Supporting Architectural Decision Making on Data Management in Microservice Architectures

Evangelos Ntentos[1], Uwe Zdun[1], Konstantinos Plakidas[1], Daniel Schall[2], Fei Li[2], and Sebastian Meixner[2]

[1] University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Austria
firstname.lastname@univie.ac.at
[2] Siemens Corporate Technology, Vienna, Austria
firstname.lastname@siemens.com

Abstract. Today many service-based systems follow the microservice architecture style. As microservices are used to build distributed systems and promote architecture properties such as independent service development, polyglot technology stacks including polyglot persistence, and loosely coupled dependencies, architecting data management is crucial in most microservice architectures. Many patterns and practices for microservice data management architectures have been proposed, but are today mainly informally discussed in the so-called "grey literature": practitioner blogs, experience reports, and system documentations. As a result, the architectural knowledge is scattered across many knowledge sources that are usually based on personal experiences, inconsistent, and, when studied on their own, incomplete. In this paper we report on a qualitative, in-depth study of 35 practitioner descriptions of best practices and patterns on microservice data management architectures. Following a model-based qualitative research method, we derived a formal architecture decision model containing 325 elements and relations. Comparing the completeness of our model with an existing pattern catalog, we conclude that our architectural decision model substantially reduces the effort needed to sufficiently understand microservice data management decisions, as well as the uncertainty in the design process.

1 Introduction

Microservice architectures [14,20] have emerged from established practices in service-oriented computing (cf. [15,18,21]). The microservices approach emphasizes business capability-based and domain-driven design, development in independent teams, cloud-native technologies and architectures, polyglot technology stacks including polyglot persistence, lightweight containers, loosely coupled service dependencies, and continuous delivery (cf. [12,14,20]). Some of these tenets introduce substantial challenges for the data management architecture. Notably, it is usually advised to decentralize all data management concerns. Such an architecture requires, in addition to the already existing non-trivial design challenges

intrinsic in distributed systems, sophisticated solutions for data integrity, data querying, transaction management, and consistency management [14,20,15,18].

Many authors have written about microservice data management and various attempts to document microservice patterns and best practices exist [18,8,12,15]. Nevertheless, most of the established practices in industry are only reported in the so-called "grey literature", consisting of practitioner blogs, experience reports, system documentations, etc. In most cases, each of these sources documents a few existing practices well, but usually they do not provide systematic architectural guidance. Instead the reported practices are largely based on personal experience, often inconsistent, and, when studied on their own, incomplete. This creates considerable uncertainty and risk in architecting microservice data management, which can be reduced either through substantial personal experience or by a careful study of a large set of knowledge sources. Our aim is to complement such knowledge sources with an unbiased, consistent, and more complete view of the current industrial practices than readily available today.

To reach this goal, we have performed a qualitative, in-depth study of 35 microservice data practice descriptions by practitioners containing informal descriptions of established practices and patterns in this field. We have based our study on the model-based qualitative research method described in [19]. It uses such practitioner sources as rather unbiased (from our perspective) knowledge sources and systematically codes them through established coding and constant comparison methods [6], combined with precise software modeling, in order to develop a rigorously specified software model of established practices, patterns, and their relations. This paper aims to study the following research questions:

- **RQ1** What are the patterns and practices currently used by practitioners for supporting data management in a microservice architecture?
- **RQ2** How are the current microservice data management patterns and practices related? In particular, which architectural design decisions (ADDs) are relevant when architecting microservice data management?
- **RQ3** What are the influencing factors (i.e., decision drivers) in architecting microservice data management in the eye of the practitioner today?

This paper makes three major contributions. First, we gathered knowledge about established industrial practices and patterns, their relations, and their decision drivers in the form of a *qualitative study on microservice data management architectures*, which included 35 knowledge sources in total. Our second contribution is the codification of this knowledge in form of a *reusable architectural design decision (ADD) model* in which we formally modeled the decisions based on a UML2 meta-model. In total we documented 9 decisions with 30 decision options and 34 decision drivers. Finally, we *evaluated the level of detail and completeness of our model* to support our claim that the chosen research method leads to a more complete treatment of the established practices than methods like informal pattern mining. For this we compared to the by far most complete of our pool of sources, the *microservices.io* patterns catalog [18], and are able to show that our ADD model captures 210% more elements and relations.

The remainder of this paper is organized as follows: In Section 2 we compare to the related work. Section 3 explains the research methods we have applied in our study and summarizes the knowledge sources. Section 4 describes our reusable ADD model on microservice data management. Section 5 compares our study with *microservices.io* in terms of completeness. Finally, Section 6 discusses the threats to validity of our study and Section 7 summarizes our findings.

2 Related Work

A number of approaches that study microservice patterns and best practices exist: The *microservices.io* collection by Richardson [18] addresses microservice design and architecture practices. As the work contains a category on data management, many of them are included in our study. Another set of patterns on microservice architecture structures has been published by Gupta [8], but those are not focused on data management. Microservice best practices are discussed in [12], and similar approaches are summarized in a recent mapping study [15]. So far, none of those approaches has been combined with a formal model; our ADD model complements these works in this sense.

