Conformance Assessment of Architectural Design Decisions on API Endpoint Designs Derived from Domain Models

Apitchaka Singjai, Uwe Zdun

University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Austria, `firstname.lastname@univie.ac.at`

Abstract

Context: Domain-driven design (DDD) is commonly used to design microservices. A crucial aspect of microservice design is API design, which includes the design of API endpoints.
Objective: Our objective is to automate the assessment of conformance to Architectural Design Decisions (ADDs) on the interrelation of DDD and APIs. In particular, we studied link mapping, API operation design, and resource segregation as API endpoint design issues that are linked to domain model design. We particularly aim to address conformance checking in the context of frequent release practices, as frequent manual conformance checking is difficult or infeasible.
Method: We suggest a new approach for the automated assessment of conformance to ADD options. The approach suggests automated detectors to detect ADD options selected in a given API endpoint design, as well as an assessment scoring scheme based on empirical results. For the evaluation of our approach, we first manually created a ground truth for 12 cases in a multi-case study, and then compared the results of our automated detectors to the ground truth for each of those cases.
Results: With our approach, all ADD options in our multi-case study possibly can be automatically detected. Without further improvements, our approach identifies 83% of the decision points in the multi-case study correctly. A statistical analysis of our data shows only a negligible effect size for differences to the ground truth.
Conclusions: Our new approach provides a pragmatic method for automated detection of conformance to ADDs on the interrelation of DDD and APIs. The approach can support the continuous analysis of API endpoint designs.

Keywords: Microservice Architecture, API design, Domain Driven Design, Architecture Conformance Assessment.

1. Introduction

Microservice architectures consist of independently deployable, scalable, and changeable services [1, 2, 3]. In microservices and related architectures, message-based APIs, such as RESTful HTTP, queue-based messaging, or event-based message exchanges, dominate over remote procedure calls [4, 2]. A critical aspect in designing a microservice architecture is API design which, in this article, is seen as all activities of planning and making design decisions when building an API [5, 4]. It includes aspects such as which microservice operations should be offered in the API, how to exchange data between client and API, how to offer links, and how to represent API messages [2, 4].

Domain-Driven Design (DDD) [6, 7] is a set of concepts and patterns that help in designing software systems based on the underlying model of the (business) domain, the Domain Model [8]. It is often used as a foundation to identify and design microservice architectures [9, 10, 11].

Architectural Design Decisions (ADDs) focus on the architecturally significant design decisions of a system, in the sense that a single decision change could significantly affect its entire architecture [12]. In the field of microservice APIs and their relations to Domain Models, many central API design problems revolve around API endpoint design. An *API endpoint* is the provider-side end of a communication channel and a specification of where the API endpoints are located so that APIs can be accessed by API clients [4, 2]. Changes in the API endpoints can significantly affect the architecture of the API clients and of the microservices in the backend serving the API [4].

In our study, we selected three empirically grounded ADDs from our prior work [11]. In particular, we studied ADDs on detailed API endpoint design explaining (1) how to derive links offered in API endpoints from the links in the domain model; (2) how to design the operations of an API endpoint; and (3) if and how to segregate the resource represented by the API endpoint. Please note that these are all architectural design issues relevant only for remote, message-based APIs. While the overlap between a DDD model and an API can exist in a local context, too, none of the issues addressed in the studied ADDs appears for local APIs or API libraries (see Section 2 for detailed explanations).

Many existing practitioner works use DDD to identify and model the microservices which are exposed via APIs [11, 4, 9, 10]. If DDD or domain modeling, in general, is used in an API project, the APIs should have a traceable mapping to the Domain Models of the microservices. In each evolution step of either the API or the microservice models, the respective other parts might have to be changed accordingly. To safeguard such evolution steps in practice, it is required to assess whether a given microservice API endpoint design correctly realizes the mapping to its microservice domain model and vice versa.

This is especially problematic if such an assessment is needed continuously, e.g.

in the context of continuous integration/delivery (CI/CD). Just consider today's large-scale microservice deployments that are deployed with high frequency (e.g. at least daily), such as those of Uber [13], Google [14], or Netflix [15]. To manually assess for each and every service, whether the mapping to the Domain Models and other such constraints imposed by backend or client architectures are still in place and correct, would be an extremely laborious and error-prone manual task.

This kind of assessment can be classified as architecture conformance assessment. Conformance is generally defined as the consistency relation between models [16]. In our API endpoint design context, conformance concerns consistency in the mapping of API models to DDD models according to the relations from our empirically studied ADDs.

To address this problem, we aim to automate the assessment of conformance to ADD options in API endpoint designs. Some might argue that APIs are intended to change rarely, and thus such automation is not required. But actually, there are many more reasons for API evolution [17], meaning that frequent system changes usually lead to frequent API changes. Also, our approach is on the links of APIs to DDD models. So, if the API stays stable but the DDD model changes, still conformance must be checked. Such an automated assessment is mainly interesting to Web API developers and architects, but also Web API client developers are indirectly affected, as they benefit in their work from consistent and stable API designs.

Here, we phrase our study design following the Goals/Questions/Metrics (GQM) approach [18]: It is the **research goal** of this article to *study how to provide support for assessing how well a microservice API is conforming to a Domain Model it was derived from.* In particular, we aim to study this goal in the context of three common API design decisions: (1) API endpoint operations in relation to Domain Model operations, (2) API links in relation to Domain Model links, and (3) resource segregation of API operations in relation to Domain Model operations. To address our research goal, we need to answer the following **research questions**:

- **RQ1** For which decision options in microservice API design decisions can we provide automated support for assessing conformance to the microservice API's Domain Model?
- **RQ2** To which extent can we assess conformance of microservice API design decisions to the microservice API's Domain Model?

The goal fulfillment will be measured using a couple of **metrics**: We use simple count-based metrics to measure for how many decision options our approach provides automatic conformance assessment. We also use count-based metrics for measuring the extent to which automated conformance assessment is supported; in addition, we use statistical methods to measure the effect size.

In particular, we performed a detailed study of the decision driver impacts in the decision options to derive an empirically grounded scoring scheme for the assessment of API endpoint designs, based on the empirical data from our prior study [11] (only summarized in this article). We collected and modeled 12 cases in a multi-case study and manually assessed each case. This way we created a ground truth for evaluating our automated approach later on. Next, to support the automated assessment we developed detectors to identify each of the decision points. Finally, we mapped the detector results to our ground truth in order to automatically assess the cases in our case study evaluation.

This paper is organized as follows: Next, in Section 2 we describe the three API endpoint ADDs from our prior study as background and an illustrative example for these ADDs. Then we describe in Section 3 our research methods. We give an overview of the case in our multi-case study and the scoring scheme in Section 4. Our automated detectors are described in Section 5, followed by a discussion of our multi-case study results, a statistical analysis, and lessons learned in Section 6. Finally, we compare our findings to related work in Section 7, discuss threats to validity in Section 8, and draw conclusions in Section 9.

2. Background and Illustrative Example

In this section, we discuss three ADDs on API endpoint design from our prior study [11] as the background of this work, summarized in Table 1 using their core decision elements (cf. [19]). In our prior study, we have conducted a Grounded Theory study [20] based on the 32 grey literature sources (i.e., practitioner sources such as blog posts or system documentations [21]). Then, we present the ADDs along with their relations (to decision options, other decisions, and patterns/practices) and their decision drivers. All decisions drivers used in this section are defined in Table 2. We define the ADDs and decision drivers here in detail to highlight the specific challenges that remote, message-based API architects are facing and which are in the focus of our approach (as opposed to e.g. the challenges of local API design).

This article assumes familiarity with basic DDD concepts, in particular: Evans [6] classifies domain objects into types such as *Entities*, *Value Objects*, and *Services*, which are then used to identify larger structures such as *Aggregates*. Such elements contain domain links in the domain model, as well as Domain Operations and/or Domain Events. At the next higher abstraction level, DDD introduces the notion of Strategic Design [6, 7] which explains how to structure large domains into a number of *Bounded Contexts* and their relations. The ADDs described here explain how to derive API endpoint designs from the information in such DDD domain models.

2.1. Link Mapping Decision (LMD)

In APIs, the links between API endpoints play a central role. In DDD the links between model elements have a similar role. Obviously not all links in the DDD

Table 1: Overview of ADDs on API Endpoint Design

ADD	**Problem Statement**	**Decision Context**	**Decision Options**	**Decision Drivers**
Link Mapping Decision (LMD)	How to map links between domain model elements to the API?	Links in the domain model	• Use Distributed or Hypermedia Links in the Payload. • Pass Object Identifiers in the Payload. • Embed Linked Data in the Payload. • Do not offer the Domain Model Link in the API.	• Data Consistency • API Evolvability • API Modifiability • Message Size • Protocol Complexity • Minimize API Calls • Performance • Scalability • Avoid Exposing Domain Model Details in API • Coupling of Clients to Server
Operation Design Decision (ODD)	How to design the operations of a resource?	Operations in the domain model	• CRUD-Style Operations on Resources • Expose Domain Events as State Transitions • Expose Domain Events via Feeds or Pub/Sub • Domain Operations on Resources • Encode Operations as Commands in the Payload	• Avoid Exposing Domain Model Details in API • Protocol Complexity • Minimize API Calls • Performance • Scalability • Avoid Interface Designs that Limit Domain Model Designs • Maintainability • Coupling of Clients to Server • Data Consistency • (API) Understandability
Resource Segregation Decision (RSD)	Segregate Resources for Reading and Updating Information in an API?	Identified interface elements in the domain model	• Expose Segregated Command and Query Resources in API • Do not Segregate Queries and Commands in an API	• Scalability • Eventual Consistency Support • API Understandability • Data Consistency

Table 2: Glossary of Decision Drivers Used in the ADDs

Decision Driver	**Definition**
Data Consistency	"[Data] consistency [...] can be seen as the guarantee that transactions started in the future see the effects of transactions committed in the past" [22].
API Evolvability	"The ability of [an API (was: a system)] to accommodate changes in its requirements throughout the system's lifespan with the least possible cost while maintaining architectural integrity" [23].
API Modifiability	"Modifiability describes that the code [and design of the API realization] facilitates the incorporation of changes, once the nature of the desired change has been determined" [24].
Message Sizes	The smaller the message sizes in an API, the less bandwidth is used. Larger messages can help avoid sending multiple distributed messages.
Protocol Complexity	Many links in API messages lead to a complex interaction protocol.
Performance	"Performance prescribes conditions on functional requirements such as speed, efficiency, availability, accuracy, throughput, response time, recovery time, and resource usage" [24].
Minimize API Calls	Each API call costs the performance of a distributed invocation, which is usually much higher than for many local operations.
Scalability	"Scalability is the ability to handle increased workload (without adding resources to a system)" [25].
Exposing Domain Model Details in API	The API should be an abstraction of the Domain Model only revealing those aspects needed for the API functions and nothing more.
Coupling of Clients to Server	API clients and servers should be as loosely coupled as possible, so that independence maintenance, testing, and motification is possible.
Interface Designs that Limit Domain Model Designs	An API design should not limit the design of the Domain Model, but provide an abstraction of it.
Maintainability	"The ability to undergo changes, including error correction and system adaptation" [24].
API Understandability	"[API] understandability describes that the purpose of the [API design and] code is clear to the inspector." [24].
Eventual Consistency Support	Eventual consistency describes a weak consistency relation which requires that all replicas of an object (here: API/DDD elements) will only eventually reach the same correct value.

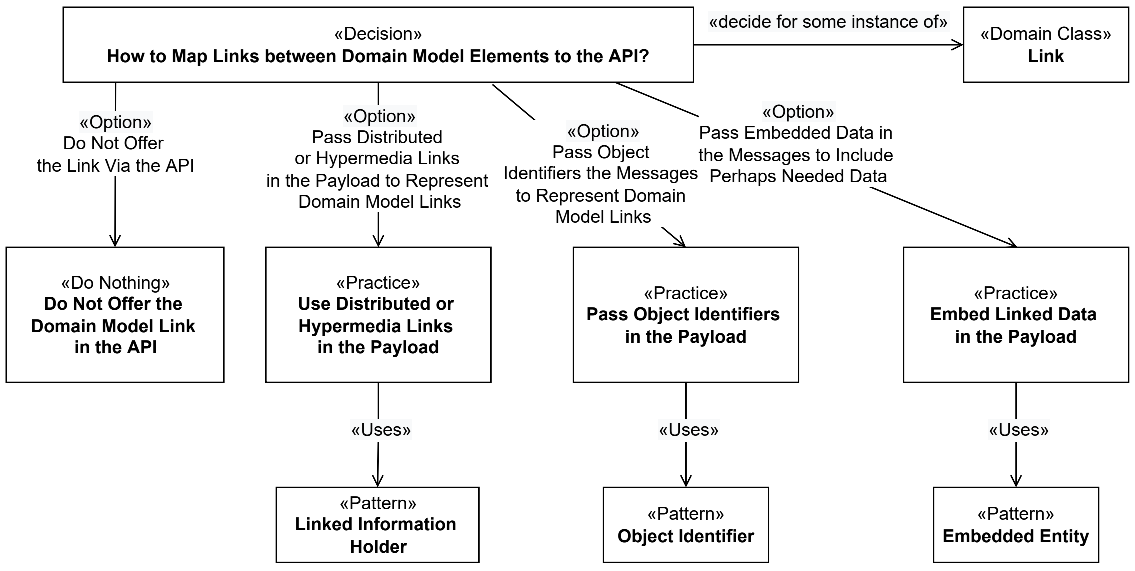
«Decision»
How to Map Links between Domain Model Elements to the API?
«decide for some instance of»
«Domain Class»
Link
«Option»
Do Not Offer the Link Via the API
«Option»
Pass Distributed or Hypermedia Links in the Payload to Represent Domain Model Links
«Option»
Pass Object Identifiers the Messages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

Figure 1: Link Mapping Decision

model are candidates to be exposed in an API, but only those between the model elements offered as API endpoints. The Link Mapping Decision with its decision options and links is shown in Figure 1. One decision option is *Use Distributed or Hypermedia Links in the Payload*. It means to use standard distributed/hypermedia links, such as URIs in RESTful HTTP or URIs and HAL/JSON-LDs in JSON. An alternative is *Embed Linked Data in the Payload* where the content is added to the message payload instead of linking to it.

