:
RQ1: Which technologies do companies use for the implementation and operation of Microservices and with what rationale?
RQ2: Which characteristics of Microservices do companies respect, which do they neglect, and for which reasons?
RQ3: How do companies perceive the influence of Microservice architectures on software quality?

Qualitative methods seem fitting to answer these questions, because they analyze the relations between concepts and directly deal with existing complexity [10]. Since qualitative results are therefore very rich and informative, they enable us to focus on participants' rationales. As a concrete method, we chose semi-structured interviews [10], [11], because they provide a basic agenda, but also allow for dynamic adaptation based on the responses. As interview participants, we wanted software professionals in technical roles with significant professional experience and solid knowledge of service orientation. Recent participation in the development of a service-based system (ideally Microservices) was also required. Recruiting of participants was achieved via personal industry contacts within the research group. We contacted companies from different domains and of different sizes.

TABLE I
COMPANY AND PARTICIPANT DEMOGRAPHICS

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

Preparation for Data Collection: Several documents were prepared before conducting the interviews. We created an *interview preamble* [9] outlining the interview process and topics. This document was sent to participants beforehand to make them familiar with the study. It also covered ethical considerations like confidentiality, asked for consent to create audio recordings, and communicated that audio as well as transcripts would not be published. To scope and organize the semi-structured interviews, we created an *interview guide* [10] that contained the most important questions grouped in thematic blocks. This guide was used as a loose structure for the interviews and was not shared with participants. Lastly, we created a *case characterization matrix* [10] with the most important participant attributes and study concepts.

Evidence Collection: Of the 17 individual interviews, six were conducted face to face and 11 via remote communication software with screen sharing. Except for one English interview, all others were held in German. Duration was between ∼45 and ∼75 minutes. Every participant agreed to the creation of an audio recording. During the interviews, we loosely followed the interview guide based on the participant's reactions. We manually transcribed each recording into a textual document. These transcripts were then sent to participants for review and final approval, where they had the possibility to remove sensitive information or change sentences of unclear or unwanted meaning. The final transcripts were then used for our detailed qualitative analysis.

Data Analysis: The first step was a detailed analysis of each individual case transcript. We relied heavily on the created case characterization matrix as well as *tabulation* [9] in general. The important parts for each research question were extracted from the transcript and documented. Based on the findings, we adapted the matrix with additional attributes. In the second step, we used *cross-case analysis* [10] to identify important generalizations and summaries. We documented common trends and deviations, which was then aggregated into findings and take-aways for each research question. Interview artifacts and results are available in our online repository.[1]

III. INTERVIEW CASE DESCRIPTIONS

We conducted interviews with 10 different companies (C1–C10) of various sizes and domains (see Table I). Half of them were software & IT services companies that developed a lot of systems for external clients. For the companies from other domains, system ownership was always internal. All our participants were based in Germany, even though some of the larger companies were active in several European countries or even globally. From our 17 participants (P1–P17), 11 filled the role of an architect and four were developers. All participants had a minimum of five years of professional experience, with a median of 12 and a mean of 14.7 years. The individual cases of our analysis were 14 systems (S1–S14, see Table II). We summarized the following case descriptions from participants' explanations. While we aimed for objectivity and tried to reflect their views as much as possible, some parts may be subject to our interpretation.

C1-S1, Derivatives Management System: C1, a small financial services company, offers a system for the management and search of derivatives as Software as a Service (SaaS). To increase maintainability, S1 was migrated from a monolithic PHP application and now comprises nine Microservices mostly written in Java using Spring Boot. Several smaller services group around one coarse-grained core service. Services use RESTful HTTP and AMQP messaging for communication. Although the current deployment automation of the JAR files via Ansible is seen as comfortable, a migration to Docker and potentially Kubernetes is planned to increase operability. The system has most Microservices characteristics except for only a basic degree of automation without continuous deployment and a small degree of technological heterogeneity.