Decision documentation models that promise to improve the situation exist (e.g. for service-oriented solutions [21], service-based platform integration [13], REST vs. SOAP [16], and big data repositories [7]). However, this kind of research does not yet encompass microservice architectures, apart from our own prior study on microservice API quality [19]. The model developed in our study can be classified as a reusable ADD model, which can provide guidance on the application of patterns [21]. Other authors have combined decision models with formal view models [9]. We apply such techniques in our work, but also extend them with a formal modeling approach based on a qualitative research method.

3 Research Method and Modelling Tool

Research Method. This paper aims to systematically study the established practices in the field of architecting data management in microservice architectures. We follow the model-based qualitative research method described in [19]. It is based on the established Grounded Theory (GT) [6] qualitative research method, in combination with methods for studying established practices like pattern mining (see e.g. [4]) and their combination with GT [10]. The method uses descriptions of established practices from the authors' own experiences as a starting point to search for a limited number of well-fitting, technically detailed sources from the so-called "grey literature" (e.g., practitioner reports, system documentations, practitioner blogs, etc.). These sources are then used as unbiased descriptions of established practices in the further analysis. Like GT, the method studies each knowledge source in depth. It also follows a similar coding process, as well as a constant comparison procedure to derive a model. In contrast to classic GT, the research begins with an initial research question, as in Charmaz's constructivist GT [3]. Whereas GT typically uses textual analysis, the method

uses textual codes only initially and then transfers them into formal software models (hence it is model-based).

The knowledge-mining procedure is applied in many iterations: we searched for new knowledge sources, applied open and axial coding [6] to identify candidate categories for model elements and decision drivers, and continuously compared the new codes with the model designed so far to incrementally improve it. A crucial question in qualitative methods is when to stop this process. Theoretical saturation [6] has attained widespread acceptance for this purpose. We stopped our analysis when 10 additional knowledge sources did not add anything new to our understanding of the research topic. While this is a rather conservative operationalisation of theoretical saturation (i.e., most qualitative research saturates with far fewer knowledge sources that add nothing new), our study converged already after 25 knowledge sources. The sources included in the study are summarized in Table 1. Our search for sources was based on our own experience, i.e., tools, methods, patterns and practices we have access to, worked with, or studied before. We also used major search engines (e.g., Google, Bing) and topic portals (e.g., InfoQ) to find more sources.

Modelling Tool Implementation. To create our decision model, we used our existing modeling tool CodeableModels[3], a Python implementation for precisely specifying meta-models, models, and model instances in code. Based on CodeableModels, we specified meta-models for components, activities, deployments and microservice-specific extensions of those, as outlined above. In addition, we realized automated constraint checkers and PlantUML code generators to generate graphical visualizations of all meta-models and models.

4 Reusable ADD model for data management in microservice architectures

In this section, we describe the reusable ADD model derived from our study [4]. All elements of the model are *emphasized* and all decision drivers derived from our sources in Table 1 are *slanted*. It contains one decision category, *Data Management Category*, relating five top-level decisions, as illustrated in Fig. 1. These decisions need to be taken for the decision contexts *all instances of* an *API*, *Service* instances, or the combination of *Data Objects* and *Service* instances, respectively. Note that all elements of our model are instances of a meta-model (with meta-classes such as *Decision*, *Category*, *Pattern*, *AND Combined Group*, etc.), which appear in the model descriptions. Each of them is described in detail below (some elements may be relevant for more than one decision, but this has been omitted from the figures for ease of presentation).

Microservice Database Architecture (Fig.2). Since most software relies on efficient data management, database architecture is a central decision in the design

[3] `https://github.com/uzdun/CodeableModels`
[4] Replication package can be found at: `https://bit.ly/2EKyTnL`