Finally, there is the option to *Pass Object Identifiers in the Payload* which sends 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*, making 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*.

2.2. Operation Design Decision (ODD)

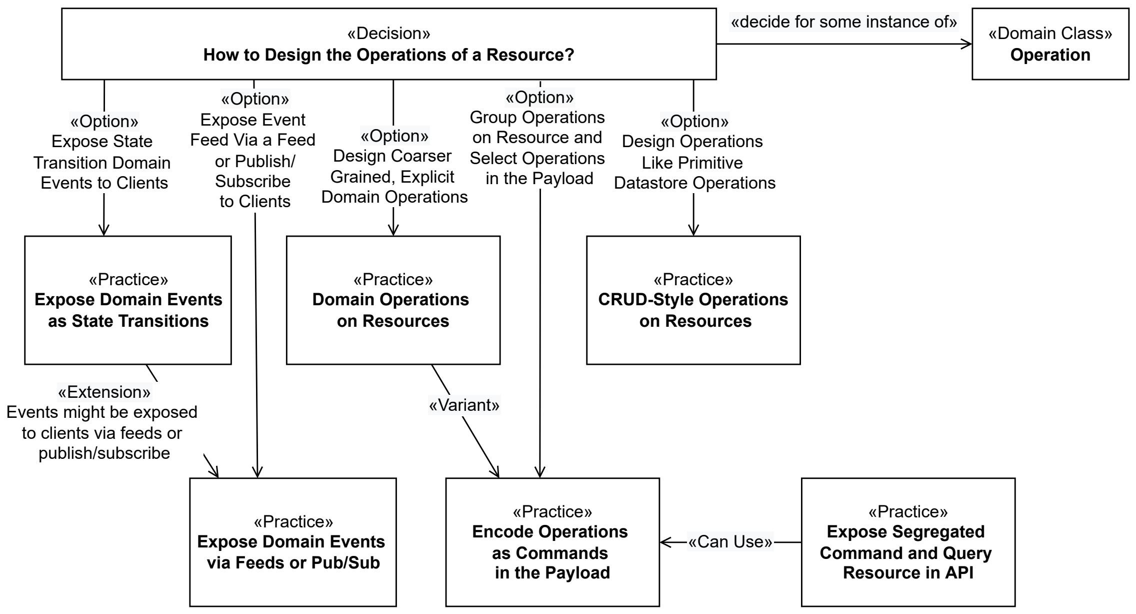
«Decision»
How to Design the Operations of a Resource?
«decide for some instance of»
«Domain Class»
Operation
«Option» Expose State Transition Domain Events to Clients
«Option» Expose Event Feed Via a Feed or Publish/Subscribe to Clients
«Option» Design Coarser Grained, Explicit Domain Operations
«Option»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

Figure 2: Operation Design Decision

The ODD decision, shown in Figure 2, is on how to design the operations of an endpoint. A simplistic option, especially in a RESTful context, are *CRUD-style*

Operations on Endpoints, which designs operations like primitive data store operations. This practice, while commonly used, is seen negatively for many decision drivers. In particular, in contrast to the options below it is negative for *Avoiding Exposing Domain Model Details in API* and can lead to Chatty APIs, i.e. *Protocol Complexity* because of many small messages (*Minimize API Calls* not followed) with bad *Performance* and *Scalability.* It is argued that it can lead to *Interface Designs that Limit Domain Model Designs*, various *Maintainability* issues including *Coupling of Clients to Server* issues, and *Data Consistency* problems. On the positive side, *CRUD-style Operations on Endpoints* are simple and good for *API Understandability.*

The option to expose *Domain Operations on Endpoints* leads to more coarse-grained designs. This option is usually more positive on the mentioned decision drivers, but needs more design work when mapped to a RESTful API. A variant of it is *Encode Operations as Commands in the Payload* which can be harder to understand than domain operations exposed via other means such as protocol features. It must further be noted that the distinction into good domain operations and bad CRUD-like operations is not valid either. In many cases, a CRUD-like operation is the best suited option. The actual decision criterion is: Is a further abstraction of the operation possible and does it make sense? For that reason, it is often beneficial that operations are exposed from an aggregated domain model element, such as an Aggregate, its Aggregate root, a Bounded Context, or a Service.

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 the Domain Model in the API*, if events are used to model the domain, and/or improving *Scalability.* Events can be harder to understand and thus these options can have some issues with regard to *API Understandability.*

2.3. Resource Segregation Decision (RSD)

For API Endpoints, 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 [26]. The idea of CQRS is to use a different model to update data than the model that is used to read data. The Resource Segregation Decision is shown in Figure 3.

The CQRS 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 lower *API Understandability* due to API complexity and less *Data Consistency.* Thus, typically, it shall not be used, if simple domain model elements (e.g. just a few Entities or Value Objects) are offered on an API endpoint. Usually, it makes sense to apply this option together with event-based API operations. For

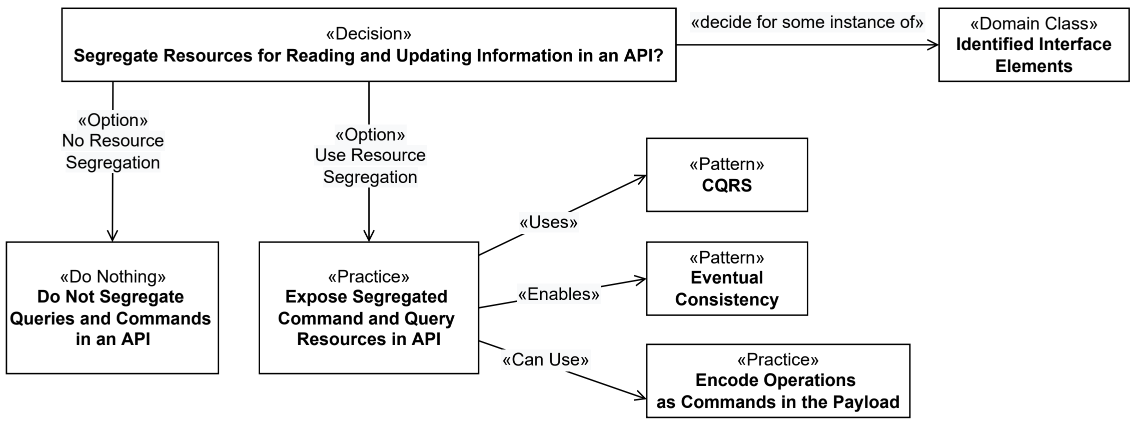
«Decision»
Segregate Resources for Reading and Updating Information in an API?
«decide for some instance of»
«Domain Class»
Identified Interface Elements
«Option»
No Resource Segregation
«Option»
Use Resource Segregation
«Do Nothing»
Do Not Segregate Queries and Commands in an API
«Practice»
Expose Segregated Command and Query Resources in API
«Uses»
«Pattern»
CQRS
«Enables»
«Pattern»
Eventual Consistency
«Can Use»
«Practice»
Encode Operations as Commands in the Payload

Figure 3: Resource Segregation Decision

the configuration, we have chosen the case studies that be able to define the ADD options.

2.4. Illustrative Example

Figure 4 shows an excerpt of a domain model, designed using DDD (adapted from the Case Study PM [27]). Such models are frequently used as a starting point to identify microservices and derive microservice APIs [9, 10, 11]. In the model, an Aggregate Paper Archive Facade is designed for managing Paper Collections, which is itself an Entity using Paper Item Entities and having Paper Item Keys as Value Objects. For exposing the Paper Archive, further a Service is designed, which contains three Domain Operations: Lookup Paper from Authors, Create Paper Item, and Convert to Markdown.

Many designs are possible to map this simple DDD model excerpt to an API. The one chosen in the PM Case Study [27] is provided in Figure 5. One API Endpoint is designed for the Paper Archive Facade, its Service, and the Collection Entity. Thus, it was chosen to not provide the domain model links from the Aggregate to the Service and Entity as explicit elements of the API model (i.e., the Do Nothing option of LMD is chosen here). The three domain operations of the Service are exposed via same-named API Operations. As Create Paper Item is a simple creation function, the CRUD-based API Operation option of the ODD decision was chosen here. The two other API operations were mapped as Domain-Based API Operations, as these operations provide coarser-grained operations, closer to the domain design. None of them uses the Encode Operations as Commands in the Payload variant from the ODD decision.

The Paper Collection Entity and the Paper Archiving Service have links to Paper Item and Paper Item Key. Here, the Embedded Data option of LMD is chosen for mapping all those links. Consequently, the Paper Item Entity and Paper Item Key Value Object are mapped to API Data Types. Further, decisions about the precise variant of the Embedded Data option of LMD are made: whether the data is required or provided, and if the data elements are usually needed by clients or not (default value: False).

Finally, for technically realizing the API, another API Data Type for the parameters of the paper creation (not modeled in the domain model) and a String type are needed. The String type is used once to provide an Object-Based Link as another option of LMD, for the other ones again the Embedded Data option is chosen.

Resource Segregation is not used in this API. Thus for RSD the option Do Nothing was chosen for all interface elements.

While this illustrative example is rather simple, many other possible designs could have been chosen even in this small-scale example that would have led to very different API designs. For instance, designers could have decided to expose all domain model links as hyperlinks, or resource segregation could have been offered as well. Assessing such designs, especially if they grow substantially larger and/or if the assessment is needed continuously, is tedious and error-prone. Our approach aims to automate the model-based assessment of the conformance between API and domain model designs.

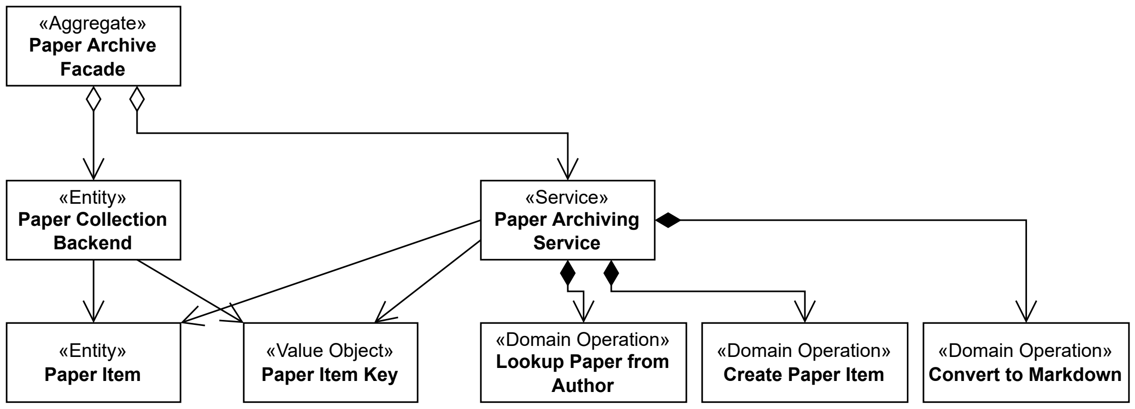

Figure 4: Example of a DDD Model (Excerpt from Case Study PM (adapted from [27])

3. Research Methods

Figure 6 illustrates the research methods applied in our study. Please note that we have applied a very similar combination of research methods in earlier works [28, 29] before. To provide an audit trail of the research and enable repeatability of the study, we provide open access to our the data set and the source code[1].

[1]We provide derived coded models in Python, and generated models (in UML, Markdown, and Latex) as a replication package for download in the Zenodo long-term open access archive [30].

