Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory

1st Apitchaka Singjai
Research Group Software Architecture
University of Vienna
Vienna, Austria
apitchaka.singjai@univie.ac.at

2nd Uwe Zdun
Research Group Software Architecture
University of Vienna
Vienna, Austria
uwe.zdun@univie.ac.at

3rd Olaf Zimmermann
University of Applied Sciences of Eastern Switzerland (OST)
Rapperswil, Switzerland
olaf.zimmermann@ost.ch

***Abstract*—Microservice API design is a critical aspect in crafting a microservice architecture. While API design in general has been studied, the specific relation of API design to design practices and models commonly used in microservice architectures is yet understudied. In particular, practitioners frequently use Domain-Driven Design (DDD) in their microservice architecture and API designs. We thus decided to study existing Architectural Design Decisions (ADDs), their solutions options, their relations, and the decision drivers in these decisions. Using the Grounded Theory research method we studied grey literature sources. In this study, we identified six ADDs with 27 decision options, numerous relations between them, and 27 decision drivers. The decisions cover mapping domain models to APIs, defining API contracts in relation to domain models, designing API resources based on domain model elements, segregation of API resources, mapping domain model links to the API, and designing the operations of an API resource.**

***Keywords*-API Design, Domain Driven Design, Grey Literature, Grounded Theory**

I. Introduction

Microservices are independently deployable, scalable, and changeable services, each having a single responsibility [1]. They typically communicate via message-based remote APIs in a loosely coupled fashion. Those remote APIs can be realized using many technologies, including RESTful HTTP, queue-based messaging, SOAP/HTTP, or remote procedure call technologies such as gRPC. A critical aspect in designing a microservice architecture is API design which includes aspects such as which microservice operations should be offered in the API, how to exchange data between client and API, how to represent API messages, and so on [2].

Microservices themselves are often identified in Domain-Driven Design (DDD) [3] artifacts. DDD is a design approach where the (business) domain is carefully modeled in software and evolved over time. Microservices and DDD have a synergistic relation. DDD concepts help to design microservices and establish their boundaries. Once established, microservice boundaries provide technical boundaries for separately modeled and evolved parts of the domain model, the *Bounded Contexts* [4]. This helps in enforcing strategic DDD design decisions e.g. modeled using a *Context Map* [4], [5].

We thus decided to investigate the current practitioner's understanding of the *relation of Microservice APIs and DDD*. In this paper, we describe a Grounded Theory based qualitative study [6], [7] for this purpose. We decided to explore literature sources representing practitioner views on this topic. According to Rainer and Williams [8] there are many benefits in using grey literature sources in software engineering research, as they promote the voice of practitioners and provide information on practitioners' contemporary perspectives on important topics relevant to practice and research. For coding processes in the Grounded Theory study, we applied text-based coding only initially and then applied UML-based modeling instead to develop a precise and consistent theory (i.e. the UML figures shown in this paper are results generated from our models).

In this paper, we focus on Architectural Design Decisions (ADDs), their decision options, their relations, and decision drivers. We set out to answer the following research questions:

- **RQ1** What are the possible ADDs and corresponding decision options in DDD-based Microservice API design?
- **RQ2** What are forces (decision drivers) relevant in those design decisions?
- **RQ3** Which relations do the decisions and decision options have?

Our result is a formal model of the ADDs [9] with decision options, forces, and relations in the field of DDD-based Microservice API design. We believe that our results can help scientists to get a better understanding of practitioner concerns in this field. Our work can also help practitioners to get an overview of the current view of other practitioners.

This article is structured as follows: First, we discuss the related work in Section II. Next, we explain our research method in Section III. In Section IV we present the detailed results of our grounded theory study, i.e. ADDs, decision options, their relations, forces, and trade-offs in each solution. Section V discusses the implications of the results for the research questions and threats to validity. Finally, in Section VI we draw conclusions.

II. Related Work

While quite a number of studies on API design exist, only a few focus on remote APIs. Nonetheless, a number of local API topics are likely transferable to a certain extent to the remote API context. APIs design is not only about technical aspects but also about the development culture [10]. There are a number of empirical study on API documentation [11]–[13] which emphasize roles such as API documenter and API designer in this context. The scientific literature has studied various API quality attributes, such as accessibility [14], stability [15], compatibility [16], [17], evolvability [18]–[20], and usability [21]–[23].

Robillard [24] raised the question what makes APIs hard to learn from the developer's perspective. In a survey with 80 experienced developers he shows that the most popular learning strategy is reading documentation. Murphy et al. [25] studied the usability aspect from API designers' perspective. They interviewed 24 professional designers from the industry. They found, among other things, that many designers learned API design on the job, that it is hard for them to discern which potential use cases of the API users will value most, and that they lack tools to gather aggregated feedback from the deployed API.

In comparison to those other works, our work is the first to study the relation of DDD and API design. Our work structures the resulting design space systematically, in terms of design problems via ADDs, their decision options, their relations, and the API decision drivers (forces) linked to them. Finally, many of the related works focus on local programming APIs. As mentioned, remote APIs are studied only in a few works yet, and it is not clear if and which properties of local APIs can be transfered to remote APIs.

Design patterns are an essential concept offering decision options in our ADDs, and they offer a systematic way to organize API design knowledge. The Microservice API Patterns [2], [26][1] collect a number of API patterns in the realm of remote APIs. Context Mapper and its MDSL generator implement parts of the design advice and options that our literature review in this paper reports [27], especially those options related to the Microservice API Patterns and API contracts. Richardson [28] describes some patterns with relevance to API design such as API Gateway or Command Query Responsibility Segregation (CQRS). Gosrki and Wojtach [29] propose the Use Case API pattern, an architectural pattern for use case based interfaces exposed e.g. via RESTful services. All three works apply DDD and explain relations to DDD patterns. However, none of these works has yet empirically investigated the relation of microservice API design and DDD.

III. Research Method

In this paper we conducted a grounded theory study based on the grey literature [30]. We used formal modeling to precisely encode our findings (similar to [26]). *Grounded Theory (GT)* [6], [7] is a systematic research method for

[1]See also: https://microservice-api-patterns.org/

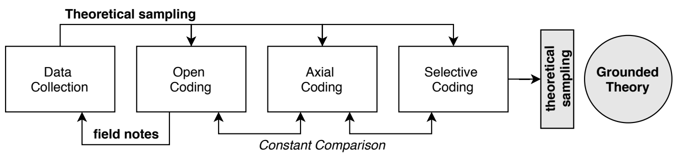


Fig. 1. Research Method Overview

discovery of theory from data. The *grey literature* [8], [30] is the main data source in our work. In software engineering, grey literature can be defined as "any material about software engineering that is not formally peer-reviewed nor formally published" [30]. We decided to study grey literature sources representing acknowledged practitioners' views on *the interrelation of distributed APIs and DDD*. These sources are then used as unbiased descriptions of established practices in the further analysis.

We studied each knowledge source in depth, followed GT's coding process, as well as a constant comparison procedure to derive a model, as illustrated in Figure 1. In contrast to classic GT, the research begins with an initial research question, as in Charmaz's constructivist GT [31]. Whereas GT typically uses textual analysis, uses textual codes only initially and then transfers them into formal software models (hence it is model-based). The data sources studied in this paper (see Table I) include *Discussion Forum Post*, *Practitioner Audience Article*, and Practitioner *Slides*. We searched for API and DDD related keywords in search engine like Bing or Google to find the initial data sources. We only selected practitioner articles from practitioners targeted at other practitioners. For this, the authors judged whether the sources focused on topics such as designing APIs for systems based on DDD, how to structure API and DDD-related parts of the system, or which DDD concepts help to structure and design an API and how. We reviewed each source in the author team. We excluded sources that seemed to sell a product or a service. As GT is mainly concerned with phenomena that have specifically been observed to exist [32], it is only necessary to find enough source that are relevant with regard to the phenomena being study. It is not necessary to find all possibly relevant sources (as it would be for instance in a systematic literature study).