Table 1. Knowledge Sources Included in the Study

Name	Description	Reference
S1 [2]	Intro to Microservices: Dependencies and Data Sharing	https://bit.ly/2YTnolQ
S2 [1]	Pattern: Shared database	https://bit.ly/3OL1PW2
S3 [4]	Enterprise Integration Patterns	https://bit.ly/2Wr1OHC
S4 [2]	Design Patterns for Microservices	https://bit.ly/2EBmIcQ
S5 [2]	6 Data Management Patterns for Microservices	https://bit.ly/2K3YMTb
S6 [1]	Pattern: Database per service	https://bit.ly/2EDDici
S7 [2]	Transaction Management in Microservices	https://bit.ly/2XSKhWL
S8 [2]	A Guide to Transactions Across Microservices	https://bit.ly/2WpQN9j
S9 [2]	Saga Pattern – How to implement business transactions using Microservices	https://bit.ly/2WpRBuR
S10 [2]	Saga Pattern and Microservices architecture	https://bit.ly/2HF6G3G
S11 [2]	Patterns for distributed transactions within a microservices architecture	https://bit.ly/2QqZgUx
S12 [2]	Data Consistency in Microservices Architecture	https://bit.ly/2K5G79y
S13 [2]	Event-Driven Data Management for Microservices	https://bit.ly/2WlSKUs
S14 [1]	Pattern: Saga	https://bit.ly/2WpS549
S15 [2]	Managing Data in Microservices	https://bit.ly/2HYIvvY
S16 [2]	Event Sourcing, Event Logging An essential Microservice Pattern	https://bit.ly/2QusIcb
S17 [1]	Pattern: Event sourcing	https://bit.ly/2K62TOn
S18 [2]	Microservices With CQRS and Event Sourcing	https://bit.ly/2JK2IZQ
S19 [2]	Microservices Communication: How to Share Data Between Microservices	https://bit.ly/2HCR94u
S20 [2]	Building Microservices: Inter-Process Communication in a Microservices Architecture	https://bit.ly/3OOVB7U
S21 [1]	Pattern: Command Query Responsibility Segregation (CQRS)	https://bit.ly/2X8OLcM
S22 [3]	Data considerations for microservices	https://bit.ly/2WrLeav
S23 [2]	Preventing Tight Data Coupling Between Microservices	https://bit.ly/2WptQmJ
S24 [3]	Challenges and solutions for distributed data management	https://bit.ly/2wp5YkO
S25 [3]	Communication in a microservice architecture	https://bit.ly/2X7UDkT
S26 [2]	Microservices: Asynchronous Request Response Pattern	https://bit.ly/2WjAFqb
S27 [2]	Patterns for Microservices Sync vs. Async	https://bit.ly/2Ezhsqg
S28 [2]	Building Microservices: Using an API Gateway	https://bit.ly/2EA3AfA
S29 [2]	Microservice Architecture: API Gateway Considerations	https://bit.ly/2YUKWqr
S30 [1]	Pattern: API Composition	https://bit.ly/2WlVqSO
S31 [1]	Pattern: Backends For Frontends	https://bit.ly/2X9I3kQ
S32 [3]	Command and Query Responsibility Segregation (CQRS) pattern	https://bit.ly/2wltdMq
S33 [2]	Introduction to CQRS	https://bit.ly/2HYOsLm
S34 [2]	CQRS	https://bit.ly/2JKI2Rz
S35 [2]	Publisher-Subscriber pattern	https://bit.ly/2JGtqCx

[1] denotes a source taken from *microservices.io*
[2] practitioner blog
[3] Microsoft technical guide
[4] book chapter

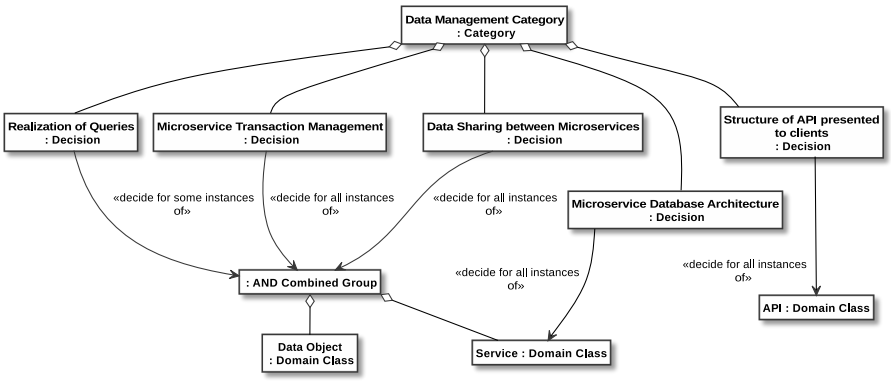

Fig. 1. Reusable ADD Model on Microservice Data Management: Overview

of a microservice architecture. Quality attributes such as performance, reliability, coupling, and scalability, need to be carefully considered in the decision making process. The simplest decision option is to choose *service stores no persistent data*, which is applicable only for services whose functions are performed solely on transient data, like pure calculations or simple routing functions. By definition, a microservice should be autonomous, loosely coupled and able to be developed, deployed, and scaled independently [12]. This is ensured by the *Database per Service* pattern [18], which we encountered, either directly or implicitly, in 33 out of 35 sources: each microservice manages its own data, and data exchange and communications with other services are realized only through a set of well-defined APIs. When choosing this option, transaction management between services becomes more difficult, as the data is distributed across the services; for the same reason making queries could become a challenge, too. Thus the optional next decisions on *Microservice Transaction Management* (see sources [S7, S8, S11]) and *Realization of Queries* [18] should be considered (both explained below). The use of this pattern may also require a next decision on the *Need for Data Composition, Transformation, or Management*. Another option, which is recommended only for special cases (e.g., when a group of services always needed to share a data object), is to use a *Shared Database* [18](see sources [S1, S19]): all involved services persist data in one and the same database.

There are a number of criteria that determine the outcome of this decision. Applying the *Database per Service* pattern in a system results in more *loosely coupled* microservices. This leads to better *scalability* than a *Shared Database* closer to the service with only transient data, since microservices can scale up individually. Especially for low loads this can reduce *performance*, as additional distributed calls are needed to get data from other services and establish *data consistency*. The pattern's impact on *performance* is not always negative: for high loads a *Shared Database* can become a bottleneck, or database replication is needed. On the other hand, *Shared Database* makes it easier to *manage transactions* and *implement queries and joins*; hence the follow-on decisions