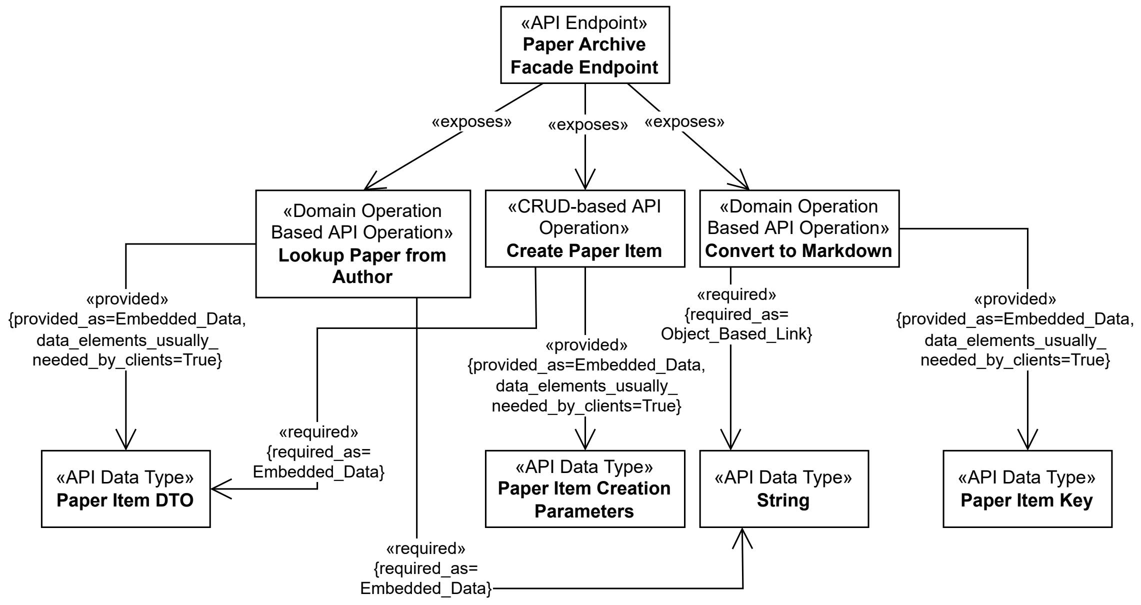
«API Endpoint» Paper Archive Facade Endpoint
«exposes»
«exposes»
«exposes»
«Domain Operation Based API Operation» Lookup Paper from Author
«CRUD-based API Operation» Create Paper Item
«Domain Operation Based API Operation» Convert to Markdown
«provided» {provided_as=Embedded_Data, data_elements_usually_ needed_by_clients=True}
«required» {required_as= Object_Based_Link}
«provided» {provided_as=Embedded_Data, data_elements_usually_ needed_by_clients=True}
«provided» {provided_as=Embedded_Data, data_elements_usually_ needed_by_clients=True}
«required» {required_as= Embedded_Data}
«API Data Type» Paper Item DTO
«API Data Type» Paper Item Creation Parameters
«API Data Type» String
«API Data Type» Paper Item Key
«required» {required_as= Embedded_Data}

Figure 5: Example of an API Model (Excerpt from Case Study PM (adapted from [27])

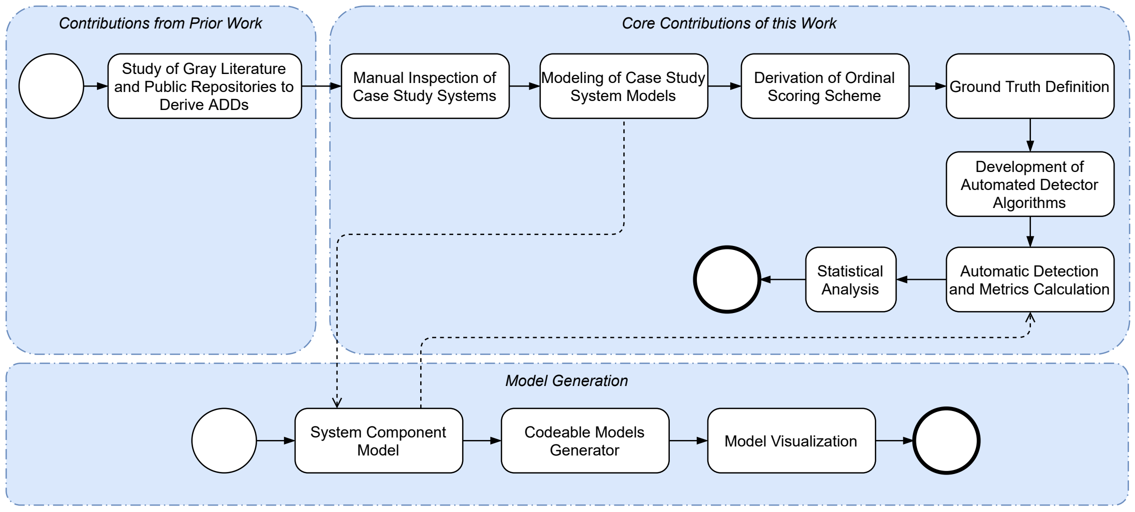
Contributions from Prior Work
Core Contributions of this Work
Study of Gray Literature and Public Repositories to Derive ADDs
Manual Inspection of Case Study Systems
Modeling of Case Study System Models
Derivation of Ordinal Scoring Scheme
Ground Truth Definition
Development of Automated Detector Algorithms
Automatic Detection and Metrics Calculation
Statistical Analysis
Model Generation
System Component Model
Codeable Models Generator
Model Visualization

Figure 6: Research Methods Overview

Gray Literature Study of ADDs on API Endpoint Designs Derived from Domain Models. In our previous work [11], we have empirically identified ADDs on deriving APIs from DDD [6] models. In particular, we conducted a Grounded Theory study [20] based on 32 gray literature [21] sources (i.e., practitioner sources such as blog posts or system documentations) concentrating on the interrelation of microservice API design and DDD. We have identified ADDs frequently employed by practitioners, along with decisions options, decision relations, and decision drivers. For this study, we selected the three ADDs summarized in Section 2, which are all focusing on API endpoint design related issues. We have selected those ADDs, as many central API design problems revolve around API endpoint design [4, 2].

Multi-Case Study Preparation and System Modeling. In the next steps, we manually inspected 12 case studies of systems realized by practitioners, either from the published source code of these systems in public repositories, their system documentations, or both. In this, we followed the guidelines for conducting and reporting case study research in software engineering by Runeson et al. [31].

We used major search engines (Google, Bing, DuckDuckGo, Baidu) and topic portals (InfoQ, Dzone, TechBeacon) to find relevant cases. One major concern about search engines in such research is their search algorithm because the results are dependent on the user [32]. To avoid personal bias in the research, we used as search words those keywords that provided decision categories (i.e. the most general categorization) in our detailed prior study of the gray literature [11], in particular: “Application Programming Interface” or “API”, “Domain Driven Design” or “DDD”, and “Microservices” (all in singular and plural form). We had to check that the found cases contained information on their domain modeling, which substantially limited the possible cases we could consider. Further, we checked for each found case whether authors who realized it have a substantial background in industrial practice. Finally, we selected systems from many different domains covering a broad range of our ADDs’ decision options and their combinations (for details see Table 3 explained below).

We assume that our evaluation systems are, or reflect, real-world practical examples of microservice architectures. As many of them are open source systems with the purpose of demonstrating practices or technologies, they are at most of medium size and modest complexity, though.

We then defined models and meta-models to precisely model the case study systems as UML models. We used our existing CodeableModels tool[2], a Python implementation for precisely specifying meta-models, models, and model instances in code. Based on CodeableModels, we realized automated code generators to generate graphical visualizations of all meta-models and models in PlantUML[3].

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

[3]https://plantuml.com/en/

Please note that this modeling step was done manually in our work for most of the case study systems. However, in our prior work, we have shown that it can be automated, too. Our existing static code analysis approach for architecture reconstruction of polyglot microservice systems [33] can be applied here. Due to the polyglot nature of microservice systems, this requires modest initial specification effort, though. So far, only two of the APIs of the systems in Table 3 have been automatically reconstructed, namely, the ES and LS API models (see [33]).

To illustrate the manual effort, in [33] we compared the lines of code: For the 35.4 KLoC of code in LS, we required 804 LoC of specification code; for the 81.4 KLoC of code in ES, we required 716 LoC of specification code. As those are among the largest models in our model set, it is highly likely that the other system models can be reconstructed with significantly less effort. Please note that in [33] we only reconstructed the API part, not the DDD model, but the DDD model is usually modeled manually while the API is often changed in the source code or in API specifications such as OpenAPI. Thus this does not limit the possible use of this code extraction technique for automation.

Another way to automate our tool-chain is to specify them using a model-driven approach such as MDSL [27]. From the MDSL models, OpenAPI specifications and API code can be generated automatically. Here, the code is derived from the models, not vice versa. This approach has been realized for our Case Study PM.

Ground Truth Definition. We then performed a systematic assessment on support or violation of the collected ADDs' decision options. For this, we derived an ordinal scoring scheme that is directly based on the decision driver impacts empirically identified in our gray literature study. That is, we manually checked for support or violation of the recommendations empirically derived from the gray literature [11]. The ordinary scale helped us turn qualitative judgments in the practitioner texts into numerical assessments. We used the scoring scheme to manually assess the cases in our multi-case study to create a ground truth (or oracle or gold standard), which we will use later on to validate our approach.

Automated Detector Algorithms and their Application. To automate the assessment provided by the scoring scheme, we developed automated detector algorithms which aim to detect each relevant criterion deciding on scores in the scheme. Furthermore, we implemented each automated detector in Python (based on our coded UML meta-models) and wrote code generators to enable running them on any system model conforming to our meta-models, such as the case study models.

We applied the automated detectors on our case study data set and calculated simple count-based metrics to measure for how many decision options our approach provides automatic conformance assessment and the extent to which automated conformance assessment is supported.

Statistical Analysis. Finally, we performed an empirical assessment of our approach using a multi-case study using the cases inspected. We used the manual assessments in the cases of the multi-case study as a "ground truth" and compared them to our automatically derived scores from the detectors. Besides analyzing and discussing the results on a case-by-case basis for each decision option, we also perform a statistical analysis of the results. In it, we use R's *shapiro.test()* function to perform Shapiro-Wilk normality tests [34]. As the data sets are non-normally distributed, we used the Wilcoxon signed rank test with Pratt method [35], provided by the *wilcoxsign_test()* function from R's *coin* package, to test for a significant difference between the ground truth and the automated detector results. Finally, Cliff's δ [36] is used for effect size estimation via *cliff.delta()* function from R's *effsize* package.

4. Multi-Case Study Preparation and Inspection

This section explains how we prepared and inspected the cases for our multi-case study. Table 3 summarizes the 12 cases which we have manually modeled based on available source code, documentation, descriptions in articles or blog posts, and so on. The table summarizes for each model the size of the cases in terms of modeled domain models and API model elements. We also modeled in each of those models the relations of the elements among each other, and the relations between DDD and API model elements in detail. The third column of the table contains a brief description of the case, the options selected in the case for each of the ADDs from Section 2, and the URL of the original case source.

As an illustrative example, we explain the case study preparation and ADD inspection of the *ES* model. Figure 7 shows the Basket RESTful Endpoint's elements and its associations, as well as the Bounded Context domain model element from which it is derived. Please note that this is just a small excerpt of the *ES* model, and we do not show other domain model elements for sake of brevity here. Most of models contain complex domain models with domain links, domain operations, and many other domain elements, which are related to API elements such as the ones shown in Figure 7.

```
[HttpGet("{id}")]
[ProducesResponseType(typeof(CustomerBasket), (int)HttpStatusCode.OK)]
public async Task<ActionResult<CustomerBasket>> GetBasketByIdAsync(string id)
{
  var basket = await _repository.GetBasketAsync(id);
  return Ok(basket ?? new CustomerBasket(id));
}
```

Listing 1: GetBasketByIdAsync Operation from Basket API

The endpoint offers four *API Operations*, namely *UpdateBasketAsync*, *GetBasketByIdAsync*, *DeleteBasketByIdAsync*, and *CheckoutAsync*. They use four *API Data Types*: (*Customer Basket Data Type*, *Customer Basket ID Data Type*, *Request ID Data Type*, and *Basket Checkout Data Type*).

Firstly, for the LMD decision, we investigated the relationship between *API Operations* and *API Data Types*. For instance, the *GetBasketByIdAsync* operation