The knowledge source acquisition is applied in many iterations. We excluded sources that seemed to sell a product or a service. This is an essential property of GT called *theoretical sampling* [6], which means that results of each data analysis step are used for the following data collection. The researchers should actively find new data sources driven by the results of the data analysis. In GT, the coding and data selection cycle stops, when *theoretical saturation* [6] is reached. We stopped our analysis when five to seven additional knowledge sources did not add anything new to our understanding of the research topic. As a result of this very conservative operationalization of theoretical saturation, we studied a rather large number of sources in depth (32 in total, summarized in Table I).

The authors analyzed every source line by line to elicit

TABLE I
LIST OF KNOWLEDGE SOURCES INCLUDED IN THE PAPER

ID	Title	Tiny URL	Source Type	Example	Source Code
s1	*Bounded Context in APIs (1/2)*	tinyurl.com/api-ddd-s1	Practitioner Audience Article	No	No
s2	*DDD & REST Domain Driven Apis for the Web*	tinyurl.com/api-ddd-s2	Slides	Yes	Yes
s3	*Why the domain model should not be used as resources in REST API?*	tinyurl.com/api-ddd-s3	Discussion Forum Post	Yes	Yes
s4	*How to clearly define boundaries of a bounded context*	tinyurl.com/api-ddd-s4	Discussion Forum Post	No	Yes
s5	*REST API Design - Resource Modeling*	tinyurl.com/api-ddd-s5	Practitioner Audience Article	Yes	No
s6	*Rest API and DDD*	tinyurl.com/api-ddd-s6	Discussion Forum Post	Yes	Yes
s7	*Introduction to DDD Lite: When microservices in Go are not enough*	tinyurl.com/api-ddd-s7	Practitioner Audience Article	Yes	Yes
s8	*REST and DDD: incompatible?*	tinyurl.com/api-ddd-s8	Practitioner Audience Article	Yes	No
s9	*Domain Driven Design - External Data API as Respository or Service*	tinyurl.com/api-ddd-s9	Discussion Forum Post	Yes	No
s10	*Conceptual mismatch between DDD Application Services and REST API*	tinyurl.com/api-ddd-s10	Discussion Forum Post	Yes	Yes
s11	*Microservices: Overview, Misinterpretations and Misuses*	tinyurl.com/api-ddd-s11	Practitioner Audience Article	No	No
s12	*Design a DDD-oriented microservice*	tinyurl.com/api-ddd-s12	Practitioner Audience Article	Yes	No
s13	*Apply Domain-Driven Design to microservices architecture*	tinyurl.com/api-ddd-s13	Practitioner Audience Article	No	No
s14	*Designing APIs and Microservices Using Domain-Driven Design*	tinyurl.com/api-ddd-s14	Slides	Yes	No
s15	*REST Service and CQRS*	tinyurl.com/api-ddd-s15	Discussion Forum Post	Yes	Yes
s16	*Exposing CQRS Through a RESTful API*	tinyurl.com/api-ddd-s16	Practitioner Audience Article	Yes	Yes
s17	*REST-first design is Imperative, DDD is Declarative [Comparison] - DDD w/ TypeScript*	tinyurl.com/api-ddd-s17	Practitioner Audience Article	Yes	Yes
s18	*Designing APIs for microservices*	tinyurl.com/api-ddd-s18	Practitioner Audience Article	Yes	Yes
s19	*Moving Towards Domain Driven Design in Go*	tinyurl.com/api-ddd-s19	Practitioner Audience Article	Yes	Yes
s20	*Microservices, Apache Kafka, and Domain-Driven Design*	tinyurl.com/api-ddd-s20	Practitioner Audience Article	Yes	No
s21	*API & Domain Driven Design*	tinyurl.com/api-ddd-s21	Slides	Yes	Yes
s22	*Implementing Domain-Driven Design for Microservice Architecture*	tinyurl.com/api-ddd-s22	Practitioner Audience Article	Yes	No
s23	*Pattern: Decompose by subdomain Context*	tinyurl.com/api-ddd-s23	Practitioner Audience Article	Yes	No
s24	*Building Microservices with Event Sourcing/CQRS in Go using gRPC, NATS Streaming and CockroachDB*	tinyurl.com/api-ddd-s24	Practitioner Audience Article	Yes	Yes
s25	*Aggregate Oriented Microservices*	tinyurl.com/api-ddd-s25	Practitioner Audience Article	Yes	Yes
s26	*Designing a Serverless Application with Domain Driven Design*	tinyurl.com/api-ddd-s26	Slides	Yes	No
s27	*Bounded Contexts With Axon*	tinyurl.com/api-ddd-s27	Practitioner Audience Article	Yes	No
s28	*Building Real-Time Web Applications using wolkenkit*	tinyurl.com/api-ddd-s28	Practitioner Audience Article	Yes	Yes
s29	*Uncovering API Implementation*	tinyurl.com/api-ddd-s29	Practitioner Audience Article	Yes	No
s30	*Implementing an API-First Design Methodology*	tinyurl.com/api-ddd-s30	Practitioner Audience Article	No	No
s31	*API First Development*	tinyurl.com/api-ddd-s31	Practitioner Audience Article	Yes	Yes
s32	*The API Design Process*	tinyurl.com/api-ddd-s32	Practitioner Audience Article	No	No

the required information, and then created one field note per source. This is a memo writing technique where we noted down the conceptual details, hidden interpretations, and other interesting details on the sources. We established traceability from lines in the sources, over each coding step, to the formal model of our theory derived during the GT study. Next, we followed Corbin's and Strauss' method for GT coding [7]: First, we applied *Open Coding* with the goal to transform conceptual details into conceptual labeling. Next, during *Axial Coding* we identified categories in the concepts, e.g., by identifying concepts that reappear in the data, synonymous concepts, related concepts, and so on. The identification helps to find out the relations between the concepts. Finally, during *Selective Coding* we carved out main ideas of the theory, i.e., understanding the big picture by reflecting on the data and analysis results. As mentioned, after initial text-based open coding, we used formal UML-based modelling for axial and selective coding, instead of the often-used text-based coding process, in order to develop a precisely defined and consistent theory. We used the Python tool CodeableModels[2] for this. Formal modelling also eased establishing an audit trail of the research, and thus enable repeatability of the study. In addition, we provide public access to the original data as well as the derived models[3]

[2]https://github.com/uzdun/CodeableModels

[3]We provide all open and axial coding files, derived formal models in Python, and generated models (in UML, Markdown, and Latex) as a replication package for download on https://doi.org/10.5281/zenodo.4569578.

IV. ARCHITECTURAL DESIGN DECISIONS

In this section, we present the results of our study in form of ADDs that appear in the discussions of the practitioners in our grey literature sources. Figure 2 shows an overview of the decisions identified and their relations (explained in detail below). For space reasons, we cannot give detailed examples. Please note that Table I[4] shows which sources provide examples (with or without source code).

A. Domain Model Mapping Decision

The core result of a DDD modeling effort is the *Domain Model*, which defines the *Ubiquitous Language*. This term is used by Evans to describe the language shared by the whole team, including developers, domain experts, and other participants [3]. For larger domains, it is usually not possible to model the whole *Ubiquitous Language* in a single unified model. Instead DDD divides larger domains into different *Bounded Contexts* and explicitly models their relationships, usually with the help of *Context Maps* [3], [5].

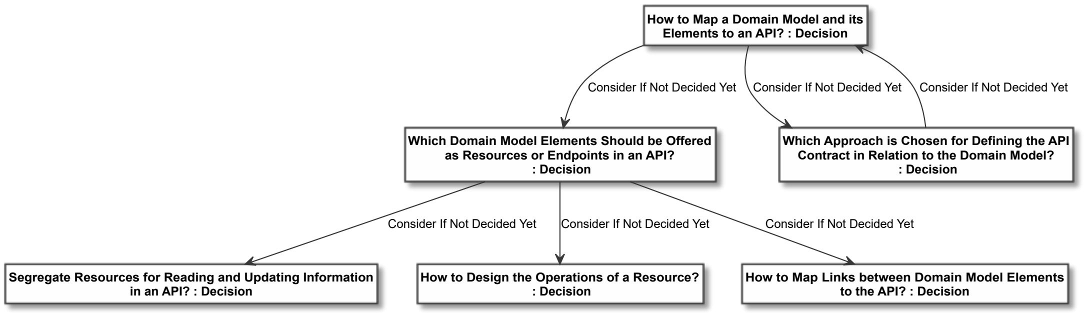
In the sources analyzed in our study, the practitioners frequently discuss the ADD *How to Map the Domain Model and its Elements to an API?* (overall 16 source discuss this ADD) as summarized in Table II. We found evidence for seven possible design options as solutions for this ADD as illustrated in Figure 3. A simple solution is to *Expose the*

[4]In the table we use the following color coding to visualize the frequency of the evidences: □ < 5%, □ < 10%, ░ < 20%, ░ < 35%, ▒ < 50%, ▓ < 70%, ■ ≥ 70%.