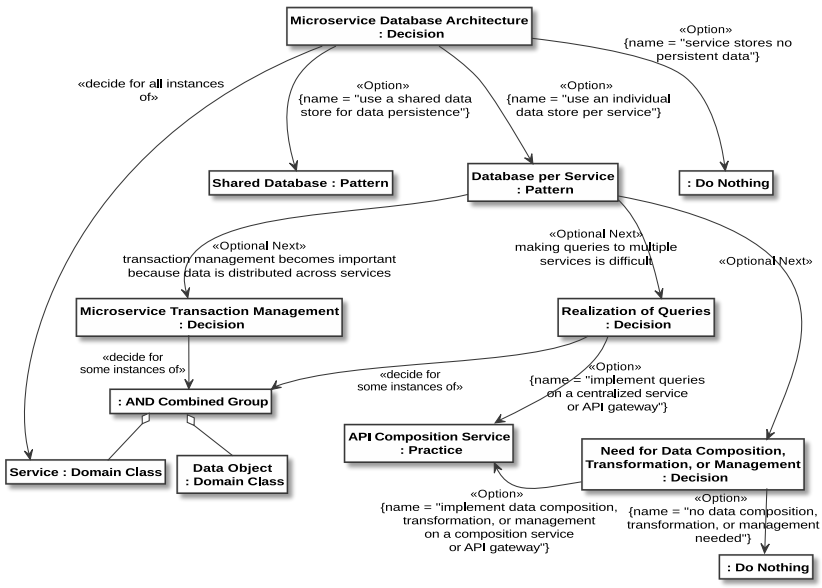

Fig. 2. Microservice Database Architecture Decision

for *Database per Service* mentioned above. Furthermore, *Database per Service* facilitates *polyglot persistence*. The *Shared Database* option could be viable only if the *integration complexity* or related challenges of *Database per Service*-based services become too difficult to handle; also, operating a single *Shared Database* is simpler. Though *Shared Database* ensures *data consistency* (since any changes to the data made in a single service are made available to all services at the time of the database commit), it would appear to completely eliminate the targeted benefits of *loose coupling*. This negatively affects both the *development and runtime coupling* and the potential *scalability*.

Structure of API Presented to Clients (Fig.3). When software is decomposed into microservices, many major challenges lie in the structure of the API. This topic has been extensively studied in our prior and ongoing work on API patterns [19]; here we concentrate only on those decision options relevant to data management. Many issues in microservice design are resolved at the API level, such as routing requests to the appropriate microservice, the distribution of multiple services, and the aggregation of results. The simplest option for structuring a system is *Clients Access Microservices Directly*: all microservices are entry points of the system, and clients can directly request the service they need (each service offers its own API endpoint to clients). However, all studied sources recommend or assume the use of the *API Gateway* pattern [18] as a common entry point for the system, through which all requests are routed. An

alternative solution, for servicing different types of clients (e.g., mobile vs. desktop clients) is the *Backends for Frontends* pattern variant [18], which offers a fine-grained API for each specific type of client. An *API Gateway* could also be realized as an *API Composition Service* [18], that is a service which invokes other microservices. Furthermore an *API Gateway* can have *Additional centralized data-related functions* (shown in Fig. 3 and discussed below as decision drivers).

The main driver affecting this decision is that *API Gateways* (and thus *API Composition Service* and *Backends for Frontends* in a more limited capacity) can provide a number of centralized services. They can work as a proxy service to *route requests* to the appropriate microservice, *convert or transform requests or data* and deliver the *data at the granularity needed by the client*, and provide the *API abstractions for the data needed by the client*. In addition, they can *handle access management* to data (i.e., authentication/authorization), serve as a *data cache*, and *handle partial failures*, e.g. by returning default or cached data. Although its presence increases the overall *complexity* of the architecture since an additional service needs to be developed and deployed, and increases *response time* due to the additional network passes through it, an *API Gateway* is generally considered as an optimal solution in a microservice-based system. *Clients Access Microservices Directly* makes it difficult to realize such centralized functions. A sidecar architecture [1] might be a possible solution, but if the service should fail, many functions are impeded, e.g. caching or handling partial failures. The same problem of centralized coordination also applies to a lesser extent to *Backends for Frontends* (centralization in each *API Gateway* is still possible). *Use API Gateway to cache data* reduces the *response time*, returning cached data faster, and increases *data availability*: if a service related to specific data is unavailable, it can return its cached data.

Data Sharing Between Microservices (Fig.4). Data sharing must be considered for each data object that is shared between at least two microservices. Before deciding how to share data, it is essential to identify the information to be shared, its update frequency, and the primary provider of the data. The decision must ensure that sharing data does not result in tightly coupled services. The simplest option is to choose *services share no data*, which is theoretically optimal in ensuring loose coupling, but is only applicable for rather independent services or those that require only transient data. Another option, already discussed above, is a *Shared Database*. In this solution services share a common database; a service publishes its data, and other services can consume it when required. A number of viable alternatives to the *Shared Database* exist. *Synchronous Invocations-Based Data Exchange* is a simple option for sharing data between microservices. *Request-Response Communication* [11] is a data exchange pattern in which a service sends a request to another service which receives and processes it, ultimately returning a response message. Another typical solution that is well suited to achieving *loose coupling* is to use *Asynchronous Invocations-Based Data Exchange*. Unlike *Request-Response Communication*, it removes the need to wait for a response, thereby decoupling the execution of the communicating services. Implementation