Table 3: Overview of Cases in the Multi-Case Study

ID	**Elements**	**Description and Summary of ADD Inspection**
AX	domain: 5 api: 14	Purchase Order System; applies DDD concepts and CQRS to decompose components; no details on link and operation design provided. ADD Options used: **LMD** (N/A), **ODD** (N/A), **RSD** (CQRS on aggregating endpoints, no details on operations). `https://dzone.com/articles/bounded-contexts-with-axon`
PM	domain: 10 api: 11	Publication Management System; applies DDD concepts to API design, detailed API specifications and code generation. ADD Options used: **LMD** (Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Explicit Domain Operations & Design Operations Like Primitive Datastore Operations, exposed to API by an aggregate root), **RSD** (N/A). `https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html`
BA	domain: 8 api: 39	Bank Account System; focuses on event-based architecture using CQRS and event sourcing patterns. ADD Options used: **LMD** (Pass Object Identifiers in the Messages to Represent Domain Model Links & Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Event State Transition Operations, based on Domain Operations & CRUD-based operations, some exposed to API by aggregating services and aggregates), **RSD** (CQRS on aggregating endpoints, event-based operations). `https://github.com/cer/event-sourcing-examples`
OS	domain: 19 api: 18	Online Shop System; uses DDD and pattern-based modeling of APIs. ADD Options used: **LMD** (Pass Object Identifiers in the Messages to Represent Domain Model Links & Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Design Operations Like Primitive Datastore Operations, all exposed to API by Entities), **RSD** (N/A). `https://github.com/socadk/design-practice-repository/blob/master/tutorials`
CM	domain: 12 api: 52	Cinema Microservice System; models multiple frontends and REST-based APIs. ADD Options used: **LMD** (Pass Embedded Data in the Payload, clients usually need the data, except for one operation), **ODD** (Explicit Domain Operations & Design Operations Like Primitive Datastore Operations, all exposed to API by aggregating services), **RSD** (CQRS on aggregating endpoints). `https://github.com/Crizstian/cinema-microservice`
ES	domain: 4 api: 142	E-Shop on Containers System; follows the CQRS pattern; uses event transitions and pub/sub for event-based interaction. ADD Options used: **LMD** (Pass Distributed or Hypermedia Links in the Payload to Represent Domain Model Links & Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Event State Transition Operations, based on Domain Operations & CRUD-based operations, & Expose Publish/Subscribe to Clients, some exposed to API by aggregating bounded contexts, some not), **RSD** (CQRS on aggregating endpoints, event-based operations). `https://github.com/dotnet-architecture/eShopOnContainers`
ET	domain: 8 api: 15	Customers and Orders System; implements transaction with the SAGA pattern and implements queries using CQRS. ADD Options used: **LMD** (N/A), **ODD** (Event State Transition Operations, based on CRUD-based operations, all exposed to API by aggregates), **RSD** (CQRS on aggregating endpoints, event-based operations). `https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders`
KB	domain: 11 api: 34	Kanban Board System; a multi-user collaborative application using event-sourcing and pub/-sub. ADD Options used: **LMD** (Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Event State Transition Operations, based on CRUD-based operations, all exposed to API by bounded context), **RSD** (CQRS on aggregating endpoints, no details on operations). `https://github.com/eventuate-examples/es-kanban-board`
NC	domain: 8 api: 72	Disease Statistics App; a public API providing a wide range of virus information. ADD Options used: **LMD** (Pass Embedded Data in the Payload, clients usually need the data, but not needed on some operations), **ODD** (Design Operations Like Primitive Datastore Operations, exposed to API by Entities), **RSD** (N/A). `https://github.com/disease-sh/API`
PC	domain: 63 api: 215	Pokemon App; a RESTful API for infos on Pokemon game series. ADD Options used: **LMD** (Pass Distributed or Hypermedia Links to Represent Domain Model Links, but in many cases it seems clients usually need the data immediately), **ODD** (Design Operations Like Primitive Datastore Operations, exposed to API by Entities), **RSD** (N/A). `https://github.com/PokeAPI/pokeapi`
RW	domain: 6 api: 67	Realworld Example App; provides API specs that support technology stack diversity. ADD Options used: **LMD** (Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Design Operations Like Primitive Datastore Operations & Group Operations on Resource and Select Operations in the Payload, all exposed to API by aggregating bounded contexts), **RSD** (N/A). `https://github.com/gothinkster/realworld`
LS	domain:170 api: 92	Lakeside Mutual System; demonstrates DDD in microservices of an insurance product. ADD Options used: **LMD** (Use Pass Distributed or Hypermedia Links on frontend to connect to backends which also Pass Embedded Data in the Payload, where clients usually need the data), **ODD** (Explicit Domain Operations & Design Operations Like Primitive Datastore Operations, all exposed to API by aggregating bounded contexts), **RSD** (N/A). `https://github.com/Microservice-API-Patterns/LakesideMutual`

shown in Listing 1, receives a Basket ID as a string which is modeled as an *ObjectIDBasedLink* and provides *CustomerBasket* as *EmbeddedData*. As for LMD the ≪*provided*≫ data elements are mainly needed to determine which decision option is chosen, we can conclude here that *GetBasketByIdAsync* represents passing embedded data in the messages. When inspecting the code closer, we can further find out that clients likely need the data provided immediately.

Secondly, for the ODD decision, we studied the *API Operation* and how its endpoint is exposed. For instance, for the *CheckoutAsync* operation shown in Listing 2, it is a domain operation which is used to perform an event-based state transition. Thus, the two respective stereotypes ≪*Event State Transition Operation*≫ and ≪*Domain Operation Based API Operation*≫ are used on that operation in the model. Moreover, the operations are offered on an endpoint that is exposed by an aggregating domain model element (a Bounded Context), which are recommended practices for ODD (see Section 2). Lastly, for the RSD decision, the system documentation reveals that this endpoint is intended as a ≪*CQRS*≫ type endpoint. Further, we can see that this endpoint exposes event-based operations and already determined that it is exposed by a Bounded Context (i.e., is aggregating), which are recommended practices for RSD (see Section 2).

```
[Route("checkout")]
[HttpPost]
[ProducesResponseType((int)HttpStatusCode.Accepted)]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
public async Task<ActionResult> CheckoutAsync(
  [FromBody] BasketCheckout basketCheckout,
  [FromHeader(Name = "x-requestid")] string requestId)
{
  var userId = _identityService.GetUserIdentity();

  basketCheckout.RequestId =
    (Guid.TryParse(requestId, out Guid guid) && guid != Guid.Empty) ?
  guid : basketCheckout.RequestId;

  var basket = await _repository.GetBasketAsync(userId);

  if (basket == null)
  {
    return BadRequest();}

  var userName =
    this.HttpContext.User.FindFirst(x => x.Type == ClaimTypes.Name).Value;

  var eventMessage = new UserCheckoutAcceptedIntegrationEvent(
    userId, userName, basketCheckout.City, basketCheckout.Street,
    basketCheckout.State, basketCheckout.Country, basketCheckout.ZipCode,
    basketCheckout.CardNumber, basketCheckout.CardHolderName,
    basketCheckout.CardExpiration, basketCheckout.CardSecurityNumber,
    basketCheckout.CardTypeId, basketCheckout.Buyer,
    basketCheckout.RequestId, basket);

  // Once basket is checkout, sends an integration event to
  // ordering.api to convert basket to order and proceeds with
  // order creation process
  try
  {
    _eventBus.Publish(eventMessage);}
  catch (Exception ex)
  {
    _logger.LogError(ex,
      "ERROR Publishing integration event: {IntegrationEventId} from {AppName}",
      eventMessage.Id, Program.AppName);
    throw;}

  return Accepted();
}
```

Listing 2: CheckoutAsync Operation from Basket API

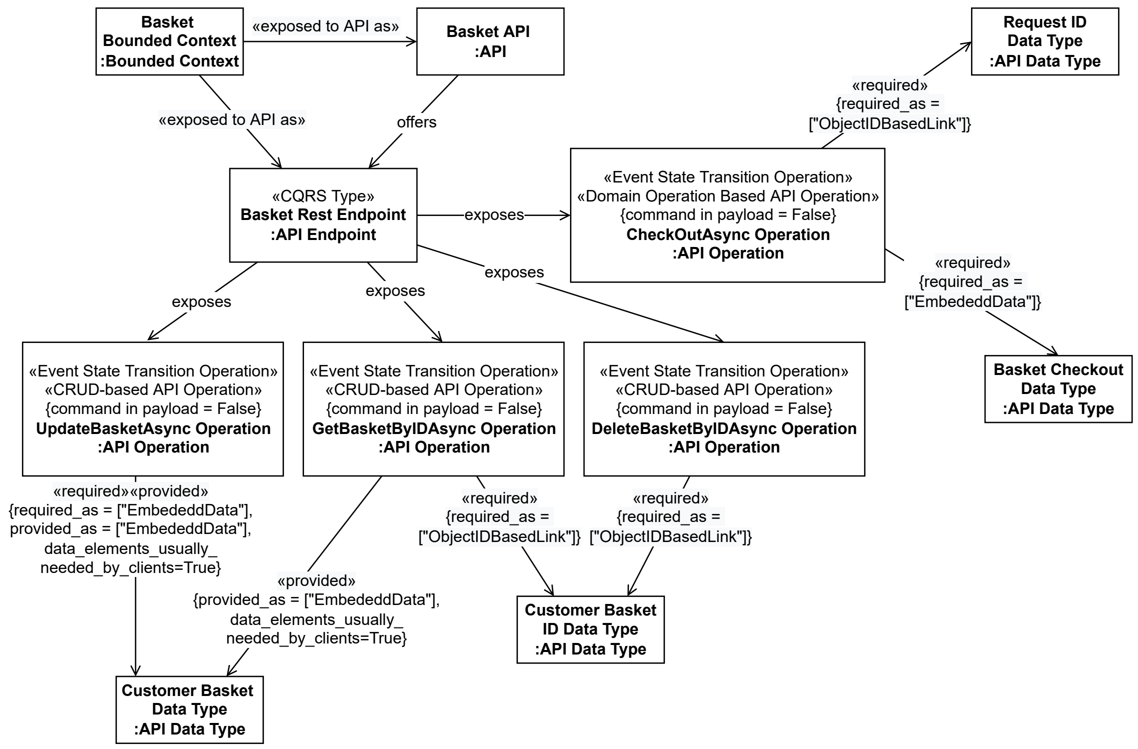
Basket Bounded Context :Bounded Context
«exposed to API as»
Basket API :API
Request ID Data Type :API Data Type
«required» {required_as = ["ObjectIDBasedLink"]}
«exposed to API as»
offers
«CQRS Type» Basket Rest Endpoint :API Endpoint
exposes
«Event State Transition Operation» «Domain Operation Based API Operation» {command in payload = False} CheckOutAsync Operation :API Operation
«required» {required_as = ["EmbededdData"]}
Basket Checkout Data Type :API Data Type
exposes
exposes
exposes
«Event State Transition Operation» «CRUD-based API Operation» {command in payload = False} UpdateBasketAsync Operation :API Operation
«Event State Transition Operation» «CRUD-based API Operation» {command in payload = False} GetBasketByIDAsync Operation :API Operation
«Event State Transition Operation» «CRUD-based API Operation» {command in payload = False} DeleteBasketByIDAsync Operation :API Operation
«required»«provided» {required_as = ["EmbededdData"], provided_as = ["EmbededdData"], data_elements_usually_ needed_by_clients=True}
«provided» {provided_as = ["EmbededdData"], data_elements_usually_ needed_by_clients=True}
«required» {required_as = ["ObjectIDBasedLink"]}
«required» {required_as = ["ObjectIDBasedLink"]}
Customer Basket ID Data Type :API Data Type
Customer Basket Data Type :API Data Type

Figure 7: ES Model Excerpt: Basket RESTful API and its Mapping to a Bounded Context

4.1. Derivation of Scoring Scheme

We have established the scoring scheme to assess the ADDs in 2. This scoring scheme is based on our prior empirical study [11] by considering the various impacts on decision drivers reported by practitioners. Grounded on this empirical evidence, we decided to provide an assessment on an ordinal scale [++: very positive, + : positive, o : neutral, -: negative, - -: very negative]. The ordinary scale could turn qualitative judgments in the practitioner texts to numerical assessment.

For example, consider the LMD decision, as discussed in [11]. According to the empirical evidence in the practitioner sources, if domain links are identified to be needed by clients, the option *Do not offer the Link via the API* shall not be used. It shall only be used for those links in the domain model, which clients do not require. If this derivation from the domain model is performed correctly, the most positive impacts on decision drivers are achieved, if *Embed Linked Data in the Payload* is used for the required linked data elements, and *Use Distributed or Hypermedia Links in the Payload* in all other cases. Embedding means to minimize the number of messages that need to be exchanged, leading to positive impacts on *Minimize API Calls*, *Performance*, *Scalability*, and *Protocol Complexity* decision drivers. However, *Message Sizes* are lower with distributed links, and as they are always up-to-date, also *API Evolvability* and *API Modifiability* are positively influenced.

A still good, but less optimal variant is to use *Pass Object Identifiers in the Payload* in those cases where distributed links are applicable. In direct comparison, it has the disadvantages of possibly *Exposing Domain Model Details in API* as well as higher *Coupling of Clients to Server*. If embedded data is not used, where clients usually require most of the linked data elements immediately, consequently a negative impact on some of the forces can be expected. Likewise, if embedded data is used, but clients do not require most of the linked data elements immediately, some of the forces are influenced negatively, too.

These considerations have led to a literal derivation of the scoring scheme of the Decision LMD (see next section). For the other decisions we have derived the scoring schemes in the same way based on the empirical results from [11].

4.1.1. Scoring Scheme for Link Mapping Decision (LMD)

- IF FOR SOME domain links which are needed by the clients: *Do not offer the Link via the API* THEN assessment = (- -).
- IF (FOR ALL domain links *dl* which are needed by the clients: (IF for *dl* clients usually require most of the linked data elements immediately: THEN *Embed Linked Data in the Payload* is used for the required linked data elements, ELSE *Use Distributed or Hypermedia Links in the Payload* is used)) THEN assessment = (++).
- IF (FOR ALL domain links *dl* which are needed by the clients: (IF For *dl* clients usually require most of the linked data elements immediately: THEN *Embed Linked Data in the Payload* is used for the required linked data elements, ELSE *Use Distributed or Hypermedia Links in the Payload* OR *Pass Object Identifiers in the Payload* is used)) THEN assessment = (+).
- IF (FOR SOME domain links *dl* which are needed by the clients: If for *dl* clients usually require most of the linked data elements immediately: *Embed Linked Data in the Payload* is NOT used for *dl*) THEN assessment = (-)
- IF any other combination is used (e.g. for some domain links *dl* which are needed by the clients, and for *dl* clients usually do NOT require most of the linked data elements immediately AND *Embed Linked Data in the Payload* is used for *dl*) THEN assessment = (o).

4.1.2. Scoring Scheme for Operation Design Decision (ODD)

Please note that a model can contain more than one API. In some of the following decision points different choices are possible for the operation of each of the APIs in one model. We use "*Has Aggregating Endpoint*" to denote that an API endpoint is exposed by a domain element of aggregating nature, such as an Aggregate (or its Aggregate root), a Bounded Context, or a Domain Service. For example, Entities and Value Objects are domain model elements that are usually not of aggregating nature.