TABLE II
STUDY RESULTS: OVERVIEW OF DESIGN DECISIONS, DECISION OPTIONS, EVIDENCES AND RELATED FORCES

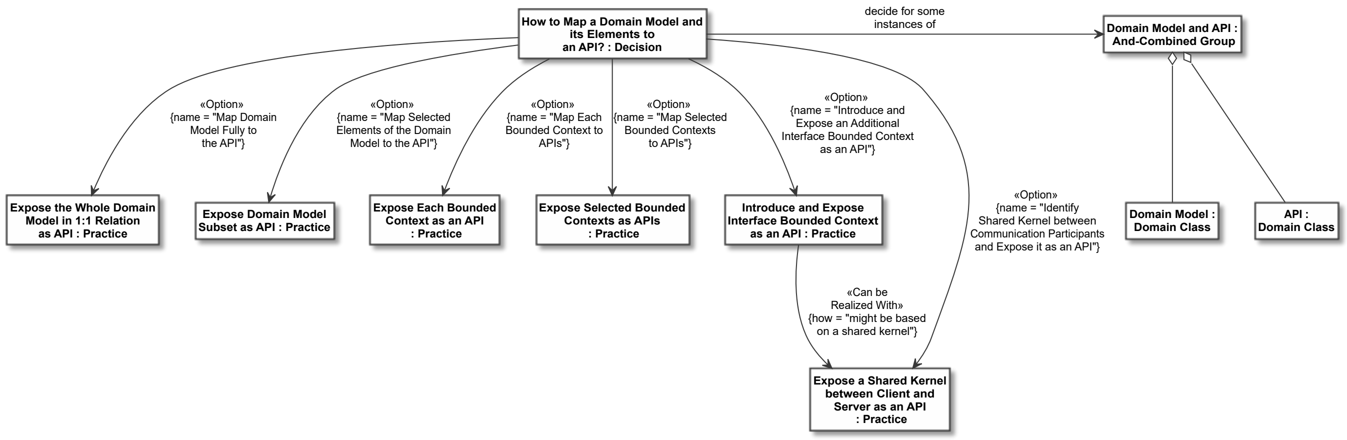
Design Decision	#	Solution	Evidences	Forces
How to Map a Domain Model and its Elements to an API?	16	1.Expose the Whole Domain Model in 1:1 Relation as API	s1, s3, s9	f1(--), f2(-), f17(+), f7(--), f8(-), f4(-), f10(-), f15(+)
		2.Expose Domain Model Subset as API	s3, s6, s9	f1(o), f2(o), f17(+), f7(o), f8(o), f4(-), f10(-), f15(+)
		3.Expose Each Bounded Context as an API	s1, s4, s13, s14, s20, s27, s29, s32	f1(-), f2(-), f17(-), f7(-), f8(-), f4(-), f10(-), f15(+)
		4.Expose Selected Bounded Contexts as APIs	s1, s3, s4, s13, s14, s20, s21, s22, s23, s27, s29, s32	f1(o), f2(o), f17(-), f7(o), f8(o), f4(-), f10(o), f15(+)
		5.Introduce and Expose Interface Bounded Context as an API	s1, s3, s23	f1(+), f2(+), f17(++), f7(+), f8(+), f4(+), f10(+), f15(-)
		6.Expose a Shared Kernel between Client and Server as an API	s1, s3	f1(+), f2(+), f17(++), f7(+), f8(+), f4(+), f10(+), f15(-)
Which Approach is Chosen for Defining the API Contract in Relation to the Domain Model?	15	1.Explicitly Specify the API Contract	s3, s8, s10, s13, s17, s18, s21, s30, s31, s32	f21(+), f24(+), f25(+), f26(o), f10(o), f18(o), f23(o), f27(++)
		2.Extract API Contract from Domain Model	s3, s10, s13, s17, s18	f21(+), f24(+), f25(+), f26(o), f10(o), f18(o), f23(o), f27(+)
		3.Domain Model Defines API Contract	s3, s10, s13, s18	f21(-), f24(--), f25(--), f26(o), f10(o), f18(o), f23(o), f27(o)
		4.Bounded Context Defines API Contract	s3, s13, s27	f21(-), f24(--), f25(--), f26(o), f10(o), f18(o), f23(o), f27(o)
		5.Write API Code First which Defines the Contract	s17, s30	f21(--), f24(--), f25(--), f26(++), f10(-), f18(-), f23(--), f27(--)
Which Domain Model Elements Should be Offered as Resources or Endpoints in an API?	21	1.Entities as API Resources	s1, s2, s5, s6, s12, s13, s14, s16, s17, s18, s19, s21, s32	f2(--), f11(--), f3(-), f5(-), f6(-), f7(-), f18(-), f19(-), f13(-)
		2.Domain Services as API Resources	s9, s13, s19	f2(o), f11(+), f3(o), f5(++), f6(++), f7(+), f18(+), f19(+), f13(+)
		3.Aggregate Roots as API Resources	s1, s2, s5, s6, s10, s12, s13, s17, s18, s19, s25, s26, s27, s28	f2(+), f11(++), f3(+), f5(+), f6(+), f7(+), f18(+), f19(+), f13(+)
		4.Bounded Contexts as API Resources	s1, s2, s5, s20, s25, s26, s29	f2(+), f11(o), f3(+), f5(+), f6(+), f7(-), f18(o), f19(o), f13(o)
		5.Domain or Business Processes as API Resources	s5, s10, s20	f2(+), f11(++), f3(+), f5(+), f6(+), f7(+), f18(+), f19(+), f13(+)
Segregate Resources for Reading and Updating Information in an API?	11	1.Expose Segregated Command and Query Resources in API	s5, s6, s8, s11, s12, s15, s16, s24, s25, s27, s28	f7(-), f11(-), f6(+), f20(+)
		2.Do Not Segregate Queries and Commands in an API	s5, s6, s8, s11, s12, s15, s16, s24, s25, s27, s28	f7(+), f11(+), f6(-), f20(-)
How to Design the Operations of a Resource?	26	1.CRUD-Style Operations on Resources	s1, s2, s3, s5, s6, s7, s8, s10, s11, s12, s14, s15, s16, s17, s18, s19, s20, s21, s29	f2(-), f3(--), f5(--), f6(--), f16(-), f18(-), f11(-), f19(-), f13(-), f22(+)
		2.Domain Operations on Resources	s1, s2, s3, s4, s5, s6, s7, s9, s10, s11, s12, s15, s16, s18, s20, s25	f2(+), f3(+), f5(+), f6(+), f16(+), f18(+), f11(+), f19(+), f13(+), f22(+)
		3.Encode Operations as Commands in the Payload	s6, s15, s16	f2(+), f3(+), f5(+), f6(+), f16(+), f18(o), f11(+), f19(+), f13(+), f22(-)
		4.Expose Domain Events as State Transitions	s2, s3, s4, s5, s9, s11, s12, s14, s16, s20, s24, s25, s26, s27, s28, s29	f2(++), f3(+), f5(+), f6(++), f16(+), f18(+), f11(+), f19(+), f13(+), f22(-)
		5.Expose Domain Events via Feeds or Pub/Sub	s2, s20, s24, s27, s28	f2(++), f3(+), f5(+), f6(++), f16(+), f18(+), f11(+), f19(+), f13(+), f22(-)
How to Map Links between Domain Model Elements to the API?	8	1.None	s1, s2, s31	
		2.Use Distributed or Hypermedia Links in the Payload	s1, s2, s3, s8, s10, s18, s31	f11(+), f9(+), f10(+), f12(++), f2(++), f13(++), f14(-), f4(-), f5(-), f6(-)
		3.Pass Object Identifiers in the Payload	s1, s2, s18, s31	f11(+), f9(+), f10(+), f12(++), f2(-), f13(-), f14(+), f4(-), f5(-), f6(-)
		4.Embed Linked Data in the Payload	s1, s2	f11(-), f9(-), f10(-), f12(-), f2(o), f13(+), f14(+), f4(+), f5(+), f6(+)