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

TABLE II
SYSTEM CHARACTERISTICS

ID	Purpose	Inception	Hosting	# of People	# of Services	Communication	Languages	Artifacts
S1	Derivatives management system (banking)	Rewrite	public cloud (AWS)	7	9	REST, AMQP	Java, Kotlin	JAR
S2	Freeway toll management system	Rewrite & Extension	private cloud (OpenShift)	10 (only devs)	10	Oracle Advanced Queuing, REST	Java, C++ for algorithm	Docker
S3	Automotive problem management system	Rewrite & Extension	private cloud (OpenShift)	50	10	REST, Kafka	Java	Docker
S4	Public transport sales system	Rewrite & Extension	on-premise, public cloud (AWS)	∼300	∼100	REST, Kafka, AmazonMQ	Java, Node.js	Docker
S5	Business analytics & data integration system	Greenfield	private cloud, on-premise	7	6	REST, Kafka	Java, Scala	Docker
S6	Automotive configuration management system	Rewrite	private cloud	20	60	REST, Kafka, MQTT, JMS	Java, Node.js	Docker
S7	Retail online shop	COTS Replacement	on-premise, public cloud (AWS, Google)	∼200	∼250	REST (JSON or OData), SOAP (legacy)	Java, Node.js, Go, Kotlin	JAR/WAR
S8	IT service monitoring platform	Continuous Evolution	private cloud	15	9	REST, AMQP	Java, Node.js	Docker
S9	Hotel search engine	Continuous Evolution	on-premise, public cloud (Google)	∼50	∼10	gRPC, Kafka, REST, Unix domain socket	Java, PHP	Docker
S10	Hotel management suite	Rewrite & Extension	on-premise, public cloud (AWS)	50	20	REST, Google Pub/Sub	PHP, Java	Docker
S11	Public transport management suite (HR management part: S11a)	Continuous Evolution	on-premise	∼175	10 products	REST, SOAP, file transfer, RPC/RMI	Java, C++	JAR/EAR
S12	Retail online shop	COTS Replacement	public cloud (AWS)	∼85	∼45	REST, Kafka	Java, Kotlin, Scala, Go	Docker
S13	Automotive end-user service mgmt. system	Rewrite & Extension	private cloud	30	7	REST, AMQP	Java	Docker
S14	Retail online shop	COTS Replacement	public cloud (AWS)	∼350	∼175	REST, Kafka, Amazon SQS	Java, Clojure, Node.js, Ruby, Scala, Swift	Docker, JAR/WAR, ZIP

C2-S2, Freeway Toll Management System: The first of the three systems (all for external customers) developed by C2, a large global software and consulting enterprise, was a management and payment system for freeway tolls. The initial decentralized solution was replaced by a central Microservice-based system with extended functionality that consists of 10 Java services. One important algorithm in C++ was reused from the legacy system, but proved difficult to integrate. Since performance is a major non-functional requirement, three larger core services form the central sequential pipeline of the application. They are decoupled via Oracle Advanced Queuing. While the system fulfills several popular Microservices characteristics, it has a project character (built for external customer, very large initial workforce), a high degree of central governance, separation between dev and ops team, as well as a low degree of technological heterogeneity.

C2-S3, Automotive Problem Management System: The second system from C2 was an automotive management system for the categorization and analysis of problems. To increase maintainability, the existing monolith is replaced and extended piece by piece. The new system currently consists of 10 dockerized Java services. Since the monolith is too complex and important to be replaced at once, the two systems are operated concurrently. Additional functionality is added as new Microservices (*strangler* pattern). Some of these are rather coarse-grained, because the domain proved difficult to cut. Service communication primarily relies on RESTful HTTP. The homogeneous services are seen as easy to reuse and analyze. The system also has some project characteristics (built for an external customer who operates the system) and a medium degree of central governance with several restrictions from the customer, e.g. prohibition of refactorings due to costs.

C2-S4, Public Transport Sales System: The last system from C2 was a sales system for public transportation. An existing complex of monoliths is currently being rewritten into ∼100 Microservices structured in 15 domains. The very large project involves over 300 people from several contractors that are organized in ∼30 teams with decent autonomy for their inner architecture. There is, however, a central architecture team that provides meta guidelines and cross-cutting concerns like Docker images or authentication. Service granularity is very different, because teams and contractors tend to cut very differently. The architecture team tries to harmonize this via frequent cutting and merging of services. Prevalent languages are Java with Spring Boot, and Node.js for frontend communication (*backends for frontends* pattern). Architect P4 remained skeptical about the future handover to the reduced maintenance workforce, if service granularity and technology usage are not harmonized.

C3-S5, Business Analytics & Data Integration System:

The small software and consulting company C3 develops a business analytics system for big data that is focused on data integration and analysis via data mining or linguistic algorithms. We interviewed architect P5 and lead developer P6. For the central backend, six Microservices have been developed in a complete greenfield approach. Except for one Scala service, all are written in Java using Spring Boot. Services communicate via Kafka topics in a decoupled way. Docker is used to allow easy on-premise deployments for customers (because of sensitive data) as well as internal SaaS hosting. Building the necessary expertise with new technology was described as a challenge. While there is some alignment with the 12-factor app principles, the degree of automation is rather low. However, a CI/CD pipeline is currently implemented to decrease cycle time and to increase software quality.