- IF FOR EACH API: (FOR ALL API Operations: (*Expose Domain Events as State Transitions* OR *Expose Domain Events via Feeds or Pub/Sub*) AND *Has Aggregating Endpoint* is applied) OR ((FOR ALL API Operations: *Domain Operations on Resources* OR *Encode Operations as Commands in the Payload*) AND *Has Aggregating Endpoint* is applied) THEN assessment = (++).
- IF FOR EACH API: (FOR ALL API Operations: *Expose Domain Events as State Transitions* OR *Expose Domain Events via Feeds or Pub/Sub* AND *Has Aggregating Endpoint* is applied) OR ((FOR ALL API Operations: *Domain Operations on Resources* OR *Encode Operations as Commands in the Payload* OR *CRUD-Style Operations on Resources*) AND *Has Aggregating Endpoint* is applied) THEN assessment = (+).
- IF FOR ALL API Operations OF ALL APIs: (*Domain Operations on Resources* OR *Encode Operations as Commands in the Payload* OR *Expose Domain Events as State Transitions* OR *Expose Domain Events via Feeds or Pub/Sub* OR *CRUD-Style Operations on Resources*) AND *Has Aggregating Endpoint* is applied. THEN assessment = (o).
- IF FOR ALL API Operations OF ALL APIs: *CRUD-Style Operations on Resources* is applied and not *Has Aggregating Endpoint.* THEN assessment = (- -).
- IF any other combination is used, e.g. some API endpoints are not aggregating endpoints THEN assessment = (-).

4.1.3. Scoring Scheme for Resource Segregation Decision (RSD)

Please note that the decision points use *Is Aggregating Endpoint* in the same way as explained for the ODD scoring scheme, just not operating on operations, but on endpoints.

- IF FOR ALL API Endpoints e that are of *CQRS type*: At least 1 API Operation is exposed by e AND all API Operations exposed by e are *Event-based Pub/Sub Operations* or *Event Transition Operations* AND e *Is Aggregating Endpoint* THEN assessment = (++).
- IF FOR ALL Endpoints e that are of *CQRS type*: At least 1 API Operation is exposed by e AND all API Operations exposed by e are *Event-based Pub/Sub Operations* or *Event Transition Operations* AND e *Is or Is Not Aggregating Endpoint* THEN assessment = (+).

- IF FOR ALL Endpoints *e* that are of *CQRS type*: All API Operations exposed by *e* are or are not *Event-based Pub/Sub Operations* or *Event Transition Operations* AND *e Is Aggregating Endpoint* THEN assessment = (o).
- IF FOR ALL Endpoints *e* that are a *CQRS type*: At least 1 API Operation is exposed by *e* AND All API Operations exposed by *e* are not *Event-based Pub-/Sub Operations* or *Event Transition Operations* AND *e Is Not Aggregating Endpoint* THEN assessment = (- -).
- IF any other combination is used for API Endpoints *e* that are of *CQRS type*, e.g. only some API operations are event-based THEN assessment = (-).

4.2. Ground Truth Assessment

To generate a ground truth for later evaluation, we have assessed each of the cases in our multi-case study manually based on our scoring scheme from Section 4.1. The results are presented in Table 5.

To illustrate the assessment using an example, Table 4 summarizes the assessment for the ES case study shown in Figure 7. Firstly, we summarize the options present in the case for each of the three ADDs, then we show the scoring scheme condition that applies, and finally, the ordinal score derived.

Please note that this is not the result of the automated detectors presented below but the manually derived ground truth which we used to compare later to our detectors' results. Some assessments are not applicable ("n/a") as this aspect is not implemented or explained in the case's source, meaning that we cannot judge the ADD based on this case's model. Table 5 contains the assessment of the 12 cases using our ordinal scoring scale (summaries of the ADD inspections can be found in Table 3). As can be seen, the case models cover for the decisions a wide range of possible combinations of the decision options.

5. Detectors for ADD Conformance Assessment

In this section, we describe details of our detector approach for automatically assessing conformance to the ADD options from Section 2. We propose a modular detector approach where one or more detectors are responsible for detecting each of the decision points in the scoring scheme from Section 4.1. Tables 6-8 gives an overview of all the detectors we defined for the three ADDs.

It also explains in the *Use in Assessment* column how each scoring scheme result of each of the three ADDs is derived during the assessment by the detectors. Please note that in many cases combinations of detectors are needed, and that they are applied in the given contexts of API elements that are passed in as lists (e.g. in ODD all API elements of each API are usually used, whereas RSD applies the detectors to the elements of CQRS type endpoints).

Table 4: ES Case Study: Assessment Overview

	LMD	**ODD**	**RSD**
Summary of chosen options (see Section 2)	Pass Distributed or Hypermedia Links in the Payload to Represent Domain Model Links & Pass Embedded Data in the Payload. Clients usually need the data.	Event State Transition Operations based on Domain Operations & CRUD-based Operations & Expose Publish/Subscribe to Clients. Some are exposed to the API by aggregating bounded contexts, some not.	CQRS is used on aggregating endpoints & event-based operations are used.
Applying condition from the scoring scheme (see Section 4.1)	IF any other combination is used (e.g. for some domain links *dl* which are needed by the clients, and for *dl* clients usually do NOT require most of the linked data elements immediately AND *Embed Linked Data in the Payload* is used for *dl*)	IF any other combination is used, e.g. some API endpoints are not aggregating endpoints	IF FOR ALL API Endpoints *e* that are of *CQRS type*: At least 1 API Operation is exposed by *e* AND all API Operations exposed by *e* are *Event-based Pub/Sub Operations* or *Event Transition Operations* AND *e Is Aggregating Endpoint*
Ordinal Score Derived	o	-	++
Meaning of the assessment result	This result indicates that the design of the API reflects the links present in the domain model in an acceptable fashion but improvements can be achieved. In particular, here some operations use embedded data but they convey possibly a lot of information that might not always be needed by clients. API developers can use this information to redesign the operations to provide only the required information or use distributed/hypermedia links instead. Note: This is one of the few cases where model-based and human assessment diverge (as explained in Section 6). Thus, the model, including the excerpt in Figure 7 seems to be optimal, but a deeper source code analysis revealed discrepancies. Here, changing the *data_elements_usually_needed_by_clients* flag manually to *False* for the affected operations is necessary to enable an automatic assessment of "o" too (see Section 6).	This result indicates that the design of the API reflects the domain operations in a suboptimal fashion. As can be seen in Figure 7, some operations are offered as CRUD-based API Operations, but unlike Figure 7 not everywhere an aggregating endpoint is used. Also, event-state transition operations and domain operation-based API operations are used. API developers can use this information to redesign the operations e.g. based on a common event model, turning endpoints into aggregating endpoints, or exposing operations as domain operation-based API operations.	This result indicates that the design of the API in relation to the domain model is optimal with regard to resource segregation. Three endpoints are offered as CQRS endpoints (one of them in shown in Figure 7) and all API operations offered by those endpoints are offering event transition operations. API developers can learn that no changes for conformance to this ADD are required.

Table 5: Ground Truth: Manual Assessment Results

ID	LMD	ODD	RSD	**ID**	LMD	ODD	RSD
AX	n/a	n/a	++	ET	n/a	++	++
PM	++	+	n/a	KB	++	++	++
BA	+	-	++	NC	o	- -	n/a
OS	+	+	n/a	PC	-	- -	n/a
CM	o	+	o	RW	++	+	n/a
ES	o	-	++	LS	++	+	n/a

Table 6: Detectors Overview: LMD

ID	Detectors	Description	Use in Assessment
d1	detect_each_API_endpoint_has _links_to_API_operations (*input: List⟨APIElements⟩*)	To detect whether each API endpoint has links to API operations. Helps to find those exposed to clients.	The detector is required (with results SUCCESS, PARTIAL SUCCESS) in the assessment.
d2	detect_api_operations_with _embedded_data_elements_usually _needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide embedded data as data elements usually needed by clients.	For '++' assessment, this detector or d5 only one of them can be successful (i.e. SUCCESS or PARTIAL_SUCCESS). If the d5 is successful, this detector must fail in '++' assessment. For '-' assessment, this detector must be FAILED when d3 or d4 is partialy successful or successful.
d3	detect_api_operations_with _distributed_links_to_data_elements _usually_needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide distributed links to data elements usually needed by clients.	For '++' assessment, this detector must be failed but it can be partial successful in '+' assessment.
d4	detect_api_operations_with_object _id_based_links_to_data_elements _usually_needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide object-ID based links to data elements usually needed by clients.	For '++' assessment, this detector must be failed but it can be partial successful in '+' assessment.
d5	detect_api_operations_with _distributed_links_to_data_elements _usually_not_needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide distributed links to data elements usually not needed by clients.	For '++' assessment, this detector or d2 only one of them can be successful (i.e. SUCCESS or PARTIAL_SUCCESS). If the d2 is successful, this detector must fail in '++' assessment
d6	detect_api_operations_with_object _id_based_links_to_data_elements _usually_not_needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide object-ID based links to data elements usually not needed by clients.	For '++' assessment, this detector must be failed but it can be successful or partial successful in '+' assessment.
d7	detect_api_operations_with _embedded_data_elements_usually _not_needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide embedded data as data elements usually not needed by clients.	For '++' assessment, this detector must be failed . For '+' assessment, this detector should not be successful nor partial sucessful, unless the assessment result will be 'o' instead.
Note: The detectors-based assessment for each decision will result in "n/a" if at least one of decision's detectors has the result NOT_APPLICABLE.			

Table 7: Detectors Overview: ODD

ID	Detectors	Description	Use in Assessment
d8	detect_each_crud_based_api _operation (*input: List⟨APIElements⟩*)	To detect each CRUD-based API operation on API endpoints.	It can be successful or partially successful for APIs for the '++' ,'+', 'o', or '-' assessments with at least one other detectors of d9-d12 used (partially) successfully with it. If this is the only detector that is successful, the assessment will turn to '--'.
d9	detect_each_domain_operation_based _api_operation (*input: List⟨APIElements⟩*)	To detect each domain operation-based API operation on API endpoints.	It can be successful or partially successful for APIs for the '++', '+', 'o', or '-' assessments.
d10	detect_each_api_operation_encoded _as_commands_in_the_payload (*input: List⟨APIElements⟩*)	To detect each API operation on API endpoints that has the operation encoded as commands in the payload.	It can be successful or partially successful for APIs for the '++', '+', 'o', or '-' assessments.
d11	detect_each_api_operation_with _event_based_state_transition (*input: List⟨APIElements⟩*)	To detect each API operation on API endpoints that represents an event-based state transition.	It can be successful or partially successful for APIs for the '++', '+', 'o', or '-' assessments.
d12	detect_each_api_operation_with _event_based_pub_sub (*input: List⟨APIElements⟩*)	To detect each API operation on API endpoints that offers events via feeds or pub/subs.	It can be successful or partially successful for APIs for the '++', '+', 'o', or '-' assessments.
d13	detect_aggregating_endpoint_and _crud_based_operation (*input: List⟨APIElements⟩*)	To ensure that API endpoints with CRUD-based API operations are aggregating endpoints.	This detector is used alongside d8, verifying the investigated endpoints are aggregating ones with CRUD-based API operations. When d8 is used together with other detectors (d9-d12), this detector should be a SUCCESS to reach '++', and it should not be FAILED to reach '+'. If it has FAILED and only d8 succeeded, the assessment will be '--'. If it and d8 are a PARTIAL_SUCCESS, the assessment will be '-'.
d14	detect_aggregating_endpoint_and _domain_based_operation (*input: List⟨APIElements⟩*)	To ensure the investigated API endpoints with domain-based API operations are aggregating endpoints.	This detector is used alongside d9 and d10, verifying that the investigated endpoints are aggregating ones with domain-based API operations. When d9 or d10 are used together with it, it should be a SUCCESS to reach '++'. It should not be FAILED to reach '+'. If it is PARTIAL_SUCCESS, the assessment will be '-'.
d15	detect_aggregating_endpoint_and _event_based_operation (*input: List⟨APIElements⟩*)	To ensure the investigated API endpoints with event-based API operations are aggregating endpoints.	This detector is used alongside d11 and d12, verifying that the investigated endpoints are aggregating ones with event-based API operations. When d11 or d12 are used together with it, it should be a SUCCESS to reach '++'. Itt should not be FAILED to reach '+' . If it is PARTIAL_SUCCESS, the assessment will be '-'.
Note: The detectors-based assessment for each decision will result in "n/a" if at least one of decision's detectors has the result NOT_APPLICABLE.			

Table 8: Detectors Overview: RSD

ID	Detectors	Description	Use in Assessment
d16	detect_api_endpoints_exposed _by_aggregating_domain_model _element (*input: List⟨APIElements⟩*)	To detect API endpoints exposed by aggregating domain elements (such as Aggregate, Aggregate root, Service, Bounded Context).	For '++' and 'o', it must succeed (SUCCESS, PARTIAL_SUCCESS) for a given endpoint, and for '--' it must fail for the endpoints under investigation.
d17	detect_each_cqrs_endpoint_with _event_based_operations (*input: List⟨APIElements⟩*)	To detect API endpoints offer CQRS Queries or Commands, as well as event-based API operations.	It must be SUCCESS, PARTIAL_SUCCESS, or FAILED for the decision to be applicable.
Note: The detectors-based assessment for each decision will result in "n/a" if at least one of decision's detectors has the result NOT_APPLICABLE.			

All detectors are implemented in Python and work mainly by traversing models. All models are implemented with CodeableModels[4], a Python tool for precise specification of meta-models, models, and model instances in code. Based on the meta-models and decision models, we manually created model instances for each case system and implemented automated PlantUML code generators to produce graphical visualizations of all model instances.