Forces Codes/Sources: **f1**:Brittle Interfaces [s1, s23], **f2**:Avoid Exposing Domain Model Details in API [s1, s2, s3, s4, s5, s7, s10, s17, s18, s29, s32], **f3**:Chatty API [s1, s5, s11, s18], **f4**:Minimize API calls [s1, s2], **f5**:Performance [s1, s11, s16, s18], **f6**:Scalability [s1, s4, s5, s8, s11, s16, s18, s20, s24, s28, s29], **f7**:API Complexity [s1, s5, s6, s11, s12, s15, s18, s25, s29], **f8**:API Usability [s1, s5, s14], **f9**:API Evolvability [s1, s5, s18, s31], **f10**:API Modifiability [s1, s3, s4, s5, s17, s18, s23, s30, s31], **f11**:Data Consistency [s1, s2, s5, s10, s12, s16, s18, s20, s21, s22, s25, s27, s28, s29], **f12**:Message Size [s1], **f13**:Coupling of Clients to Server [s2, s5, s6, s8, s10, s14, s18, s20, s22, s23, s27], **f14**:Protocol Complexity in Client [s2, s8, s10], **f15**:Design and Implementation Effort [s3, s21, s23, s29, s30], **f16**:Interface Design Limits Domain Model Design [s3, s10], **f17**:Clients Need to Manage Crossing Model Boundaries [s3, s4], **f18**:Maintainability of API and API Consumers [s5, s6, s7, s10, s17, s29, s30], **f19**:Reliability [s4, s5], **f20**:Eventual Consistency Support [s5, s15, s16, s24, s25, s27, s28], **f21**:Separation of API Contract and Domain Concerns [s1, s3, s5, s10, s13, s24, s29, s30, s32], **f22**:API Understandability [s6, s7, s15, s16, s17, s30, s31, s32], **f23**:Can Lead to Anemic Domain Model Anti-Pattern [s8, s17], **f24**:API Stability [s10, s23, s30, s31], **f25**:Domain Model Flexibility [s10], **f26**:Initial Effort Required [s17, s30, s31], **f27**:Support for External or Public Clients [s18]



How to Map a Domain Model and its Elements to an API? : Decision
Consider If Not Decided Yet
Consider If Not Decided Yet
Consider If Not Decided Yet
Which Domain Model Elements Should be Offered as Resources or Endpoints in an API? : Decision
Which Approach is Chosen for Defining the API Contract in Relation to the Domain Model? : Decision
Consider If Not Decided Yet
Consider If Not Decided Yet
Consider If Not Decided Yet
Segregate Resources for Reading and Updating Information in an API? : Decision
How to Design the Operations of a Resource? : Decision
How to Map Links between Domain Model Elements to the API? : Decision

Fig. 2. ADDs and ADD Relations: Overview



How to Map a Domain Model and its Elements to an API? : Decision
decide for some instances of
Domain Model and API : And-Combined Group
«Option» {name = "Map Domain Model Fully to the API"}
«Option» {name = "Map Selected Elements of the Domain Model to the API"}
«Option» {name = "Map Each Bounded Context to APIs"}
«Option» {name = "Map Selected Bounded Contexts to APIs"}
«Option» {name = "Introduce and Expose an Additional Interface Bounded Context as an API"}
«Option» {name = "Identify Shared Kernel between Communication Participants and Expose it as an API"}
Expose the Whole Domain Model in 1:1 Relation as API : Practice
Expose Domain Model Subset as API : Practice
Expose Each Bounded Context as an API : Practice
Expose Selected Bounded Contexts as APIs : Practice
Introduce and Expose Interface Bounded Context as an API : Practice
Domain Model : Domain Class
API : Domain Class
«Can be Realized With» {how = "might be based on a shared kernel"}
Expose a Shared Kernel between Client and Server as an API : Practice

Fig. 3. Domain Model Mapping Decision

Whole Domain Model in 1:1 Relation as API but this is seen as working only for small examples, as it leads to negative impacts on coupling and maintainability forces such as *Brittle Interfaces*, *API Complexity*, and *Avoiding Exposing Domain Model Details in API*. A usually better working solution is *Expose Domain Model Subset as API* which partly improves on the negative force impacts. Both solutions have benefits like little required *Design and Implementation Effort*. For complex domains, it can be advisable to consider the *Bounded Contexts* as well. Then there are the options to *Expose Each Bounded Context as an API* or *Expose Selected Bounded Contexts as APIs*, both leading to a sub-division of the API along the *Bounded Context* boundaries. In most large domains, the latter solution is seen as being better suited to avoid *Brittle Interfaces*, *Exposing of Domain Model Details in the API*, and *API Complexity*, and improve *API Usability* and *API Modifiability*. Both solutions offer positive impact on *Design and Implementation Effort*, but require more effort than the first two solutions. The downside of those solutions is that *Clients Need to Manage Crossing Model Boundaries*, i.e., the boundaries between the *Bounded Contexts*. One suggested solution to this problem is to *Introduce and Expose Interface Bounded Context as an API*. That is, a new special *Bounded Context* that represent the API interface is exposed. This solution is neutral or positive on all so far mentioned forces, except the *Design and Implementation Effort* where it leads to additional effort compared to all other so far mentioned solutions.

In some cases, it might make sense to consider *Expose a Shared Kernel between Client and Server as an API*, which can be seen as an option how to realize the solution *Introduce and Expose Interface Bounded Context as an API*. *Shared Kernel* is a DDD relation between two *Bounded Contexts* in which some subset of the domain model is shared between the two teams developing the contexts. For example, it is often implemented as a local code library available to API client and server. Here, the client and server contexts would share such a *Shared Kernel*. This is a good solution with similar force impacts as *Introduce and Expose Interface Bounded Context as an API* in cases, where close interaction between the teams developing client and server is acceptable.

Please note that the two options explained before *Expose Each Bounded Context as an API* or *Expose Selected Bounded Contexts as APIs* also use *Bounded Context* relations to determine what is exposed as a remote API: They would use *Bounded Context* relations that are more frequently leading to remote interconnections between services in the implementation such as *Open Host Service*, *Customer/Supplier* or *Anti-Corruption Layer*. Implementation-wise such solutions lead to use of the *Service Layer* pattern [33] for the services exposing the API, with each service being the *Remote Facade* [33] for the elements shielded by the API.

As can be seen in Figure 2, this decision has a number of direct and indirect follow-on decisions that should be considered as well. It is also suggested to first consider the API contract related ADD in Section IV-B and then consider the domain model mapping ADD. Many of the DDD elements in both decisions are sometimes using *Event Storming*; while not directly related to API design, such techniques have an indirect influence on API design which is interesting to explore in future work.

B. API as Contract Decision

The *API Contract* is an API-related concept in which the API is defined in some formal language, e.g., based on Open API or RAML for RESTful APIs, WSDL for SOAP APIs, some special Domain-specific Language for *API Contract* definition, and so on. *API Contracts* can support the decoupling between API consumers and API providers. The contracts are essential in terms of mutual understanding between those parties. Ideally, the *API Contract* is more stable than the backend systems, i.e., it is only changed rarely, whereas the backend systems often constantly evolve.

The relation between DDD and *API Contract* leads to the ADD *Which Approach is Chosen for Defining the API Contract in Relation to the Domain Model?* This ADD is mentioned in 15 sources (see Table II). We have identified 5 decision options in the sources. The first option *Explicitly Specify the API Contract* describes the practice to design the *API Contract* before or relatively independent from the domain model. A specific variant of this, sometimes mentioned, is *Specify the API Contract First*; usually it meant as continuous improvement of a contract specification, starting from a specification, not a rigid, pre-defined contract. *Explicitly Specify the API Contract* can for instance be achieved using the *Expose a Shared Kernel between Client and Server as an API* or *Introduce and Expose Interface Bounded Context as an API* options from the previous decision. Another option is *Extract API Contract from Domain Model*, i.e., the practice of selecting elements of the domain model to be exposed in the contract. Both are practices that help in reaching *Separation of API Contract and Domain Concerns*, are positive for *API Stability*, and enable *Domain Model Flexibility*.