C4-S6, Automotive Configuration Management System: The software services company C4 is in the process of rewriting a very large monolithic configuration management system for an automotive customer. Even though architect P7 wanted to avoid too fine-grained services, there are already 60 of them, most of them developed in Java. Despite this strong Java focus, S6 inhibits a medium degree of diversity, due to different middleware, means of communication, or frontend technology. The development team also operates the system and so far no complete hand-over is planned. A high degree of automation is used (GitLab CI), although continuous deployment will be implemented at a later stage. Teams have a medium amount of autonomy. However, there is also central governance, e.g. concerning the placing of new functionality. Moreover, the customer wants to be closely involved and emphasized his power to override decisions.

C4-S7, Retail Online Shop: The second system developed by C4 is an online shop for a large retailer. This newly developed eCommerce platform replaces and extends an existing commercial off-the-shelf (COTS) product. So far, the nearly 200 involved people have created ∼250 services that are structured into domains and products. Most of these are written in Java based on Spring Boot, but there are also instances of Node.js, Go, and Kotlin. The default communication mechanism is RESTful HTTP. Currently, most deployment artifacts are JAR or WAR files, but the move to Docker containers is ongoing. A dedicated DevOps team implemented CI/CD pipelines. Later on, this team was integrated into existing development teams. However, operations is handled by the customer using a multi-cloud strategy. S7 has a minor project character, even though the developers continue to support their services after the move to production. Teams have a medium amount of autonomy and the agile principle of "individuals and interactions over processes and tools" is generally followed. There is, however, central governance for macro architecture, security, or quality assurance.

C5-S8, IT Service Monitoring Platform: The global IT services provider C5 develops a monitoring solution with an operations dashboard. Lead developer P9 described the initial architecture as a RESTful monolith that is currently migrated towards a more fine-grained architecture. The preparation for Kubernetes and the implementation of 12-factor app guidelines were seen as difficult. So far, nine services have been created while the central API still makes up the largest one. Most are written in Java with Spring, while a new larger service relies on Node.js. The hitherto exclusively used RESTful HTTP protocol was lately accompanied by more reliable AMQP messaging. A medium degree of technological diversity is also indicated by the use of four different databases. Jenkins is used for CI/CD pipelines, but production deployments still involve manual work. During the migration, there are separate dev, ops, and DevOps teams that are in close collaboration.

C6-S9, Hotel Search Engine: Developer P10 and data engineer P11 both talked with us about S9, an online search platform for hotels. The monolithic application core has been rewritten by a small team and now comprises six Java services operated by the same development team. To guarantee low response times and fluent end-user interactions while waiting for search results, gRPC streaming was used for service communication. Getting production-ready with gRPC was described as painful by P10, but worth the effort in retrospect. Additionally, Kafka is used for event sourcing to supply state to each service at start-up. Teams at C6 have a high degree of autonomy and choose their own technologies and development model. The minimum degree of central governance is focused on a common IT strategy. Data engineer P11, however, perceived inter-team collaboration as well as high-level organizational alignment as suboptimal. Developer P10, on the other hand, mentioned that the new Microservice-based system had much better scalability, maintainability, as well as operability.

C6-S10, Hotel Management Suite: The second system from C6 was a management suite for hoteliers with several modular products. The suite followed the Self-Contained System (SCS) paradigm and was developed as 20 coarse-grained and vertically cut services that included their own UI. While this follows a functional or feature-oriented decomposition, Domain-Driven Design (DDD) was not used due to its perceived complexity. PHP was the primary language for S10 with two services being written in Java. Architect P12 saw Java as unfit to be used in a polyglot environment, since it would pose integration difficulties and development overheads. To avoid low performance due to several service requests in the SPA frontend, a GraphQL middleware has been introduced (*backends for frontends* pattern). While P12 in principle saw the team autonomy at C6 as positive, he also made negative experiences with teams choosing technologies just for the sake of being new. Besides an established basic CI/CD pipeline not many additional DevOps practices were used.

C7-S11, Public Transport Management Suite: C7 develops a product suite for managing the whole process of public transportation (S11). DevOps engineer P14 described the ongoing modernization of the whole suite while architect P13 specifically focused the human resource management product (S11a). The nearly 30 years old ecosystem consists of a diverse mix of legacy and modern technology with large Java EE parts. Current modernization efforts aim to standardize and automate deployment processes and to migrate to a more fine-grained architecture. Even though Microservices are desirable for C7, P13 did not see this as a goal for the HR management part (S11a). The Java EE deployment monolith does not hold up to most characteristics of service-based systems (e.g. platform-agnostic interfaces or standardized data formats). Moreover, 50% of development efforts were required for modernization in too long cycles. Still, P13 saw no major issues that would warrant a Microservices migration. S11a is a good example that Microservices are not a silver bullet solution.