To illustrate the approach further let us explain one exemplary detector in detail as pseudo code. Algorithm 3 shows the detector to detect each API operation in a list of API elements *AE* that is of event state transition type. In this algorithm we first get the API operations from the *AE* list. If no operations are defined, the detector is *NOT_APPLICABLE*. If this is not the case, we next detect the possible violations by traversing all API operations and checking whether they conform to the respective stereotype ≪*Event State Transition Operation*≫. If there are only members of this stereotype, we report a *SUCCESS*; if there are violations and members of this stereotype, we report a *PARTIAL_SUCCESS*; else a *FAILURE* is reported.

Please note that we also return the list of *violations*, which helps humans to later inspect the violation, and maybe correct either the model or the detection. The other detectors in Tables 6-8 are constructed in a similar fashion, inspecting and detecting other aspects in the model, as explained in the *Description* column of Tables 6-8.

```
input: List ⟨Api_Element⟩ $AE$
output: Tuple⟨Detector_Results_Enum, Set, String⟩
begin
  $API\_OPS \leftarrow$ api_operations($AE$)
  if ($API\_OPS = \emptyset$):
    return (NOT_APPLICABLE, $\emptyset$,
            'No API operations defined')
  $violations \leftarrow \emptyset$
  $members \leftarrow \emptyset$
```

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

```
  for OP in API_OPS:
    if (OP.is_of_stereotype(
         ≪Event State Transition Operation≫):
      $members \leftarrow members \cup OP$
    else:
      $violations \leftarrow violations \cup OP$
  if ($violations \neq \emptyset$):
    if ($members \neq \emptyset$):
      return (PARTIAL_SUCCESS, violations,
      'Some operations are Domain Events ' +
      'as State Transitions')
    else:
      return (FAILURE, violations,
         'No operation is Domain Events ' +
         'as State Transitions')
  return (SUCCESS, $\emptyset$, 'All operations are Domain ' +
          'Events as State Transitions')
end
```

Algorithm 1: Detector d12 — detect_each_api_operation_with_event_based_state_transition

6. Multi-Case Study Results

In this section, we discuss the results of our multi-case study. We first analyze the results case by case for each decision option. Here, we simply count the correctly identified results and discuss interesting insights. Next, we analyze our data set statistically and show (1) that there is no significant difference between ground truth and detector result data and (2) that the effect size between the two variables is negligible.

6.1. Detailed Discussion of the Results

Table 9 summarizes the results of the automated assessments based on our detectors approach for the 12 models in our multi-case study. Comparing the overall results in Table 9 to our ground truth in Table 5, we can calculate matching score ratios for each of the decisions: LMD – 8/12 (67%), ODD – 12/12 (100%), RSD – 10/12 (83%). In total, 83% of the decision points in our multi-case study have been correctly identified by the automated detectors.

The places where ground truth and detector results are different are all cases where additional human judgment is needed. When we apply the detectors, we do not only get the results, but in case of violations, we also get the violation set as part of the detector result. These violations can help to spot the aspects humans need to inspect more closely.

Let us discuss a few notable cases where detector results diverge or could have diverged in detail. The *RSD* detector delivers "o" for the *AX* and *KB* models. This deviates from our manual assessment, where we judged those models as "++". The reason for this is that the model is detailed enough to see that all CQRS endpoints are aggregating endpoints, but is missing detailed modeling of operations. Thus it cannot automatically be judged whether event-based operations are used. However,

Table 9: Automated Assessment Results for Each Model from the Detectors

ID	**LMD**	**Reasons: Detector Results**	**ODD**	**Reasons: Detector Results**	**RSD**	**Reasons: Detector Results**
AX	n/a	*Not Applicable*: d1, d2, d3, d4, d5, d6, d7	n/a	*Failed Detectors*: d13, d14, d15 *Not Applicable*: d8, d9, d10, d11, d12	o	*Successful Detectors*: d16 *Failed Detectors*: d17
PM	++	*Successful Detectors*: d1, d2 *Failed Detectors*: d3, d4, d5, d6, d7	+	*Successful Detectors*: d13, d14 *Partially Successful Detectors*: d8, d9 *Failed Detectors*: d10, d11, d12, d15	n/a	*Successful Detectors*: d16 *Not Applicable*: d17
BA	+	*Successful Detectors*: d1 *Partially Successful Detectors*: d2 *Failed Detectors*: d3, d4, d5, d6, d7	-	*Successful Detectors*: d11 *Partially Successful Detectors*: d8, d9, d10, d13, d14, d15 *Failed Detectors*: d12	++	*Successful Detectors*: d17 *Partially Successful Detectors*: d16
OS	+	*Successful Detectors*: d1 *Partially Successful Detectors*: d2, d6 *Failed Detectors*: d3, d4, d5, d7	+	*Successful Detectors*: d8 *Partially Successful Detectors*: d13 *Failed Detectors*: d9, d10, d11, d12, d14, d15	n/a	*Partially Successful Detectors*: d16 *Not Applicable*: d17
CM	++	*Partially Successful Detectors*: d1, d2 *Failed Detectors*: d3, d4, d5, d6, d7	+	*Partially Successful Detectors*: d8, d9, d13, d14 *Failed Detectors*: d10, d11, d12, d15	o	*Successful Detectors*: d16 *Failed Detectors*: d17
ES	++	*Partially Successful Detectors*: d1, d2, d5 *Failed Detectors*: d3, d4, d6, d7	-	*Partially Successful Detectors*: d8, d9, d11, d12, d13, d14, d15 *Failed Detectors*: d10	++	*Successful Detectors*: d17 *Partially Successful Detectors*: d16
ET	n/a	*Successful Detectors*: d1 *Not Applicable*: d2, d3, d4, d5, d6, d7	++	*Successful Detectors*: d8, d11, d13, d15 *Failed Detectors*: d9, d10, d12, d14	++	*Successful Detectors*: d16, d17
KB	++	*Successful Detectors*: d2 *Partially Successful Detectors*: d1 *Failed Detectors*: d3, d4, d5, d6, d7	++	*Successful Detectors*: d8, d11, d13, d15 *Failed Detectors*: d9, d10, d12, d14	o	*Partially Successful Detectors*: d16 *Failed Detectors*: d17
NC	++	*Successful Detectors*: d1, d2 *Failed Detectors*: d3, d4, d5, d6, d7	- -	*Successful Detectors*: d8 *Failed Detectors*: d9, d10, d11, d12, d13, d14, d15	n/a	*Failed Detectors*: d16 *Not Applicable*: d17
PC	++	*Successful Detectors*: d1, d5 *Failed Detectors*: d2, d3, d4, d6, d7	- -	*Successful Detectors*: d8 *Failed Detectors*: d9, d10, d11, d12, d13, d14, d15	n/a	*Failed Detectors*: d16 *Not Applicable*: d17
RW	++	*Successful Detectors*: d1, d2 *Failed Detectors*: d3, d4, d5, d6, d7	+	*Successful Detectors*: d8, d13 *Failed Detectors*: d9, d10, d11, d12, d14, d15	n/a	*Successful Detectors*: d16 *Not Applicable*: d17
LS	++	*Partially Successful Detectors*: d1, d2 *Failed Detectors*: d3, d4, d5, d6, d7	+	*Partially Successful Detectors*: d8, d9, d13, d14 *Failed Detectors*: d10, d11, d12, d15	n/a	*Partially Successful Detectors*: d16 *Not Applicable*: d17

as in both cases of *AX* and *KB* models, model and system documentation contain information on the event-based endpoints (the naming of endpoints), and there is a description that event-based communication is used in the documentation, too, we can make a manual conclusion here. As the detailed operation modeling is missing, the detectors cannot decide for any of the other options and thus uses the fallback "o". This could be mitigated by more detailed modeling of operations or introducing a flag that signals the event-driven nature of the system to the detectors.

The *LMD* detector for the *CM* model delivers "++" which deviates from the "o" in our manual assessment. We inspected the operations carefully and found one that uses embedded data, but seems to convey possibly a lot of information that might not always be needed by clients. If this interpretation is correct, "o" is the correct assessment, but this can only be found out by source code analysis, not on the model. This issue could be mitigated by changing the *data_elements_usually_needed_by_clients* flag manually to *False*. The same case happens for the *ES* model, also with just one operation. It also occurs for the *NC* model, where this violation is even worse, as a couple of operations are candidate for using embedded data, but it seems the operations convey a lot of information that is not always needed by clients.

The *LMD* detector for the *PC* model reports "++" which deviates from the "-" in our manual assessment. We inspected the operations carefully and that many of the operations providing URL-based links actually contain data immediately needed by clients. If this interpretation is correct, "-" is the correct assessment, but this can only be found out by source code analysis, not on the model. This issue could be mitigated by changing the *data_elements_usually_needed_by_clients* flag manually to *True*.

The *LS* model is another interesting case. The system uses frontend endpoints that deliver URLs to access the backend services, which then deliver the actual data in embedded form. If we would model just based on the frontends, *LMD* would be falsely detected as in the *PC* model, explained in the previous paragraph. This technical feature could also have led to the misinterpretation in ODD (and RSD) that the frontends do not belong to the aggregating bounded contexts; which both would have led to yet another deviation in manual and automatic assessment. As we, however, modeled the *LS* model considering the relation to backend services, manual and automatic assessment match.

The *ODD* cases of the *BA* and *ES* models are assessed as "-", consistently in ground truth and automatic assessment. But with a few changes those could turn into "++", as just a few endpoints are not covered in domain model. If corresponding aggregating domain model elements were modeled, the asssessments could easily be improved to the best score. Here, the failed detectors clearly pinpoint the few endpoints that require attention.

The "--" assessment of LMD did not appear in our cases. We did not implement a detector for it yet, as this detector would be pretty trivial. Here, human judgment if a domain link is needed by clients but not in the API is needed. This cannot be found out automatically. Thus a simple flag to indicate this fact on domain links in our models, which can be set by a human designer, together with a simple detector whether there is an API link corresponding to the domain link would be enough to support this assessment, too.

The *ODD* cases of the *BA* and *ES* models are assessed as "-", consistently in ground truth and automatic assessment. But with a few changes those could turn into "++", as just a few endpoints are not covered in domain model. If corresponding aggregating domain model elements were modeled, the asssessments could easily be improved to the best score. Here, the failed detectors clearly pinpoint the few endpoints that require attention.

In summary, our multi-case study have shown that all ADD options could possibly automatically be detected based on our scoring scheme (**RQ1**). Regarding **RQ2**, 83% of the decision points have been correctly identified, and the remaining cases are those where more detailed modeling of aspects beyond architecture-level models, e.g. with a simple additional flag, could have solved the issue. That is, a single manual inspection and correction could lead to correct detection in future runs, e.g. in a continuous delivery pipeline context.

6.2. Statistical Analysis of the Results

To confirm the results for **RQ2** and get a more precise estimate for the effect size, we statistically analyzed the results in R. In our data set, we had to deal with ordinal variables, a rather small sample size, and data that is not normally distributed. We first confirmed the non-normal distribution with a Shapiro-Wilk test [34] using R's *shapiro.test()* function. The data in Table 10 shows that, as the p-values for both ground truth and detector results data are significant, we must reject the null hypothesis that the data is normally distributed for both variables. Thus, the t-test, which assumes the normal distribution, is not applicable.

In general, for our problem the Wilcoxon signed-rank test would be applicable, but as many data points are identical, we get many zero values, which are in Wilcoxon's method removed from the test, making the results non-exact for our data set. The Wilcoxon signed rank test with Pratt method can handle those zero values [35], which means that it is more appropriate for our data set. For Wilcoxon-Pratt Signed-Rank Test calculation, we used the *wilcoxsign_test()* function from R's *coin* package. The results of the test in Table 10 are not significant, meaning that the null hypothesis cannot be rejected. Thus, we must assume the true μ is close to 0.

To confirm this result and assess the relevance of the result further, we computed the effect size. Here, Kitchenham et al. [37] suggest Cliff's δ [36] as a robust method for empirical software engineering. For this, we used the *cliff.delta()* function from R's *effsize* package. As shown in Table 10, the delta estimate is -0.1042524, which is interpreted as: the effect size is negligible [38].

Table 10: Statistical Analysis Results

Shapiro-Wilk Test for Ground Truth Data	
p-value	0.000448
Shapiro-Wilk Test for Detector Results Data	
p-value	2.728e-05
Wilcoxon-Pratt Signed-Rank Test	
p-value	0.369
Cliff's Delta	
delta estimate	-0.1042524 (negligible)

6.3. Lessons Learned

In this section, we summarize the major lessons learned of our study, especially from the multi-case study that illustrate the relevance of our results.

Lesson 1. *A rough estimation of manual architectural conformance checking strongly indicates that automation is required, if a rapid release schedule is used.*

As discussed below in Section 8, each of the two authors has performed a complete manual conformance assessment for our case study model set at least three times during the writing of this article, and before that numerous times for all single models during the development of our approach. Thus, we can provide a rough estimate of a manual conformance checking effort.

An initial manual checking takes in our experience substantially more time than subsequent re-assessments. While some links require initially a few minutes to check, with experience and practice one can reach about 10-30 seconds per link between domain model and API element on average. That is, our smaller models initially took about 10-20 minutes to check, going down to a few minutes to check in later iterations. The larger models required about 40-60 minutes initially, and went down to 8-15 minutes in later iterations.