Not all possible solutions focus on *Separation of API Contract and Domain Concerns*: *Domain Model Defines API Contract* and *Bounded Context Defines API Contract* use the elements of the domain model or a *Bounded Context* as the elements of the *API Contract*. This leads to a worse *Separation of API Contract and Domain Concerns* as the API elements are tightly coupled to the faster evolving domain concept, which in turn is negative for either *API Stability* or *Domain Model Flexibility*. *Domain Model Defines API Contract* can be realized with the *Expose the whole Domain Model in 1:1 Relation as API* or *Expose Domain Model Subset as API* options of the previous decision. *Bounded Context Defines API Contract* can be realized with the *Expose Each Bounded Context as an API* or *Expose Selected Bounded Contexts as API* options of the previous decision.

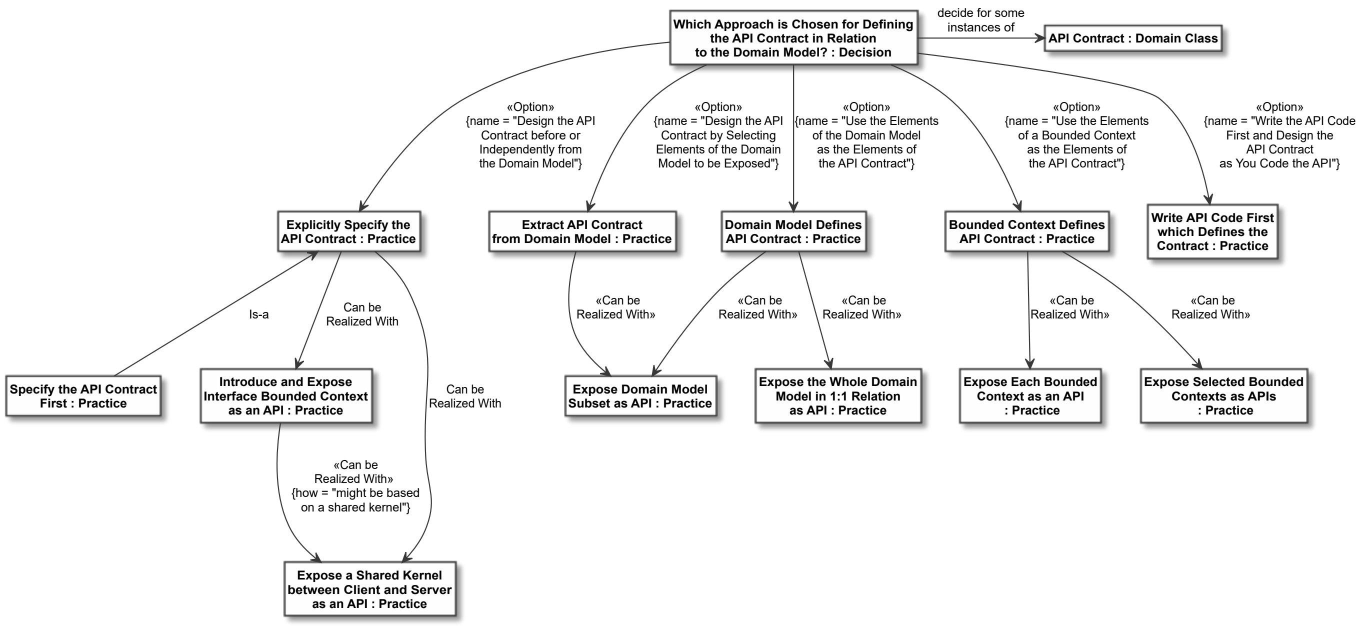
The option *Write API Code First Which Defines the Contract* describes an API first approach which is seen, in contrast to the prior options, rather negatively: It can lead to bad *Separation of API Contract and Domain Concerns* as the domain model is based on premature, often low-level API design decisions, which *Can Lead to Anemic Domain Model Anti-pattern*, i.e., a domain model which contains little deep domain knowledge. This results in bad *API Stability* or *Domain Model Flexibility*, as well as bad *API Modifiability* and other issues in *Maintainability of API and API Consumers*.

Finally, there is the force *Support for External or Public Clients* to be considered. It is important to distinguish between two types of APIs: public APIs that client applications call, and backend APIs that are used for communication between services. For external clients or public clients the maintainability, stability, and modifiability forces are usually of much higher importance than for internal APIs where a certain level of control of changes is possible. Thus for *Support for External or Public clients*, the option *Write API Code First which Defines the Contract* performs worst. *Domain Model Defines API Contract* and *Bounded Context Defines API Contract* can be less positive here than *Explicitly Specify the API Contract* and *Extract API Contract from Domain Model*, as they lead to a less strict separation of API and domain model, which might impede stability. The API as Contract decision has a strong connection to the domain model mapping decision and vice versa.

C. Designing API Resources Decision

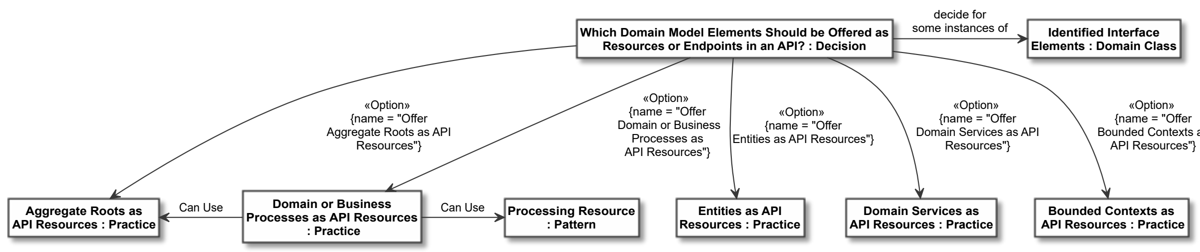
We use the term *API Resource* for a set of related interface elements exposed in an API. Sometimes the term *API Endpoint* is used instead in a similar sense, even though endpoint usually denotes a remote location of a resource such as a URI. Please note that we use *API Resource* for any API technology, not limited to the RESTful resources. On the other hand, the decision outcomes of the two previous decisions determine the scope in which *Interface Elements* can be identified in the *Domain Model* and/or *API contract*. These need to be mapped to *API Resources*, which is the purpose of this ADD.

The decision on *Support for External or Public clients Which Domain Model Elements Should be Offered as Resources or Endpoints in an API?* (21 evidences as summarized in Table II) consist of five options namely *Entities*, *Aggregate Roots*, *Domain Services*, *Bounded Contexts*, and *Domain or Business Process as API Resources* (typically realized as a *Processing Resource* [2]). That is, the basic abstractions *Entity*, *Aggregate*, and *Service* in tactical DDD [3] are discussed as possible sources for *API Resources*, as well as the two "broader" concepts *Bounded Contexts* and *Processes* (the later is applicable only in process-based system designs). Interestingly *Bounded Contexts* have already been used as API scopes in some options of the previous two ADDs, and are rather a concept of strategic design in DDD: *Strategic Design* shapes the big picture, while *Tactical Design* dives into design



Which Approach is Chosen for Defining the API Contract in Relation to the Domain Model? : Decision
decide for some instances of
API Contract : Domain Class
«Option» {name = "Design the API Contract before or Independently from the Domain Model"}
«Option» {name = "Design the API Contract by Selecting Elements of the Domain Model to be Exposed"}
«Option» {name = "Use the Elements of the Domain Model as the Elements of the API Contract"}
«Option» {name = "Use the Elements of a Bounded Context as the Elements of the API Contract"}
«Option» {name = "Write the API Code First and Design the API Contract as You Code the API"}
Explicitly Specify the API Contract : Practice
Extract API Contract from Domain Model : Practice
Domain Model Defines API Contract : Practice
Bounded Context Defines API Contract : Practice
Write API Code First which Defines the Contract : Practice
Is-a
Can be Realized With
Can be Realized With
«Can be Realized With»
«Can be Realized With»
«Can be Realized With»
«Can be Realized With»
«Can be Realized With»
Specify the API Contract First : Practice
Introduce and Expose Interface Bounded Context as an API : Practice
Expose Domain Model Subset as API : Practice
Expose the Whole Domain Model in 1:1 Relation as API : Practice
Expose Each Bounded Context as an API : Practice
Expose Selected Bounded Contexts as APIs : Practice
«Can be Realized With» {how = "might be based on a shared kernel"}
Expose a Shared Kernel between Client and Server as an API : Practice