C8-S12, Retail Online Shop: To improve time to market, retailer C8 replaced their COTS online shop (SAP Hybris) with a custom eCommerce platform developed in-house as Microservices which drastically improved reliability. The omnichannel online shop consists of $\sim$45 services organized in five domains with several subdomains or "verticals". The decomposition relied on DDD and techniques like event storming. There is a high degree of decentralization and heterogeneity. The autonomous teams choose their own languages and technologies. Data exchange between services relies on replication to guarantee independence. Kafka messaging is used a lot for communication within verticals, but is prohibited for cross-domain communication to avoid coupling. A very high degree of automation with continuous deployment pipelines ensures that every accepted commit goes live without manual efforts. Developers operate the services themselves and have on-call duty. Architects like P15 perform basic central governance for strategy, modernization, or macroarchitecture, but see themselves more as coaches.

C9-S13, Automotive End-User Services Management System: Software and IT services company C9 migrated an system of an external automotive customer to a more cloud-native architecture. The system handles management and payment of end-user services in the car. The monolith was moved from traditional WebSphere to Liberty running on Docker, while service cutting was postponed. Instead, the *strangler* pattern was used for ongoing development. Currently, Java is the exclusive language for the seven services. During the migration, SOAP and JMS were removed in favor of RESTful HTTP and AMQP. Teams were organized in cross-functional *squads* following the Spotify model, but with limited autonomy. Central governance and the customer prescribed most decisions. Jenkins pipelines provided a medium degree of automation and there was some alignment with 12-factor app principles. Politically motivated changes by the customer (move from Cloud Foundry to Kubernetes, switching providers) made mastering the new technologies challenging.

C10-S14, Retail Online Shop: Similar to the other two online shops (S7, S12), retailer C10 replaced Intershop Enfinity with a custom solution that was initially based on Self-Contained Systems (SCS) with a strong Java focus. Over the years, the architecture gradually evolved towards Microservices accompanied by a variety of technologies and languages. Teams have complete autonomy and employ various AWS offerings to host their services (e.g. ElasticBeanstalk, Fargate, Lambda functions). P17 perceived this strong decentralization and heterogeneity as overall beneficial, even though he acknowledged associated risks. The optimal solution would be to empower teams while making them simultaneously responsible for their actions. This is achieved by a very high degree of automation in combination with a strong DevOps mindset ("you build it, you run it"). Central governance involves IT strategy and general architectural guidelines only, not even the exact number of services (150–200) is documented in this very large decentralized ecosystem.

IV. Results and Discussion

We aggregated and summarized our findings from the analysis of the individual cases described above. In the following subsections, we present the results for each research question.

A. Service Technology (RQ1)

One important technological aspect of service orientation is communication. In our interviews, the de facto standard was **RESTful HTTP**. Even though it was not the primary protocol in each of the 14 cases, it existed in all of them, sometimes for minor interfaces. Participants named interoperability, technology independence, and loose coupling as advantages, even though most participants that used REST felt no need to justify this decision. Some participants saw direct synchronous RESTful communication between services as harmful (P5, P6, P15) and relied more on messaging to decouple services further, which P6 saw as follows: "*We also have some REST-based communication between services, which is not 100% clean. In some cases, we had to choose between performance or clear data ownership, so we compromised.*" Kafka was the preferred messaging solution followed by AMQP. In one case, gRPC was chosen to replace REST, because its streaming nature was seen as more efficient and end-user friendly (P10). Reactive Microservices and event sourcing were used by some participants (P10, P15, P17). No new system relied on SOAP for communication. It was sometimes simply kept to integrate with legacy systems.