Assuming that a very small-size industrial system is about the size of our largest models, a small medium-sized industrial system is about the size of our complete model set, and large-scale systems (such as those in [13, 14, 15]) are way beyond our models' scale, it is immediately clear that even for small to medium real-life systems it is almost impossible to check them manually in each run of a Continuous Integration/Delivery (CI/CD) pipeline (e.g. for daily or hourly release). For large-scale systems, even a single manual check might be infeasible. Thus, a significant lesson learned is that manual conformance checking might be possible for release schedules measured in years, but for contemporary rapid release schedules automation is often the only feasible option.

Please note that this kind of automation for frequent releases might require automatic reconstruction of the model. In our approach, all aspects except the case study modeling were automated. We discuss in Section 3 how this step can be automated, too, using our existing static code analysis approach [33]. In fact, the evaluation [33] of our prior work demonstrates the automatic API reconstruction of our case study systems ES and LS used in this article, too (see Table 3).

Lesson 2. *Accurate automatic conformance checking is possible, especially if domain-specific system knowledge is utilized.*

In our approach we have shown that for a given modeling of 12 systems according to existing industry best practices (empirically studied in [11]) and using a given interpretation of these industry best practices (i.e., our scoring scheme), following our approach, it is straightforward to provide detectors that accurately detect conformance to the ADDs describing the industry best practices. Further our detailed assessment in Section 6 shows that with domain-specific knowledge about the systems (which an e.g. industry team working on a system has), the detection can be improved up to a 100% accuracy.

Lesson 3. *The approach can be adapted or calibrated if the detector results provide detailed traceability.*

The results in our evaluation use concrete modeling or interpretation of the industry best practices (i.e. our scoring scheme), but do not depend on them. During our many iterations and refactorings, we have many times improved the models, scoring scheme, and detectors, and still about the same level of accuracy of the detectors was possible to achieve. A consequence of this observation is that once one is acquainted with the modeling approach, it is possible to adapt models, change the detectors, or calibrate the scoring scheme.

All steps in our approach are fully automated and can be re-run after such a change. The major reason why this works in our experience is the provided traceability in detector results—i.e. status (Success, Failed, or Partial Success), the affected modeling elements (violations), and a reason for the outcome. That is, if our detectors fail to produce the expected results, the traceability information provided by the detectors helps to pinpoint the model elements or detector implementations that failed. This is for instance important for API developers to benefit from an assessment, as e.g. explained in Table 4, for instance, while realizing fixes or refactorings of the API. In a large-scale model, only obtaining a score like "o" or "-" might not be helpful because it is not clear which parts of the model caused the non-optimal results. But with the detailed traceability links, each root cause of a failure can be easily pinpointed and then fixed or refactored.

Lesson 4. *Once the detector infrastructure is realized, writing or changing detectors is straightforward and comparatively low effort.*

The effort for providing the automation of a single detector, once the general detector framework, modeling framework, model visualization, model traversal methods, traceability support, reporting framework, and other generic components are in place, is relatively low. Most single detectors have been written in 20-40 minutes time, with a couple of refactorings, requiring on average again as much time. That is, it is feasible to adapt detectors to new circumstances, if needed. Furthermore, as we provide our results as open access artifacts, even our implementation can be reused, thus reducing the initial required effort substantially.

Lesson 5. *For full automation of our approach, the API model must either be automatically extracted or API code must be generated from it.*

Even though a detector is often not difficult to implement and maintain, the initial creation and maintenance of the API models need to be considered, too (This does usually not apply to the DDD models as they are usually intended to be provided as manually modeled artifacts.) This often requires some level of architecture reconstruction, as not only the API calls need to be understood, but also calls to the backend microservices they interact with.

We have outlined two feasible approaches in Section 3 to deal with this challenge: Firstly, our extraction from the code approach described in [33] can be applied, which enables automatic reconstruction of APIs but requires an initial specification effort. This is applied to the API models of our case studies ES and LS (see [33]). Secondly, a model-driven approach to generate API code from a model can be chosen, as e.g. offered in MDSL [27] and used in our Case Study PM.

7. Related Work

In this section, we compare to the related work. We first discuss related works on API and microservice design, and show that we close the gap in the literature. Most of the existing works mainly provide informal guidance that is not yet supported in an automated fashion. Works that provide automation for conformance assessment are already being discussed, but they either only cover very general conformance assessment not suitable for API/DDD problems or address specific microservice patterns. We also include related works on checking local APIs. None exist for the problem of supporting the mapping of APIs and DDD yet. On the other hand, in the existing works on the relation of APIs and DDD, mostly broader microservice architecture aspects are covered, and API topics are rather a side-note. An automated mapping or conformance assessment approach does not exist in this area of research either.

7.1. Related Works on API and Microservice Design

Various works address the design of API endpoints, often in the form of patterns, which we consider in our decision options, or as decision models. The Microservice API patterns [4] contain patterns on operation and link/embedded data design [39, 2]. The Enterprise Integration patterns [40] contain various patterns on message construction, such as Command Message or Event Message, that are relevant in message-based endpoint design. The Service design patterns [41] contain high-level patterns on API styles, as well as client-service interactions, which all are relevant for service-based endpoint design.

Fowler [8] links patterns such as Domain Model to distributed system patterns in an enterprise context. His work is one of the first patterns works linking domain models with distributed systems designs using pattern-based design advice.

Richardson [26] presents microservice patterns which are linked in examples to DDD models. Some solutions for those patterns relevant to microservice API endpoints are in our model offered as ADD options. None of these approaches is supported by automated detectors as in our approach yet.

Our work considers solutions akin to such patterns in the decision options, and the patterns contain forces akin to the decision drivers which have led to our ground truth assessment. However, none of the pattern works contains an automated approach, as proposed in our work.

7.2. Related Works on Conformance Assessment

Conformance assessment has been applied in various areas of software engineering such as service composition [16] and traceability to guidelines [42]. In general, the conformance relation is defined as the consistency between models [16]. In software architecture, conformance assessment addresses relation between a software system's architecture and its intended architecture [43]. With those works our approach shares the general notion of architectural conformance assessment.

We study conformance of an ADD model and a microservice API model for API endpoints. Very close to our work is architectural conformance assessment [44, 28, 29] in which first metrics are derived from a knowledge source and then those metrics are compared to expert judgments based on patterns. Our work has some commonalities with those earlier works, such as deriving a scoring scheme based on human judgement (in the earlier works from patterns; in our work from an empirical study of ADDs) and evaluating the investigated systems using the scoring scheme.

In contrast to these works, our work uses detectors instead of special-purpose metrics to automatically detect the core decision points from an empirically grounded ADD model. The reason for the differences is that we investigate ADDs on the mapping of API and DDD models, whereas the prior works study conformance to architectural microservice patterns. To establish traceability and clear judgments on each decision point in the more complex mappings we investigated, we required more detailed assessments (Success, Failure, and Partial Success) with links to the identified violations for traceability, and reasons for detected failures (see Section 5) for details).

7.3. Related Works on Checking Local APIs

Prior works have proposed various approaches to perform checks of certain kinds of conformance for local APIs or API libraries as opposed to the remote, message-based microservice APIs in focus of our work. For example, Zhong et al. [45]

propose a method for deriving an API specification from a natural language API documentation. They show that APIs specifications can be inferred with high accuracy scores and the derived specifications can help to find defects in the projects. Tan et al. [46] propose an approach to test the conformance of Javadoc comments and source code, specifically for testing method properties about null values and related exceptions. The approach is tested on six open-source projects and 24 inconsistencies between Javadoc comments and method bodies have been found.

In the PaRu approach [47] API parameter rules describe constraints on parameters of API methods. These can be described as document rules in Javadoc or code rules where the rule is inferred from the source code. PaRu enables the automatic extraction of such local API parameter rules. It can also find overlaps between these two types of rules, which very seldomly occur in the studied open-source systems. In contrast, if a DDD model is modeled for an API, the API and DDD models addressed in our approach should always have substantial overlaps, as one is derived from the other, and it is the goal of our approach to infer whether the two kinds of models are correctly mapped according to the ADDs informally expressed in Section 2.

Such local API approaches are substantially different from the conformance checking provided in our approach, as firstly they do not address properties that require architectural abstraction from the code, such as our ADDs. Thus, the approaches can check directly against source code fragments, and the code's documented defects or existing inconsistencies can be used for building a ground truth. In contrast, as the realization of ADDs cannot easily be spotted from the code, we needed to design our detector approach and construct the ground truth based on our scoring scheme.

Secondly, local APIs do not expose the level of complexity of interaction that remote, message-based APIs offer, e.g. none of the ADDs described in Section 2 needs to be considered at all in a local context. Finally, our focus is not on the link of documentation/comments to source code fragments, but on the consistency between API and DDD models. Nonetheless, the named approaches use natural language-based methods, and it would be interesting to study as possible future work if such approaches would work well in our context, too, e.g. applied on structured API documentation.

7.4. Related Works on Microservices/APIs and DDD

Various approaches to modeling microservices with DDD, or integrating the two concepts have been proposed [9, 10, 48, 49, 50]. Evans [9] a microservice partitioning approach based on DDD's Bounded Contexts. Merson and Yoder suggest five strategies for microservice design based on DDD aggregates, bounded contexts (BC), domain events and other strategies. Rademacher et al. [48] propose a UML profile for Domain-driven microservice modeling. In another work, Rademacher et al. [49] discuss challenges of using DDD modeling for microservice architectures in the context of model-driven software development. Pautasso et al. [50] discuss the use of DDD in the context of microservice design in practice. Fan and Ma [51] provide an experience report on migrating a mobile application to microservices, in which they use DDD concept to establish the mapping.

Our approach includes modeling and integrating such concepts, with more detailed mappings to API concepts than prior approaches, and in contrast to the other approach also offers conformance assessment. While most of the named approaches consider API aspects, most cover broader microservice architecture aspects as well, which are not part of our approach.

The Context Mapper tool [52] goes one step further than the other microservice/DDD integration approaches and generates various kinds of API specifications from high-level DDD domain models. It is a tool that implements parts of the design options covered in this paper, especially those options related to the API contracts, including links and operations. In this it is complementary to our work, as our Context Mapper could help in the manual creation of the models required for our approach, and our detectors could check models created with Context Mapper.

Petrillo et al. [53] conducted an empirical study to answer the question how well REST APIs for cloud computing are designed. The conformance of these REST APIs is evaluated by investigating best practices. While their work concentrates on REST APIs only, our work focus on wider range of microservices APIs and includes the DDD and modeling perspective.

CHARMY [54] aims to provide a tool for the model-based design and validation of software architectures. In contrast to our work, it is focused on the early stages of software development.

8. Threats to Validity

Our ground truth assessment depends on the interpretation of the ADDs, and different practitioners might come to slightly different assessments. We mitigated this subjectivity by comparing the ground truth assessment in each iteration of our research study to multiple data points from our prior study [11]. In this empirical work, we considered a relatively high number of practitioner sources (32). Nonetheless, some misinterpretation or bias could have been introduced.

Regarding internal validity, we avoided researcher bias by faithful modeling from the evidence-based information. However, the modeling process can be another source of an internal validity threat. We mitigated this threat by independently cross-checking our models numerous times in the author team. Each author studied the sources line-by-line, and then refined them in at least five alternating iterations by both authors, until consensus of correct and consistent modeling was reached. We also confirmed that all modeled practices conform to industry best practices reported in [11]. That is, our modeling procedure has produced acceptable models with a high degree of confidence. Even if minor misinterpretations made their way into our models, as all models follow existing industry best practices, the systems still could have been implemented in this way. This means, such misinterpretations might invalidate the empirical results for a specific model (which are a by-product of our study, not the study goal), but not the evaluation results of our automation approach.

Regarding validity of the detector results, there is a threat that our detector implementation contains defects. To mitigate this threat, each detector implementation and changed detector result, throughout our study, was double-checked independently by both authors. Due to the full traceability provided by our approach, our implementation alerts us of changes. It offers a full trace to respective detector causing a change, its status (Success, Failed, or Partial Success), the affected modeling elements (violations), and a reason for the outcome (see Section 5). Upon a change in an assessment, both authors inspected the change in-depth to determine whether the result conforms to the expected result.

In addition to these measures, during writing of this article both authors independently checked the complete data and all models of this study 3 times each, to avoid mistakes in earlier steps of the research. We also provide all artifacts (code, data set) as open access artifacts to enable reproducibility, so that our modeling and assessments can be independently reproduced.

To avoid system composition and structure bias, we studied many cases in a multi-case study from various third-party authors. Also, the generalizability (external validity) is increased due to the broad range of third-party systems. Nevertheless the threat to validity remains that most of our systems and the case authors have a business/enterprise system background (where DDD is usually applied). Thus our results might not be easily transferable to other contexts such as embedded systems.

Further, some systems are built for demonstration purpose. Thus, it is possible that some aspects important in full-scale commercial systems are missing. To mitigate these threats, we aimed in our selection of the systems, summarized in Table 3, to cover many different kinds of domains (all within the broader enterprise system context), including purchase ordering, publication management, banking, shopping, process tools, game-related knowledge-bases, disease statistics, and insurance. In addition, they cover all of the possible decision options in our ADDs in many different combinations.

The search process for the case systems might have led to the unconscious exclusion of certain sources. We mitigated this by collecting a relatively high number of cases (12), and checking for each the background, for example including that all case authors are practitioners or have a practitioner background.