Fig. 4. API as Contract Decision



Which Domain Model Elements Should be Offered as Resources or Endpoints in an API? : Decision
decide for some instances of
Identified Interface Elements : Domain Class
«Option» {name = "Offer Aggregate Roots as API Resources"}
«Option» {name = "Offer Domain or Business Processes as API Resources"}
«Option» {name = "Offer Entities as API Resources"}
«Option» {name = "Offer Domain Services as API Resources"}
«Option» {name = "Offer Bounded Contexts as API Resources"}
Aggregate Roots as API Resources : Practice
Can Use
Domain or Business Processes as API Resources : Practice
Can Use
Processing Resource : Pattern
Entities as API Resources : Practice
Domain Services as API Resources : Practice
Bounded Contexts as API Resources : Practice

Fig. 5. Designing API Resources Decision

details [5]. This shows that this ADD is located at the border of these two central DDD design levels.

Entities as API Resources has the highest number of evidences (13 sources) as shown in Table II. While being an obvious option, a number of practitioners in our sources advise strongly against using *Entities* as foundations for *API Resource*. With 10 sources, *Aggregate Roots as API Resources* has the second highest number of evidences, and seems to be the most often recommended option how to start looking for *API Resources*. As an *Aggregate* abstracts the implementation details of a number of related *Entities* and other DDD model elements, it naturally serves as an *Identified Interface Element*. Practitioners agree that *Aggregate Roots* are a good starting point for *API Resources*, but many other options exist and often deliberate, incremental design is needed to find good *API Resources*. Some practitioners suggest *Domain Services as API Resources*. For instance, they can lead to stateless *API Resources* in addition to the usually stateful resources based on *Aggregates*. Both *Bounded Contexts* and *Processes* bundle a number of related DDD model elements and can thus, both according to a few practitioners, be candidates for defining *API Resources*. Some sources suggest to consider first *Aggregates*, and then the other options.

Main drawbacks of exposing *Entities* are issues related to *Avoiding Exposing Domain Model Details in API*, *Data Consistency*, and *Chatty APIs*. This can lead to bad *Performance*, *Scalability* issues, and high *API Complexity*. Other issues are possibly *Coupling of Clients and Server* and other issues in *Maintainability of API and API Consumers*. *Aggregate Roots as API Resources* and where applicable *Domain Services as API Resources* can help to avoid most of these issues. *Domain or Business Processes as API Resources* can have a similar positive impact on the forces, if a process-based abstraction makes sense in the domain context. Certain *Bounded Contexts* can work well for many of the forces, but there is a risk of higher *API Complexity* due the size of the *Bounded Contexts*. Also *Data Consistency* can be more natural to manage on an *Aggregate* than on a *Bounded Context* and low *Coupling of Clients and Server* can be harder to reach.

D. Resource Segregation Decision

After *API Resources* have been identified, e.g. using the options from the previous decision, there sometimes is the option to decide whether or not to *Segregate Resources for Reading and Updating Information in an API*. This is closely related to the *Command Query Responsibility Segregation (CQRS)* Pattern [28]. The idea of CQRS is to use a different model to update data than the model that is used to read data. A common implementation technique in the context of event-based microservices is *Event Sourcing* [28]. As querying the corresponding event store can be hard to realize efficiently, often CQRS is used in this context.

If CQRS is used e.g. for a microservice design in the back-end, often it makes sense to offer the segregated commands and queries as two resources in an API which this ADD is about (see Figure 6). Practitioners agree that CQRS is a complex pattern and it should only be chosen if it offers a substantial benefit. It can be chosen in the backend and not exposed in the API, too. It is possible to use *Encoding Operations as Command in Payload* in the operations design decision in Section IV-F as a practice to realize the segregation option. The segregation option would enable *Eventual Consistency* as a practice, not only in the backend, but covering the clients.

Regarding forces, this option has the benefit of possibly improving *Scalability* and enabling *Eventual Consistency Support* where it is needed, e.g. in long running transactions. Downsides are higher *API Complexity* and less *Data Consistency*.

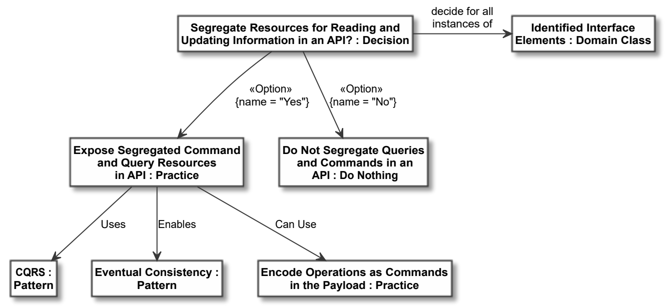

Segregate Resources for Reading and Updating Information in an API? : Decision
decide for all instances of
Identified Interface Elements : Domain Class
«Option» {name = "Yes"}
«Option» {name = "No"}
Expose Segregated Command and Query Resources in API : Practice
Do Not Segregate Queries and Commands in an API : Do Nothing
Uses
Enables
Can Use
CQRS : Pattern
Eventual Consistency : Pattern
Encode Operations as Commands in the Payload : Practice

Fig. 6. Resource Segregation Decision

E. Link Mapping Decision

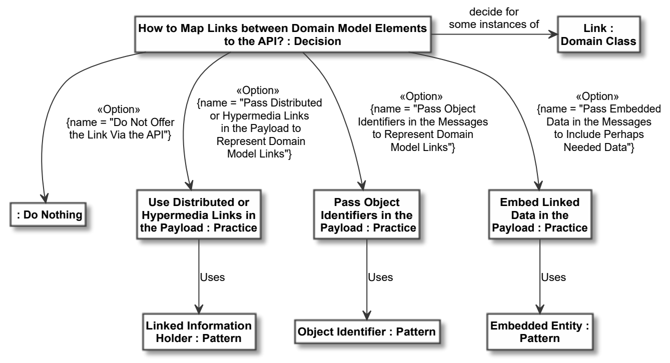

How to Map Links between Domain Model Elements to the API? : Decision
decide for some instances of
Link : Domain Class
«Option» {name = "Do Not Offer the Link Via the API"}
«Option» {name = "Pass Distributed or Hypermedia Links in the Payload to Represent Domain Model Links"}
«Option» {name = "Pass Object Identifiers in the Messages to Represent Domain Model Links"}
«Option» {name = "Pass Embedded Data in the Messages to Include Perhaps Needed Data"}
: Do Nothing
Use Distributed or Hypermedia Links in the Payload : Practice
Pass Object Identifiers in the Payload : Practice
Embed Linked Data in the Payload : Practice
Uses
Uses
Uses
Linked Information Holder : Pattern
Object Identifier : Pattern
Embedded Entity : Pattern

Fig. 7. Link Mapping Decision

Another decision following the API resource decision is on link mapping. In APIs the links between *API Resources* play a central role. In DDD the links between model elements have a similar role. Obviously not all links in the DDD model are candidates to be exposed in an API, but only those between the model elements offered as *API Resources*, e.g. following the ADD from Section IV-C.

Figure 7 shows the link mapping decision. Many links are not mapped, as explained. The *Use Distributed or Hypermedia Links in the Payload* option means to use standard distributed/hypermedia links, such as URIs in RESTful HTTP or URIs and HAL/JSON-LDs in JSON. This conforms to using the *Linked Information Holder* pattern [2], which describes a linked API Resource. An alternative is *Embed Linked Data in the Payload* which follows the *Embedded Entity* pattern [2] where the content (or a part of it) is added to the message payload instead of linking to it. Finally, there is the option to *Pass Object Identifiers in the Payload*, i.e. to follow the *Object Identifier* pattern [34]. This means to send an *Object Identifier* that is meaningful (only) in the server context, but contains no remote location information such as a distributed link.