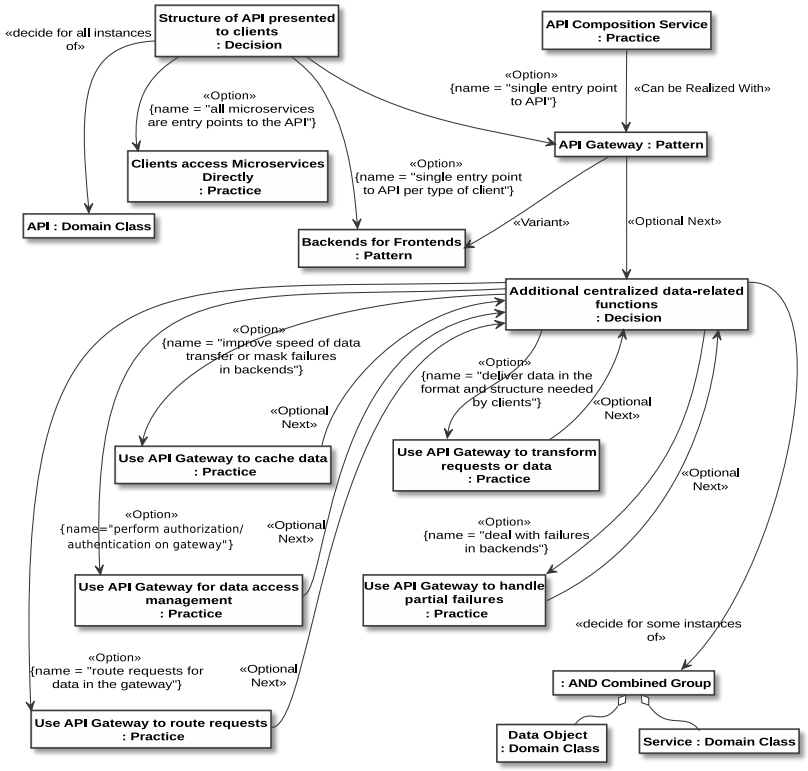

Fig. 3. Structure of API Presented to Clients Decision

of asynchronous communication leads to *Eventual Consistency*[17]. There are several possible *Asynchronous Data Exchange Mechanisms*: *Publish/Subscribe* [11], in which services can subscribe to an event; use of a *Messaging* [11] middleware; *Data Polling*, in which services periodically poll for data changes in other services; and the *Event Sourcing* [18] pattern that ensures that all changes to application state are stored as a sequence of events.

The choices in this decision are determined by a number of factors. With a *Shared Database*, the system tends to be more *tightly coupled* and *less scalable*. Conversely, an *Asynchronous Data Exchange Mechanism* ensures that the services are more *loosely coupled*, since they interact mostly via events, use message buffering for queuing requests until processed by the consumer, support *flexible* client-service interactions, or provide an explicit interprocess communication mechanism. It has minimal impact on quality attributes related to network interactions, such as *latency* and *performance*. However, *operational complexity* is negatively impacted, since an additional service must be configured and operated. On the other hand, a *Request-Response Communication* mechanism does not require a broker, resulting in a *less complex* system architecture. Despite this, in a

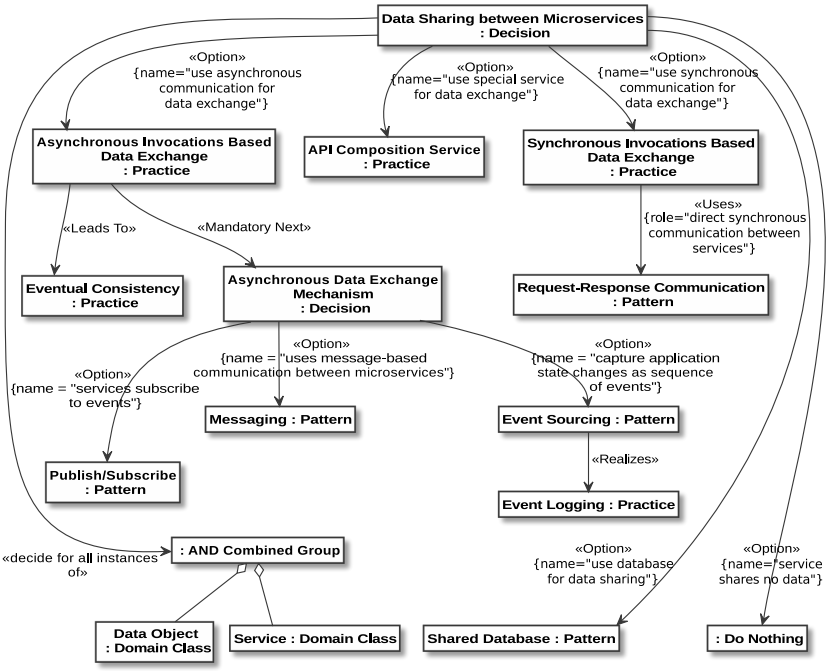

Fig. 4. Data Sharing Between Microservices Decision

Request-Response Communication-based system, the communicating services are more *tightly coupled* and the communication is less *reliable*, as they must both be running until the exchange is completed. Applying the *Event Sourcing* pattern increases *reliability*, since events are published whenever state changes, and the system is more *loosely coupled*. Patterns supporting message persistence such as *Messaging*, *Event Sourcing*, and messaging-based *Publish/Subscribe* increase the *reliability* of message transfers and thus the *availability* of the system.