For the deployment of services, our interviewees most often relied on **Docker containers** (11 of 14 cases). In the only three cases were it was not yet used (S1, S7, S11), the move to Docker is planned. An exception was C10: While it uses Docker for a lot of services, it is not a standard and a variety of deployment artifacts is in use. Overall, the operability and portability of Docker is valued very highly. Two participants also made positive experiences with Cloud Foundry (P8, P15) that was described as more developer-friendly as Kubernetes which was generally seen as powerful but complex. Even though there is the option for diverse languages with Microservices, **Java** is used in each of our 14 cases, in several even (almost) exclusively or in combination with other JVM languages (S1, S2, S3, S5, S6, S13). Reasons are the availability of skilled developers plus mature frameworks like Spring and a solid tool ecosystem. One participant advocated for going "all-in" with Java or not using it at all, since the coexistence with other languages is seen as problematic (P12): "*Java is not generally bad for me, but if you choose Java, take it for everything to leverage the existing* *ecosystem. Then it's awesome. But mixing it with something else is completely the contrary of awesome.*"

In nine cases, **Single Page Applications** (SPA) with Angular, React, or Vue.js are used as end-user frontends (S1, S3, S4, S5, S6, S7, S8, S10, S14). However, some participants saw SPAs with Microservices as problematic. Multiple service requests per page could lead to large amounts of business logic for data integration in client-side code or low performance with ultimately bad user experience. Some participants (P15, P17) therefore favored dynamic server-side rendering like Server Side Includes (SSI) or Edge Side includes (ESI) while others employ the *backends for frontends* pattern (P4, P12) with e.g. GraphQL as a middleware. **Serverless/FaaS** is so far not seen as a viable future option in most presented cases. Reasons are fear of vendor lock-in, high costs for constant workloads, focus on request-response interactions, slow start-up, or immaturity of the technology. Some participants also see it as unfitting for complex custom solutions. Others highlight the need for periods of technological or organizational stability before such a disrupting move. It would also be often difficult to explain the FaaS mindset and operation model to external customers or management. The few participants that think about using it (P7, P8, P16) or already do so (P15, P17) are very aware of the pros and cons and choose their use cases very selectively.

B. "Pure" Microservices? (RQ2)

While a number of characteristics [3], [4] are popular with Microservices, not every implementation will adhere to all of them. We therefore wanted to analyze which characteristics were commonly neglected and for what reasons. While the criteria *lightweight communication*, *design for failure*, and *evolutionary design* were generally followed (albeit to varying degrees), the situation looks different for other postulated attributes. When looking at **DevOps practices and automation**, only 5 of the 14 cases strictly followed the *you build it, you run it* principle (S1, S6, S9, S12, S14). For the remaining nine systems, operations was done by different people, even though collaboration was often very close. In most cases where IT service providers developed a system for an external customer, operations was done by the customer or even a different company (S2, S3, S4, S7, S13). The degree of test, build, and deployment automation was also very different. While most participants reported a decent degree of automation and the usage of CI/CD pipelines, fully automated continuous deployment existed in only 3 of the 14 cases (S9, S12, S14). P15 described it as follows: "*Every commit goes live via continuous deployment. This obviously requires a very high test coverage.*" Some participants mentioned alignment with the 12-factor app guidelines (S5, S9, S13). Finally, several participants described the implementation of CI/CD pipelines as challenging (P2, P3, P5, P6, P9, P14). Finding qualified DevOps engineers would be difficult (P1, P9, P12).

While the **products, not projects** criterion was followed for all internally developed systems, the systems developed for external customers (S2, S3, S4, S6, S7, S13) had varying degrees of project characteristics. In several cases, a large team was assembled for the initial development and later reduced. For some larger projects, several contractors were involved or the system passed through several migration project phases until transitioning into a mode similar to continuous product development. Except for S6 though, someone else was always responsible for the operation of the system.

The project nature also influenced the **decentralization** characteristic. Naturally, these systems (especially the larger ones) were developed with a higher degree of central governance and coordination, even though in some cases teams had a decent amount of autonomy for their inner architecture. 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 amount of guidelines and rules for service creation.*" For several systems (S2, S3, S6, S13), the customer also made concrete initial specifications or prescribed technological or architectural decisions. The internal development of the eight remaining systems was organized in a rather decentralized way, even though this was only partially applicable to the smaller products with just one development team (S1, S5, S8). A very high degree of decentralization and team autonomy only existed at three of the companies (C6, C8, C10) and always in combination with the *you build it you, you run it* philosophy.

The different degree of decentralization also lead to different levels of **technological heterogeneity**. In general, technological diversity was not as large as Microservices proponents would suggest, even though participants consciously left the possibility for new languages or technology open. In most cases, the possibility is not used though, exceptions being C6 and the two retailers C8 and C10 that go "all-in" on decentralization and heterogeneity. Several companies reported problems to find skilled developers familiar with Microservices technologies. Learning efforts were described as high (P1, P3, P5, P6, P9, P14, P16). Some participants were worried about teams choosing a new language, framework, or even the complete architectural style of Microservices just because it is en vogue and not because it makes sense for the use case at hand (P3, P5, P11, P12, P16). Others deal with it by specifying one candidate per category (e.g. one relational DB, one NoSQL DB) or by requesting solid arguments for new technology. Architect P12 stressed the danger of starting the selection of new technologies without talking about the underlying problems first. For him, the "*hipsterization of technology*" had to stop before decentralization could work in sustainable fashion.