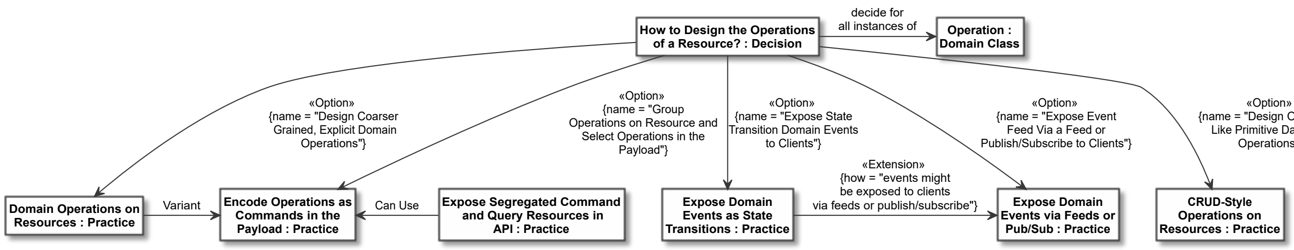
Compared to the embedding option, use of distributed links is beneficial for *Data Consistency* as the link is always up-to-date, and thus *API Evolvability* and *API Modifiability* are positively influenced. It leads also to smaller *Message Sizes*. Links however lead to higher *Protocol Complexity*, make it harder to *Minimize API Calls*, and can have a worse *Performance* and *Scalability* because of many resulting distributed calls. The *Object Identifier* based option is very similar in its effects to the distributed links based option. In direct comparison, it has the disadvantages of possibly *Exposing Domain Model Details in API* as well as higher *Coupling of Clients to Server*.

F. Operation Design Decision

The ADD *How to Design the Operations of a Resource?* shown in Figure 8 appeared in 26 investigated sources (see Table II).

A simplistic option, especially in a RESTful context, are *CRUD-style Operations on Resources* which are discussed in 16 sources. *CRUD-style Operations on Resources* designs operations like primitive data store operations. This practice, while commonly used, is seen negatively for many forces. In particular, in contrast to the options below it is negative for the *Avoiding Exposing Domain Model Details in API* force, and can lead to *Chatty APIs* with bad *Performance* and *Scalability*. It is argued that it can lead to *Interface Design Limits Domain Model Design*, various *Maintainability* issues including *Coupling* issues, *Reliability* problems, and *Data Consistency* problems. On the positive side, *CRUD-style Operations on Resources* are simple and good for *API Understandability*.

The option to expose *Domain Operations on Resources* has the second highest in a number of evidences (15 sources). *Domain Operations on Resources* designs are focused on coarser-grained, explicit domain operations. This option is usually positive on the mentioned forces, but needs more design work when mapped to a RESTful API. A variant of it is *Encode Operations as Commands in the Payload* which



How to Design the Operations of a Resource? : Decision
decide for all instances of
Operation : Domain Class
«Option» {name = "Design Coarser Grained, Explicit Domain Operations"}
«Option» {name = "Group Operations on Resource and Select Operations in the Payload"}
«Option» {name = "Expose State Transition Domain Events to Clients"}
«Option» {name = "Expose Event Feed Via a Feed or Publish/Subscribe to Clients"}
«Option» {name = "Design Operations Like Primitive Datastore Operations"}
«Extension» {how = "events might be exposed to clients via feeds or publish/subscribe"}
Domain Operations on Resources : Practice
Variant
Encode Operations as Commands in the Payload : Practice
Can Use
Expose Segregated Command and Query Resources in API : Practice
Expose Domain Events as State Transitions : Practice
Expose Domain Events via Feeds or Pub/Sub : Practice
CRUD-Style Operations on Resources : Practice

Fig. 8. Operation Design Decision

can be harder to understand than domain operations exposed via other means such as protocol features (if supported by the protocol). It is often used e.g. when CQRS is exposed in the API, as then the operation commands can be encoded in a similar way as the queries.

Many microservice systems are event-based systems. In such systems, another option is to *Expose Domain Events as State Transitions* (or its variant *Expose Domain Events via Feeds or Pub/Sub*). These options are also positive for most of the mentioned forces. They can even lead to a better solution for *Exposing Domain Model in API*, if events are used to model the domain, and/or improving *Scalability*. Events can be harder to understand and thus these options also can have some issues with regard to *API Understandability*.

V. Discussion

A. How Practitioners Understand DDD-Based Microservice API Design

In the previous section, we have presented detailed findings on each of our research questions. **RQ1** investigates what the possible architectural design decisions and corresponding decision options in DDD-based microservice API design are. We have found evidence for six principal ADDs with 27 decision options. It is possible to classify those ADDs based on their context and DDD design level. The *Domain Model Mapping Decision* in Section IV-A and the *API as Contract Decision* in Section IV-B clearly focus on DDD elements of Strategic Design, such as *Domain Model*, *Bounded Context*, and *Shared Kernel*. On the API side, they focus on coarse-grained API concepts such as the whole *API* or the *API Contract*. The next set of decisions, i.e., the *Designing API Resources Decision*, and the *Resource Segregation Decision*, focus at the transition from Strategic to Tactical Design considering atomic model elements such as *Entities* but also composite structures such as *Aggregates* or *Processes*, as well as *Bounded Contexts*. Finally, the *Link Mapping Decision* and the *Operation Design Decision* focus on detailed mapping options. That is, it is considered how links are mapped to messages and how DDD operations are mapped to API operations.

RQ2 considers what the relevant decision drivers in those ADDs are. We have observed 27 relevant decision drivers as shown in Table II From this detailed list we can generalize a number of main concerns API developers care about. Please note that GT aims to explain phenomena that have been observed to exist [32]; that is the number of evidences listed in Table II can at most provide a rough indication of force importance. More interestingly, the forces can be categorized into a number of recurring themes:

- A number of forces focus on a *loosely coupled relation of API/its clients and the Domain Model* (f2, f16, f17, f21, f23) which is related to *independent modifiability and evolvability of the API* (f9, f10) as well as *API stability* (f24, f1) in relation to Domain Model changes (f25).
- Various forces concern *maintainability aspects of the API e.g. complexity and understandability* (f18, f7, f14, f22).
- Some forces consider the *consistency of data* (f11, f20).
- A number of forces are related to *runtime properties of the API* including performance and scalability (f3, f4, f5, f6), bandwidth use (f12), and reliability (f19).
- The *relation of API client and server (API provider)* is also important; it should be loosely coupled (f13), usable (f8), and consider different types of API clients (f27).
- *Costs and effort* are reappearing as forces (f15, f26).

RQ3 investigates which relations the decisions and decision options have. Here, the decisions define most of the relations via their options in our model. While the first two decisions *Domain Model Mapping Decision* and the *API as Contract Decision* mainly need to be decided once per API/Domain Model, lower-level decisions need to be made repeatedly. For example, the *Designing API Resources Decision* needs to be made for each *Identified Interface Element* in the *Domain Model* and those *Identified Interface Elements* can change based on prior decisions. Thus there are *no unequivocal recommendations* in the practitioner literature, such as "*Aggregate Roots as API Resources* work best in the most common design situation and the other work well in certain niches". Rather a deliberate, incremental, and iterative human (often collaborative) design approach required. Always decisions depend on project context, goals, and requirements. For instance, for the *Designing API Resources Decision* (as an example), the advice is given to start off considering *Aggregate Roots as API Resources*. If they do not provide a well working abstraction, *Domain Services* or *Processes* can be considered, if applicable. If those do not work well either, then maybe smaller-scale *Entities* or coarser *Bounded Contexts* might provide a well working solution. Similar considerations need to be made in the contexts of the *Resource Segregation Decision*, *Link Mapping Decision* and *Operation Design Decision*. Such a deliberate, incremental design process to map domain models to technical realizations is commonly suggested in the practitioner literature. For example, in the related area of identifying microservice boundaries, one of our sources (s18) suggests a very similar process of first analyzing *Bounded Contexts*, then considering *Aggregates*, then analyzing *Domain Services*, and finally considering non-functional requirements[5]. In this context, it is also interesting to observe that the recommended practices for microservice identification and API or API resource identification differ. Just exposing all microservices in system in an API without a deliberate, incremental API design effort, as explained above, might lead to bad API designs.