Microservice Transaction Management (Fig.5). One common problem in microservice-based systems is how to manage distributed transactions across multiple services. As explained above, the *Database per Service* pattern often introduces this need, as the relevant data objects of a transaction are scattered across different services and their databases. Issues concerning transaction atomicity and isolation of user actions for concurrent requests need to be dealt with. One of the easiest and most efficient options to solve the problem of distributed transactions is to completely avoid them. This can be done through a *Shared Database* (with all its drawbacks in a microservice architecture) or by service redesign so that all data objects of the transaction reside in one microservice. If this is not possible, another option is to apply the *Saga Transaction Management* [18] pattern, where each transaction updates data within a single service, in a se-

quence of local transactions [S9]; every step is triggered only if the previous one has been completed. The implementation requires an additional decision for the *Saga Coordination Architecture*. There are two possible options for implementing this pattern: *Event/Choreography Coordination* and *Command/Orchestration Coordination* [S9]. *Event/Choreography Coordination* is a distributed coordination approach where a service produces and publishes events, that are listened to by other services which then decide their next action. *Command/Orchestration Coordination* is a centralized approach where a dedicated service informs other involved services, through a command/reply mechanism, what operation should be performed. Moreover, *Saga Transaction Management* supports failure analysis and handling using *Event Log* and *Compensation Action* practices [S12]. Implementing this pattern leads also to *Eventual Consistency*. Another typical option for implementing a transaction across different services is to apply the *Two-Phase Commit Protocol* [2] pattern: in the first phase, services which are part of the transaction prepare for commit and notify the coordinator that they are ready to complete the transaction; in the second phase, the transaction coordinator issues a commit or a rollback to all involved microservices. Here, the *Rollback* [S7] practice is used for handling failed transactions.

There are a number of criteria that need to be considered in this decision. When implementing the *Saga Transaction Management* pattern, the *Event/Choreography Coordination* option results in a more *loosely coupled* system where the services are more *independent* and *scalable*, as they have no direct knowledge of each other. On the other hand, the *Command/Orchestration Coordination* option has its own advantages: it *avoids cyclic dependencies between services*, *centralizes the orchestration of the distributed transaction*, reduces the participants' *complexity*, and makes rollbacks easier to manage. The *Two-Phase Commit Protocol* pattern is not a typical solution for managing distributed transactions in microservices, but it provides a *strong consistency* protocol, guarantees *atomicity* of transactions, and allows read-write *isolation*. However, it can significantly impair system *performance* in high load scenarios.

Realization of Queries (Fig.6). For every data object and data object combination in a microservice-based system, and its services, it must be considered whether queries are needed. As data objects may reside in different services, e.g., as a consequence of applying *Database per Service*, queries may be more difficult to design and implement than when utilizing a single data source. The simplest option is of course to implement *no queries* in the system, but this is often not realistic. An efficient option for managing queries is to apply the *Command-Query-Responsibility-Segregation* (CQRS) pattern [5]. CQRS is a process of separation between read and write operations into a "command" and a "query" side. The "command" side manages the "create", "update" and "delete" operations; the "query" side segregates the operations that read data from the "update" operation utilizing separated interfaces. This is very efficient if multiple operations are performed in parallel on the same data. The other option is to implement queries in a *API Composition Service* or in the *API Gateway*.

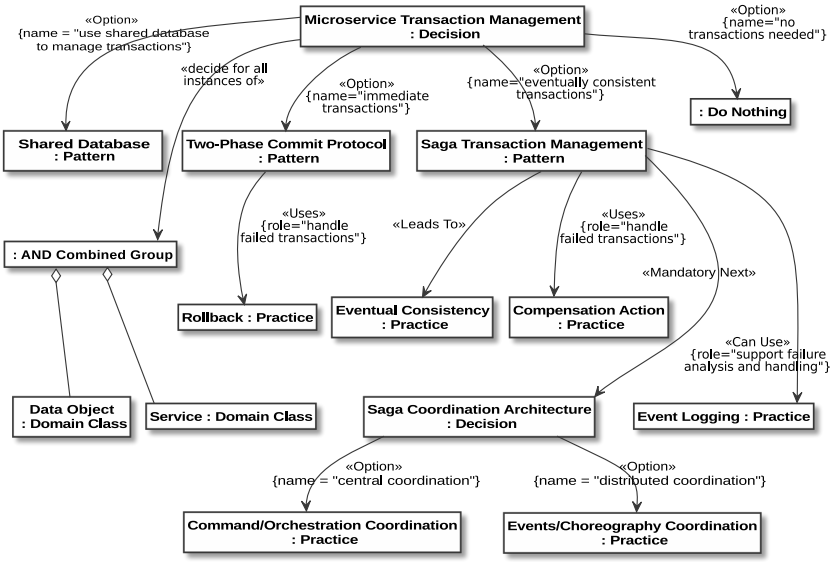

Fig. 5. Microservice Transaction Management Decision

A number of criteria determine the outcome of this decision. The *Command-Query-Responsibility-Segregation* (CQRS) option increases *scalability* since it supports independent horizontal and vertical scaling, improves *security* since the read and write responsibilities are separated. It also increases *availability*: when the "command" side is down the last data update remains available on the "query" side. Despite these benefits, using CQRS has some drawbacks: it adds significant *complexity*, and is not suitable to every system. On the other hand, implementing queries in an *API Composition Service* or *API Gateway* introduces an overhead and decreases *performance*, entails the risk of reduced *availability*, and makes it more difficult to ensure transactional *data consistency*.