Lastly, **service granularity** was also very heterogeneous, even within the same system. Team autonomy and the usage of different contractors in large projects increased this heterogeneity. There was often one or several core service(s) larger than the rest. Coarse-grained services were also used for reasons of performance, to avoid dependencies between services, for domains that are hard to cut, or simply to avoid very large numbers of services. While half of the analyzed systems had 10 or less services, there were also very large systems with more than 100 fine-grained services (S4, S7, S14). In general, participants consciously aimed for smaller numbers and more coarse-grained services in a lot of cases, or as P7 described it: "*If we went all the way with Microservices, we probably would have to create a separate service for each business domain entity. That would be too much, we can't go that route.*" Even though Domain-Driven Design (DDD) is often cited in literature to achieve service cuts [12], only three participants reported its explicit usage (P4, P15, P17).

C. Microservices and Software Quality (RQ3)

The impact on the overall software product quality (ISO 25010) was mostly reported as positive. Figure 1 shows an overview, in which we summarized the interviewees assessments resulting in one mention per attribute which can be either neutral, negative, or positive (17 mentions per attribute).

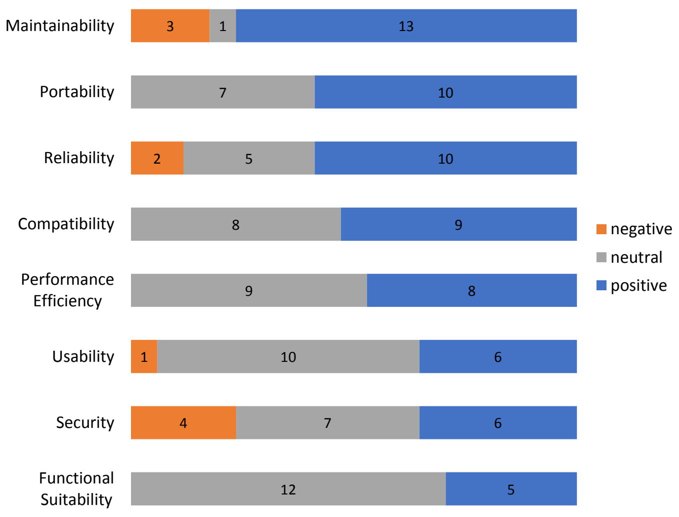

Fig. 1. Perceived Impact of Microservices on Software Quality (ISO 25010)

In many of the migration cases, the *"killer attribute"* (P17) **maintainability** was either the main migration driver or reported as the most obvious improvement over the monolith (P1, P2, P5, P7, P10, P12, P17). Microservices entailed modularity by design, which facilitated agility through independent release cycles (P6, P15). *Modifiability* was reported as much improved due to small and independent units (P1, P3, P8, P15, P17). *Reusability* was seen as rather ambiguous. One participant explicitly warned against sharing libraries between services (P7). While *testability* of a single service would be greatly improved (P3, P14, P15), the increased outer complexity could prevent the same on system level (P3). The impact on *analyzability* was assessed similarly (P17), in this context P11 criticized the consequences of separated code repositories per services. Architect P17 appropriately summarized the general perception of **reliability**: *"With 150 Microservices, there is always something down. Users just don't necessarily notice it, since the vast majority of the system is available."* Several participants agree with this perception (P1, P7, P10, P15) while others remain skeptical (P2, P4, P5, P8, P12). Architect P12 emphasized the significant effort: *"They say that people move to Microservices for reliability. But you need a lot of tools and knowledge to be more reliable than your old monolith. The complexity has definitely increased, even though some stuff is better."* Nonetheless, reliability was perceived as predominantly positive once those obstacles are overcome (P10, P15) or features like Kubernetes liveliness probes (P9) were implemented.

According to our interviewees, **portability** would generally profit from container technologies like Docker and Kubernetes (P3, P5, P7, P16). Some participants tended to attribute it only to containerization (P9, P12) or tools like Ansible (P1), not in particular to the architectural style. Other participants highlighted better *installability* (P2, P3, P4) which allowed to effortlessly change the platform (P2, P5). P2 described it as follows: *"In theory, we could run the system in an arbitrary cloud or on-premise, even though this is more due to containers and not so much Microservices. Portability has definitely increased."* Moreover, adapting and replacing modules became easier, but would not necessarily apply for the entire system (P2, P17). The influence on **compatibility** was also commonly perceived as beneficial, mostly due to standard communication protocols. Architect P4 reported compatibility as one of the main drivers for Microservices to unify distribution channels of their product. In particular, the ability to co-exist and integrate with legacy systems was positively perceived and a strong requirement for many companies during the migration phase (P1, P8, P10, P12).