The construction of our scoring scheme is based on an interpretation and aggregation of practitioner texts in a qualitative, empirical study [11]. While precise decision drivers and impacts have been identified in the empirical study and followed by us in our scheme, an exact mapping e.g. to crisp numerical assessments would have introduced a significant threat of misinterpretations. In contrast, the used ordinal scale enabled us to turn the qualitative practitioner judgments into numerical information, significantly reducing this threat.

While ordinal scales are commonly used to reflect qualitative judgments [55, 56, 57], the threat to validity remains that some interpretations might not reflect the practitioner judgments accurately. Please note that this threat is mitigated by the fact that our study in first place provides a method for automation, not an empirical assessment of 12 system. If others have a different interpretation of some practitioner judgments in [11], it is easily possible to adapt the respective failing detector(s) accordingly.

As we provide all artifacts (code, data set) as open access artifacts to enable reproducibility, such a calibration of our approach can be easily performed. Our approach even provides traceability helping to locate the failing detectors. The concrete system assessment results in Table 9 would then change, but the automation approach would not require any alterations.

Our approach assumes that there is a domain model in an API project that follows DDD practices. We conducted a study of practitioners' work in our previous empirical study [11], which shows that domain models for API design are indeed commonly used in practice. But of course, for many existing APIs, DDD or domain models might not be available. This would mean that before applying our approach in such projects, a manageable effort is required on the part of the domain experts or developers of such a project to model the relevant domain model extract for the API. The conformance testing that our approach provides is intended to be used to continuously check conformance, e.g., as part of a continuous delivery pipeline. Therefore, in projects where developers need to check APIs frequently, modeling the domain model in conjunction with our approach could actually lead to a reduction in effort. In addition, one gains the other benefits of such domain models such as an easier and better understanding of the API domain. However, there remains the threat that in some API projects the assumption that a domain model exists or can be created with little effort is not justified. In these projects, our approach would then not be applicable.

9. Conclusions

In this paper, we introduced an automated assessment approach for conformance to ADDs on API endpoint design based on DDD domain models, specifically focusing on operations, links, and resource segregation (to answer **RQ1**). Such an assessment is mainly required in the work for API developers and architects. Using the data set from an empirical study on those ADDs, we created an empirically grounded scoring scheme. Based on this, we developed novel automated detectors to identify each of the decision points in our scoring scheme.

We evaluated this approach in a multi-case study in which we compared the manually created ground truth to the detector results. In the cases of the multi-case study we were able to identify 83% of the decision points from our scoring scheme correctly (to answer **RQ2**). The remaining approximately 17% are those cases where more detailed modeling of aspects beyond architecture-level models, e.g. with a simple additional flag, could have solved the issue. That is, a single manual inspection and correction could lead to correct detection in future runs, e.g. in a continuous delivery pipeline context. To confirm the results, we performed a statistical evaluation which showed no statistically significant difference between the two variables (ground truth and automated detector results), as well as a negligible effect size between the two variables. In general, manually establishing ADD conformance requires high and continuous effort. Our detectors can spot the violations automatically, support the human-in-the-loop, and support the automated assessment.

Our results indicate that such an automated approach is needed for architectural conformance checking, especially when a frequent release schedule is used, as often repeated manual checking is rather infeasible. A modeling framework and traceability support, as provided in our approach, can help to adapt or calibrate the approach to a given system setting and set of best practices, and create or improve detectors in a straightforward manner with low effort. In our approach all aspects, except the case study modeling were automated; we discuss in Section 3 how this step can be automated, too, using our existing static code analysis approach [33].

In our future work, we plan to work on other ADDs for API design, as well as related design patterns. We aim to apply our approach as part of Continuous Integration/Delivery (CI/CD) pipelines which perform a series of related architectural conformance assessments. Additionally, it would also be interesting to study the combination of our approach and natural language-based approaches, e.g. to obtain information from architecture documentations.

Acknowledgments. This work was supported by FWF (Austrian Science Fund) project API-ACE: I 4268. Our work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952647 (AssureMOSS project).

References

[1] O. Zimmermann, Microservices tenets, Computer Science-Research and Development 32 (3-4) (2017) 301–310. doi:10.1007/s00450-016-0337-0.

[2] O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, U. Zdun, Introduction to microservice api patterns (map), Post-proceedings of Microservices 2017/2019 78 (4) (2020) 1–17. doi:10.4230/OASIcs.Microservices.2017-2019.4.

[3] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly, 2015.

[4] O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, U. Zdun, Microservice api patterns, https://microservice-api-patterns.org/ (2021).

[5] U. Zdun, M. Stocker, O. Zimmermann, C. Pautasso, D. Lübke, Guiding architectural decision making on quality aspects in microservice APIs, in: 16th International Conference on Service-Oriented Computing ICSOC 2018, 2018, pp. 78–89.
URL http://eprints.cs.univie.ac.at/5956/

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

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

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

[9] E. Evans, Ddd and microservices: At last, some boundaries!, https://www.youtube.com/watch?v=sFCgXH7DwxM (2016).

[10] P. Merson, J. Yoder, Modeling microservices with ddd, in: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), IEEE, 2020, pp. 7–8.

[11] A. Singjai, U. Zdun, O. Zimmermann, Practitioner views on the interrelation of microservice apis and domain-driven design: A grey literature study based on grounded theory, in: 18th IEEE International Conference on Software Architecture (ICSA 2021), IEEE, Washington, DC, USA, 2021, pp. –.

[12] L. Lee, P. Kruchten, A tool to visualize architectural design decisions, in: International Conference on the Quality of Software Architectures, Springer, 2008, pp. 43–54.

[13] M. Schwarz, Uber engineering's micro deploy: Deploying daily with confidence, https://eng.uber.com/micro-deploy-code/ (2016).

[14] Google, Devops tech: Architecture, https://cloud.google.com/architecture/devops/devops-tech-architecture (2021).

[15] C. D. Nguyen, A design analysis of cloud-based microservices architecture at netflix, https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45fe (2020).

[16] D. Quartel, M. van Sinderen, On interoperability and conformance assessment in service composition, in: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), IEEE, 2007, pp. 229–229.

[17] D. Lübke, O. Zimmermann, C. Pautasso, U. Zdun, M. Stocker, Interface evolution patterns: balancing compatibility and extensibility across service life cycles, in: Proceedings of the 24th European Conference on Pattern Languages of Programs, 2019, pp. 1–24.

[18] V. R. Basili, D. M. Weiss, A methodology for collecting valid software engineering data, IEEE Transactions on Software Engineering SE-10 (6) (1984) 728–738. doi:10.1109/TSE.1984.5010301.

[19] O. Zimmermann, J. Koehler, F. Leymann, R. Polley, N. Schuster, Managing architectural decision models with dependency relations, integrity constraints, and production rules, J. Syst. Softw. 82 (8) (Aug. 2009). doi:10.1016/j.jss.2009.01.039.
URL `http://dx.doi.org/10.1016/j.jss.2009.01.039`

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

[21] V. Garousi, M. Felderer, M. V. Mäntylä, A. Rainer, Benefitting from the grey literature in software engineering research, in: Contemporary Empirical Methods in Software Engineering, Springer, 2020, pp. 385–413.

[22] J. R. Lourenço, B. Cabral, P. Carreiro, M. Vieira, J. Bernardino, Choosing the right nosql database for the job: a quality attribute evaluation, Journal of Big Data 2 (1) (2015) 1–26.

[23] D. Rowe, J. Leaney, D. Lowe, Defining systems evolvability-a taxonomy of change, Change 94 (1994) 541–545.

[24] H. P. Breivold, I. Crnkovic, Analysis of software evolvability in quality models, in: 2009 35th Euromicro Conference on Software Engineering and Advanced Applications, IEEE, 2009, pp. 279–282.

[25] C. B. Weinstock, J. B. Goodenough, On system scalability, Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST (2006).

[26] C. Richardson, A pattern language for microservices, `http://microservices.io/patterns/index.html` (2017).

[27] O. Zimmermann, Domain-driven service design with context mapper and mdsl, `https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html` (2020).

[28] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, S. Geiger, Assessing architecture conformance to coupling-related patterns and practices in microservices, in: European Conference on Software Architecture, Springer, 2020, pp. 3–20.

[29] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, S. Geiger, Metrics for assessing architecture conformance to microservice architecture patterns and practices, in: International Conference on Service-Oriented Computing, Springer, 2020, pp. 580–596.

[30] A. Singjai, U. Zdun, Conformance Assessment of Architectural Design Decisions on API Endpoint Designs Derived from Domain Models: Dataset and Code (Jun. 2021). doi:10.5281/zenodo.5031272.
URL `https://doi.org/10.5281/zenodo.5031272`

[31] P. Runeson, M. Host, A. Rainer, B. Regnell, Case study research in software engineering: Guidelines and examples, Wiley, 2012.

[32] J. Piasecki, M. Waligora, V. Dranseika, Google search as an additional source in systematic reviews, Science and engineering ethics 24 (2) (2018) 809–810.

[33] E. Ntentos, U. Zdun, K. Plakidas, P. Genfer, S. Geiger, S. Meixner, W. Hasselbring, Detector-based component model abstraction for microservice-based systems, Computing 103 (2021) 2521–2551.

[34] S. S. Shapiro, M. B. Wilk, An analysis of variance test for normality (complete samples), Biometrika 52 (3/4) (1965) 591–611.

[35] J. W. Pratt, Remarks on zeros and ties in the wilcoxon signed rank procedures, Journal of the American Statistical Association 54 (287) (1959) 655–667.

[36] N. Cliff, Ordinal methods for behavioral data analysis, Psychology Press, 2014.

[37] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, A. Pohthong, Robust statistical methods for empirical software engineering, Empirical Software Engineering 22 (2) (2017) 579–630.

[38] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys, in: annual meeting of the Florida Association of Institutional Research, Vol. 13, 2006, pp. –.

[39] O. Zimmermann, D. Lübke, U. Zdun, C. Pautasso, M. Stocker, Interface responsibility patterns: Processing resources and operation responsibilities, in: Proceedings of the European Conference on Pattern Languages of Programs 2020, EuroPLoP '20, ACM, New York, NY, USA, 2020, pp. –. doi:10.1145/3424771.3424822.

[40] G. Hohpe, B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003.

[41] R. Daigneau, Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services, Addison-Wesley, 2011.

[42] P. Rempel, P. Mäder, T. Kuschke, J. Cleland-Huang, Mind the gap: assessing the conformance of software traceability to relevant guidelines, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 943–954.

[43] F. Deissenboeck, L. Heinemann, B. Hummel, E. Juergens, Flexible architecture conformance assessment with conqat, in: 2010 ACM/IEEE 32nd International Conference on Software Engineering, Vol. 2, 2010, pp. 247–250. doi:10.1145/1810295.1810343.

[44] U. Zdun, E. Navarro, F. Leymann, Ensuring and assessing architecture conformance to microservice decomposition patterns, in: International Conference on Service-Oriented Computing, Springer, 2017, pp. 411–429.

[45] H. Zhong, L. Zhang, T. Xie, H. Mei, Inferring resource specifications from natural language api documentation, in: 2009 IEEE/ACM International Conference on Automated Software Engineering, 2009, pp. 307–318. doi:10.1109/ASE.2009.94.

[46] S. H. Tan, D. Marinov, L. Tan, G. T. Leavens, @tcomment: Testing javadoc comments to detect comment-code inconsistencies, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012, pp. 260–269. doi:10.1109/ICST.2012.106.

[47] H. Zhong, N. Meng, Z. Li, L. Jia, An empirical study on api parameter rules, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 2020, pp. 899–911.

[48] F. Rademacher, S. Sachweh, A. Zündorf, Towards a uml profile for domain-driven design of microservice architectures, in: International Conference on Software Engineering and Formal Methods, Springer, 2017, pp. 230–245.

[49] F. Rademacher, J. Sorgalla, S. Sachweh, Challenges of domain-driven microservice design: a model-driven perspective, IEEE Software 35 (3) (2018) 36–43.

[50] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, N. Josuttis, Microservices in practice, part 1: Reality check and service design, IEEE software 34 (01) (2017) 91–98.

[51] C.-Y. Fan, S.-P. Ma, Migrating monolithic mobile application to microservice architecture: An experiment report, in: 2017 IEEE International Conference on AI & Mobile Services (AIMS), IEEE, 2017, pp. 109–112.

[52] S. Kapferer, 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, Feb 2020, Scitepress, 2020, pp. 299–306. doi:10.5220/0008910502990306.

[53] F. Petrillo, P. Merle, N. Moha, Y.-G. Guéhéneuc, Are rest apis for cloud computing well-designed? an exploratory study, in: International Conference on Service-Oriented Computing, Springer, 2016, pp. 157–170.

[54] P. Pelliccione, P. Inverardi, H. Muccini, Charmy: A framework for designing and verifying architectural specifications, IEEE Transactions on Software Engineering 35 (3) (2008) 325–346.

[55] O. Larichev, H. Moskovich, Unstructured problems and development of prescriptive decision making methods, in: Advances in multicriteria analysis, Springer, 1995, pp. 47–80.

[56] W. D. Perreault Jr, L. E. Leigh, Reliability of nominal data based on qualitative judgments, Journal of marketing research 26 (2) (1989) 135–148.

[57] G. Canfora, L. Cerulo, L. Troiano, Transforming quantities into qualities in assessment of software systems, in: Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003, IEEE, 2003, pp. 312–319.