5 Evaluation

We used our model-based qualitative research method described in Section 3 because informal pattern mining, or just reporting the author's own experience in a field (which is the foundation of most of the practitioner sources we encountered), entail the high risk of missing important knowledge elements or relations between them. To evaluate the effect of our method, we measure the improvement yielded by our study compared to the individual sources; specifically *microservices.io* [18], the by far most complete and detailed of our sources. This is an informally collected pattern catalog based on the author's experience and pattern mining. As such, it is a work with similar aims to this study. Of course, our formal model offers the knowledge in a much more systematically structured fashion; whereas

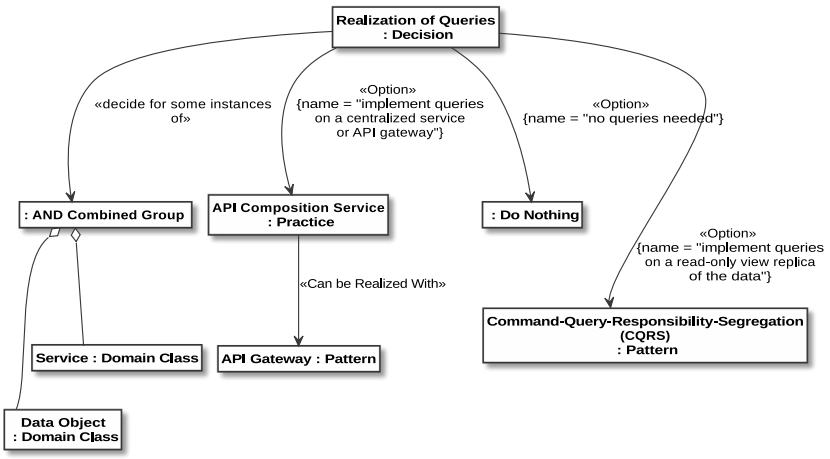

Fig. 6. Realization of Queries Decision

in the *microservices.io* texts the knowledge is often scattered throughout the text, requiring careful study of the entire text to find a particular piece of knowledge. For this reason, we believe the formal ADD model to be a useful complement to this type of sources, even if the two contain identical information.

For evaluation of our results, we studied the *microservices.io* texts in detail a second time after completing the initial run of our study, to compare which of the model elements and relations we found are also covered by *microservices.io*. Some parts of this comparison might be unfair in the sense that the *microservices.io* author does not present a decision model and covers the topic in a broad manner, so that some elements or relations may have been excluded on purpose. In addition, there may be some differences in granularity between *microservices.io* and our model, but we tried to maintain consistency with the granularity in the analysis and coding during the GT process. Considering the relatively high similarity of those *microservices.io* parts that overlap with the results of our study, and the general goal of pattern mining of representing the current practice in a field correctly and completely, we nevertheless believe that our assumption that the two studies are comparable is not totally off.

Table 2 shows the comparison for all element and relation types in our model. Only 105 of the 325 elements and relations in our model are contained in *microservices.io*: a 210% improvement in completeness has resulted from systematically studying and formally modeling the knowledge in the larger set of knowledge sources summarized in Table 1. Apart from the trivial *Categories* element type, most elements and relation types display high improvement, most notably, the *Decision driver to patterns/practices relations*. That is mainly because design options (and consequently their relations) are missing entirely. Apart from *Categories*, only the *Domain model elements* type shows no improvement, because we only considered those domain elements directly connected to our decisions

here. In the larger context of our work, we use a large and detailed microservice domain object model, but as there is nothing comparable in the microservice patterns, we only counted the directly related contexts here (else the improvement of our model would be considerably higher).

Table 2. Comparison of number of found elements and relation types our ADD model and *microservices.io*

Element and Relation Types	**ADD Model**	***microservices.io***	**Improvement**
Domain model elements	4	4	0%
Decisions	9	4	125%
Decision context relations	6	3	100%
Patterns/practices	32	15	113%
Decision to option relations	30	13	131%
Relations between patterns/practices	10	4	150%
Patterns/practices to decision relations	12	4	200%
Categories	1	1	0%
Category to decision relations	5	3	67%
Unique decision drivers	34	17	100%
Decision drivers to patterns/practices relations	182	37	392%
Total number of elements	325	105	210%

6 Threats to Validity

To increase internal validity we used practitioner reports produced independently of our study. This avoids bias, for example, compared to interviews in which the practitioners would be aware that their answers would be used in a study. This introduces the internal validity threat that some important information might be missing in the reports, which could have been revealed in an interview. We tried to mitigate this threat by looking at many more sources than needed for theoretical saturation, as it is unlikely that all different sources miss the same important information.

The different members of the author team have cross-checked all models independently to minimize researcher bias. The threat to internal validity that the researcher team is biased in some sense remains, however. The same applies to our coding procedure and the formal modeling: other researchers might have coded or modeled differently, leading to different models. As our goal was only to find one model that is able to specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study.