With regards to **performance efficiency**, the consensus on scalable Microservices was confirmed by many participants as the most notable advantage (P2, P7, P9, P15, P17). Architect P17 commented on his experiences as follows: *"Constant and fast response times while increasing load by a factor of 10 or 100: I believe this will only work in a Microservices architecture."* On the other hand, efficient *resource utilization* could suffer from high redundancy to some extent (P3, P15). Architect P3 saw benefits in choosing the best performing technology for a single service, but many others do not perceive or expect changes for performance efficiency (P4, P6, P8, P10, P11, P14, P16).

Usability was generally seen as not impacted, or as P7 termed it: *"In general, end-user experience shouldn't be affected by Microservices, except for maybe increased performance."* One participant highlighted the suitability for easy prototyping or A/B testing (P15). Some companies in the process of migrating used this opportunity to improve their user experience with a new frontend (P1, P12). *Operability* in turn was described as challenging due to the variety of involved components and technologies, but would eventually improve with a high degree of automation (P1, P2) or very homogeneous services (P3). The positive mentions mainly refer to this aspect of usability. A similar perception was observed for **functional suitability**, which many regard as not impacted by switching to Microservices (P3, P7, P9, P12, P15, P16, P17). Some were still in a migration phase and had not migrated all functionality yet, but commonly expected a higher *appropriateness* and *correctness* due to a better overview of the individual services (P4, P6, P8, P10).

Security was perceived as two-fold by our participants. One group appreciated the possibility to apply individual security settings to services depending on their criticality (P2, P4, P15). In one case, it helped to ensure conformance with the General Data Protection Regulation (P4). Still, no participant explicitly mentioned that Microservices would ease securing a system. The other group stressed the increased attack surface (P3, P7, P12), or as architect P12 termed it: *"Before, you had to secure one door, now you have 20."* To tackle this challenge, it would be beneficial to follow existing reference implementations (P5) or use the *API gateway* pattern (P8).

V. Related Work

We identified several related papers that investigate Microservices in industry practice. Soldani et al. [13] conducted a systematic grey literature review on pains and gains of Microservices. The authors investigated 51 industrial studies in the form of whitepapers, blog posts, and videos published from 2014 to 2017. The focus was on technical and operational aspects of Microservice-based architectures covering design, development, and operation. The authors concluded that the understanding of this field is quite mature in industry already, but not yet in academia. This confirms the relevance our work which aims to analyze current practice.

Several studies primarily address the migration aspect, e.g. Luz et al. [14] performed a long term observation study on Microservices adoption. The authors discussed benefits and challenges as well as motivations and technical decisions made. The experience report contains in-depth findings of three institutions in the same domain. Di Francesco et al. [15] aimed to achieve generalizable insights on migration activities and challenges of adopting Microservices by consulting 18 practitioners from 16 IT companies. Results are presented as conducted activities and faced challenges specifically for each phase in a migration process. Taibi et al. [16] went one step further by deriving a migration process framework out of 21 interviews with experienced practitioners. While [15] and [16] interviewed groups of similar size, they focused on migration-related activities, motivations, benefits, and challenges. Similar aspects are covered in the study by Knoche et al. [8]. In their survey among 71 participants they investigated drivers, obstacles, and superior strategies for introducing Microservices in the German industry. This very detailed analysis can be seen as a similar yet quantitative study among practitioners with the same cultural background as our interviewees.

VI. Threats to Validity

Several limitations have to be mentioned with this study. With respect to **internal validity**, there is a small possibility that participants did not reveal their true opinions in some cases. While this is a common threat in survey-based research, we believe the risk for our study to be small. The discussed topics were not very sensitive and most participants were not afraid to talk about negative sides of their projects. This was strengthened by the guaranteed confidentiality and anonymity. Similarly, participants could have misunderstood questions or concepts, therefore providing incorrect answers. To address this, we defined important concepts before the corresponding topics and used clarifying questions if the usage of terms seemed fuzzy. Several participants also asked questions, if terms were not clear to them, e.g. certain quality attributes. To ensure interpretation validity and limit researcher bias, both moderators proofread every transcript and participants were strongly encouraged to correct wrong or unclear statements in the transcripts. Nonetheless, there remains the small possibility that we misinterpreted some of the more metaphorical statements. Moreover, summary and aggregation of results in qualitative studies rely on the subjective perception of researchers. We tried to limit this bias by critically discussing all findings between the moderators.