In addition to these main relations, the two decisions *Domain Model Mapping Decision* and *API as Contract Decision* have options that are closely related, as shown in Figure 4. Finally, a couple of decision options from the designing API resources, link mapping, and resource segregation decisions have relations to well-established software patterns such as *CQRS*, *Processing Resource*, *Linked Information Holder*, and so on, as detailed above.

Our resulting formal model describes a theory with a precise classification (or categorization) of model elements, as design decision, related context, decision options, decision driver, and related decisions and options. The model reveals interesting insights on which conceptual elements are present in practitioner discussions. Our work can help scientists to gain insights into current practitioner views, whereas practitioners can get a consolidated view integrating many practitioner views based on empirical research methods.

B. Threats to Validity

As GT is mainly concerned with phenomena that have specifically been observed to exist, threats to the results' validity are mainly restricted to inappropriate conceptualization [32]. There is a risk that generalizing from some of our results might be misleading, but as we do not claim completeness and use a rather high number of sources (i.e., more than needed for theoretical saturation), it is likely that generalization is valid to at least some extent. At any rate, our results are only valid in our set scope.

To increase internal validity or credibility, we decided to use practitioner reports that were produced independently from our study. This avoids bias, e.g. compared to interviews in which the practitioners would have known that their answers are used in a study. However, this introduces a different threat: Some important information might be missing in the reports, which would have been revealed in interviews. We tried to mitigate this threat by looking at more sources than needed to reach theoretical saturation, as it is unlikely that all different sources miss the same important information. Different members of the author team have cross-checked all models independently to minimize researcher bias. The threat to 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. 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 today. However, 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.

VI. Conclusions

We have studied the relation of DDD practices and microservice API design. We have identified six ADDs with 27 decision options concerning the mapping of domain models to APIs, defining API contracts in relation to domain models, designing API resources based on domain model elements, segregation of API resources, mapping domain model links to the API, and designing the operations of an API resource. In addition, we identified 27 decision drivers (forces) considered by practitioners in those decisions, which can be broadly categorized into forces on *loosely coupled relation of API/its clients and the Domain Model*, *maintainability aspects of the API such as complexity and understandability*, *consistency of data*, *runtime properties*, the *relation of API client and server (API provider)*, and *costs and effort* (see Section V). Finally, we identified numerous decision-option relations and other relations. Those usually require a deliberate, incremental human design effort. Our results can help scientists to get a better understanding of practitioner concerns, and practitioners to get an overview of the current view of other practitioners on the interrelation of microservice APIs and DDD. Moreover, the design guidance by our ADD model also can help to reduce design efforts and risks. As future work, we plan to use our findings to provide automated design advice to API designers and improve tools that generate API code. We plan to perform research on further ADDs model elaboration and validation. Further research on related practices such as event storming and event sourcing might be interesting as well.

Acknowledgments

This work was supported by the API-ACE project funded by the FWF (Austrian Science Fund) project I 4268.

References

[1] O. Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, vol. 32, no. 3-4, pp. 301–310, Jul. 2017. [Online]. Available: https://doi.org/10.1007/s00450-016-0337-0

[5]See: https://docs.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis

[2] O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, and U. Zdun, "Introduction to microservice api patterns (map)," *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*, vol. 78, no. 4, pp. 1–17, 2020.

[3] E. Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Reading, MA.: Addison-Wesley, 2003.

[4] ——, "Ddd and microservices: At last, some boundaries!" https://www.youtube.com/watch?v=sFCgXH7DwxM, 2016.

[5] V. Vernon, *Implementing Domain-Driven Design*. Boston, USA: Addison-Wesley Professional, 2013.

[6] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. New York, NY: de Gruyter, 1967.

[7] J. Corbin and A. L. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociology*, vol. 13, pp. 3–20, 1990.

[8] A. Rainer and A. Williams, "Using blog-like documents to investigate software practice: Benefits, challenges, and research directions," *Journal of Software: Evolution and Process*, vol. 31, no. 11, p. e2197, 2019.

[9] O. Zimmermann, J. Koehler, F. Leymann, R. Polley, and N. Schuster, "Managing architectural decision models with dependency relations, integrity constraints, and production rules," *J. Syst. Softw.*, vol. 82, no. 8, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2009.01.039

[10] M. Henning, "Api design matters," *Queue*, vol. 5, no. 4, pp. 24–36, 2007.

[11] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? an empirical study on the directives of api documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012.

[12] L. Shi, H. Zhong, T. Xie, and M. Li, "An empirical study on evolution of api documentation," in *International Conference on Fundamental Approaches To Software Engineering*. Berlin, Heidelberg: Springer, 2011, pp. 416–431.

[13] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey, "Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow," Georgia Institute of Technology, USA, Tech. Rep., 2012.

[14] L. Li, T. F. Bissyandé, Y. Le Traon, and J. Klein, "Accessing inaccessible android apis: An empirical study," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Washington, DC, USA: IEEE, 2016, pp. 411–422.

[15] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *2013 IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE, 2013, pp. 70–79.

[16] S. Scalabrino, G. Bavota, M. Linares-Vásquez, M. Lanza, and R. Oliveto, "Data-driven solutions to detect api compatibility issues in android: an empirical study," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. Washington, DC, USA: IEEE, 2019, pp. 288–298.

[17] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. ACM, 2018, pp. 153–163.

[18] A. Hora, R. Robbes, M. T. Valente, N. Anquetil, A. Etien, and S. Ducasse, "How do developers react to api evolution? a large-scale empirical study," *Software Quality Journal*, vol. 26, no. 1, pp. 161–191, 2018.

[19] W. Wu, A. Serveaux, Y.-G. Guéhéneuc, and G. Antoniol, "The impact of imperfect change rules on framework api evolution identification: an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1126–1158, 2015.

[20] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How does web service api evolution affect clients?" in *2013 IEEE 20th International Conference on Web Services*. Washington, DC, USA: IEEE, 2013, pp. 300–307.

[21] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Washington, DC, USA: IEEE, 2013, pp. 5–14.

[22] H. Zhong and H. Mei, "An empirical study on api usages," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 319–334, 2017.

[23] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "On the use of replacement messages in api deprecation: An empirical study," *Journal of Systems and Software*, vol. 137, pp. 306–321, 2018.

[24] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[25] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers, "Api designers in the field: Design practices and challenges for creating usable apis," in *2018 ieee symposium on visual languages and human-centric computing (vl/hcc)*. IEEE, 2018, pp. 249–258.

[26] U. Zdun, M. Stocker, O. Zimmermann, C. Pautasso, and D. Lübke, "Guiding architectural decision making on quality aspects in microservice apis," in *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018*, ser. LNCS, vol. 11236. Springer, 2018, pp. 73–89.

[27] S. Kapferer and O. Zimmermann, "Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling," in *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020*. Scitepress, 2020, pp. 299–306.

[28] C. Richardson, "A pattern language for microservices," http://microservices.io/patterns/index.html, 2017.

[29] T. Górski and E. Wojtach, "Use case api-design pattern for shared data," in *2018 26th International Conference on Systems Engineering (ICSEng)*. Washington, DC, USA: IEEE, 2018, pp. 1–8.

[30] V. Garousi, M. Felderer, M. V. Mäntylä, and A. Rainer, "Benefitting from the grey literature in software engineering research," 2019.

[31] K. Charmaz, *Constructing grounded theory*. sage, 2014.

[32] F. Zieris and L. Prechelt, "On knowledge transfer skill in pair programming," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM'14. New York, NY, USA: Association for Computing Machinery, 2014.

[33] M. Fowler, *Patterns of Enterprise Application Architecture*. USA: Addison-Wesley, 2002.

[34] M. Voelter, M. Kircher, and U. Zdun, *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. Hoboken, NJ, USA: J. Wiley & Sons, 2004.