The experience and search-based procedure for finding knowledge sources may have introduced some kind of bias as well. However, this threat is mitigated to a large extent by the chosen research method, which requires just additional sources corresponding to the inclusion and exclusion criteria, not a specific distribution of sources. Note that our procedure is in this regard rather similar

to how interview partners are typically found in qualitative research studies in software engineering. The threat remains that our procedures introduced some kind of unconscious exclusion of certain sources; we mitigated this by assembling an author team with many years of experience in the field, and performing very general and broad searches. Due to the many included sources, it is likely our results can be generalized to many kinds of architecture requiring microservice data management. However, the threat to external validity remains that our results are only applicable to similar kinds of microservice architectures. The generalization to novel or unusual microservice architectures might not be possible without modification of our models.

7 Conclusion

In this paper, we have reported on an in-depth qualitative study of existing practices in industry for data management in microservice architectures. The study uses a model-based approach to provide a systematic and consistent, reusable ADD model which can complement the rich literature of detailed descriptions of individual practices by practitioners. It aims to provide an unbiased and more complete treatment of industry practices. To answer RQ1 we have found in 32 common patterns and established practices. To answer RQ2, we have grouped 5 top-level decisions in the data management category and documented in total 9 *decisions* with 6 *decision context relations*. Further we were able to document 30 *decision to option relations* and 22 (10+12) further relations between patterns and practices and decisions. Finally, to answer RQ3, we have found 34 *unique decision drivers* with 182 links to patterns and practices influencing the decisions. The 325 elements in our model represent, according to our rough comparison to *microservices.io*, an 210% improvement in completeness. We can conclude from this that to get a full picture of the possible microservice data management practices, as conveyed in our ADD model, many practical sources need to be studied, in which the knowledge is scattered in substantial amounts of text. Alternatively, substantial personal experiences need to be made to gather the same level of knowledge. Both require a tremendous effort and run the risk that some important decisions, practices, relations, or decision drivers might be missed. Our rough evaluation underlines that the knowledge in microservice data management is complex and scattered, and existing knowledge sources are inconsistent and incomplete, even if they attempt to systematically report best practices (such as *microservices.io*, compared to here). A systematic and unbiased study of many sources, and an integration of those sources via formal modeling, as suggested in this paper, can help to alleviate such problems and provide a rigorous and unbiased account of the current practices in a field (like presently on microservice data management practices).

Acknowledgments. This work was supported by: FFG (Austrian Research Promotion Agency) project DECO, no. 864707; FWF (Austrian Science Fund) project ADDCompliance: I 2885-N33

References

1. Sidecar pattern (2017), https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar
2. Al-houmaily, Y., Samaras, G.: Two-phase commit. In: Encyclopedia of Database Systems, pp. 3204–3209 (2009)
3. Charmaz, K.: Constructing grounded theory. Sage (2014)
4. Coplien, J.: Software Patterns: Management Briefings. SIGS, New York (1996)
5. Fowler, M.: Command and Query Responsibility Segregation (CQRS) pattern (2011), https://martinfowler.com/bliki/CQRS.html
6. Glaser, B.G., Strauss, A.L.: The Discovery of Grounded Theory: Strategies for Qualitative Research. de Gruyter (1967)
7. Gorton, I., Klein, J., Nurgaliev, A.: Architecture knowledge for evaluating scalable databases. In: Proc. of the 12th Working IEEE/IFIP Conference on Software Architecture. pp. 95–104 (2015)
8. Gupta, A.: Microservice design patterns. http://blog.arungupta.me/microservice-design-patterns/ (2017)
9. van Heesch, U., Avgeriou, P., Hilliard, R.: A documentation framework for architecture decisions. J. Syst. Softw. **85**(4), 795 – 820 (2012)
10. Hentrich, C., Zdun, U., Hlupic, V., Dotsika, F.: An Approach for Pattern Mining Through Grounded Theory Techniques and Its Applications to Process-driven SOA Patterns. In: Proc. of the 18th European Conference on Pattern Languages of Program. pp. 9:1–9:16 (2015)
11. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley (2003)
12. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. http://martinfowler.com/articles/microservices.html (2014)
13. Lytra, I., Sobernig, S., Zdun, U.: Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study. In: Proc. of WICSA/ECSA (2012)
14. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly (2015)
15. Pahl, C., Jamshidi, P.: Microservices: A systematic mapping study. In: 6th International Conference on Cloud Computing and Services Science. pp. 137–146 (2016)
16. Pautasso, C., Zimmermann, O., Leymann, F.: RESTful Web Services vs. Big Web Services: Making the right architectural decision. In: Proc. of the 17th World Wide Web Conference. pp. 805–814 (2008)
17. Perrin, M.: Overview of existing models. In: Perrin, M. (ed.) Distributed Systems, pp. 23–52. Elsevier (2017)
18. Richardson, C.: A pattern language for microservices. http://microservices.io/patterns/index.html (2017)
19. Zdun, U., Stocker, M., Zimmermann, O., Pautasso, C., Lübke, D.: Supporting Architectural Decision Making on Quality Aspects of Microservice APIs. In: 16th International Conference on Service-Oriented Computing. Springer (2018)
20. Zimmermann, O.: Microservices tenets. Computer Science - Research and Development **32**(3), 301–310 (2017)
21. Zimmermann, O., Koehler, J., Leymann, F., Polley, R., Schuster, N.: Managing architectural decision models with dependency relations, integrity constraints, and production rules. J. Syst. Softw. **82**(8), 1249–1267 (2009)