With respect to **external validity**, we cannot generalize distributions from the 14 cases, e.g. the industry usage of RESTful HTTP. Since this is a qualitative study, we focus on participants' rationales and the relations between concepts. Another limitation for generalizability could be that our participants were exclusively based in Germany. Moreover, we tried to achieve a certain degree of heterogeneity with company domains and sizes, but nonetheless had nine participants from software and IT services companies (∼52%). Only in the case of C2, we applied a random sampling (three out of seven candidates) to reduce selection bias. All other participants were personal industry contacts or suggested by them.

VII. Conclusion

We conducted 17 semi-structured interviews with experienced software professionals from 10 different companies. The German-based participants talked with us about 14 service-based systems, the adherence to Microservices characteristics, and the impact of Microservices on software quality. The analysis revealed that RESTful HTTP and Docker containers were valued for their interoperability and portability while Java was chosen because of many available developers. Furthermore, several Microservices characteristics (e.g. decentralization) were not or only partially followed, especially when the system was created for an external customer. Participants also expressed valid arguments why they generally build fewer and more coarse-grained services, which seems to blur the line between service- and Microservice-based systems even further. The impact of Microservices on software quality was predominantly rated as positive or neutral. Maintainability received most positive mentions while only security was seen as controversial. Researchers creating industry-focused methods should take these insights into account. Future research could perform deeper analysis of discovered rationales and trends to make them more actionable for the creation of methods.

Acknowledgment

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program "Services Computing" (http://www.services-computing.de/?lang=en).

References

[1] M. Papazoglou, “Service-oriented computing: concepts, characteristics and directions,” in *Proceedings of the 7th International Conference on Properties and Applications of Dielectric Materials (Cat. No.03CH37417)*. IEEE Comput. Soc, 2003.

[2] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[3] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 1st ed. O’Reilly Media, 2015.

[4] M. Fowler, “Microservices Resource Guide,” 2015. [Online]. Available: http://martinfowler.com/microservices

[5] O. Zimmermann, “Microservices tenets,” *Computer Science - Research and Development*, vol. 32, no. 3-4, jul 2017.

[6] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: Yesterday, Today, and Tomorrow,” in *Present and Ulterior Software Engineering*. Cham: Springer International Publishing, 2017.

[7] G. Schermann, J. Cito, and P. Leitner, “All the Services Large and Micro: Revisiting Industrial Practice in Services Computing,” in *Lecture Notes in Computer Science*, ser. Lecture Notes in Computer Science, A. Norta, W. Gaaloul, G. R. Gangadharan, and H. K. Dam, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, vol. 9586.

[8] H. Knoche and W. Hasselbring, “Drivers and Barriers for Microservice Adoption – A Survey among Professionals in Germany,” *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, vol. 14, no. 1, pp. 1–35, 2019.

[9] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering*, vol. 14, no. 2, apr 2009.

[10] C. B. Seaman, “Qualitative Methods,” in *Guide to Advanced Empirical Software Engineering*. London: Springer London, 2008.

[11] S. Hove and B. Anda, “Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research,” in *11th IEEE International Software Metrics Symposium (METRICS’05)*, no. Metrics. IEEE, 2005.

[12] F. Rademacher, J. Sorgalla, and S. Sachweh, “Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective,” *IEEE Software*, vol. 35, no. 3, may 2018.

[13] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, “The pains and gains of microservices: A Systematic grey literature review,” *Journal of Systems and Software*, vol. 146, no. September, dec 2018.

[14] W. Luz, E. Agilar, M. C. de Oliveira, C. E. R. de Melo, G. Pinto, and R. Bonifácio, “An experience report on the adoption of microservices in three Brazilian government institutions,” in *Proceedings of the XXXII Brazilian Symposium on Software Engineering - SBES ’18*, vol. 10. New York, New York, USA: ACM Press, 2018.

[15] P. Di Francesco, P. Lago, and I. Malavolta, “Migrating Towards Microservice Architectures: An Industrial Survey,” in *2018 IEEE International Conference on Software Architecture (ICSA)*, no. August. IEEE, apr 2018.

[16] D. Taibi, V. Lenarduzzi, and C. Pahl, “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation,” *IEEE Cloud Computing*, vol. 4, no. 5, sep 2017.