Guiding Architectural Decision Making on Quality Aspects in Microservice APIs

Uwe Zdun[1](✉), Mirko Stocker[2], Olaf Zimmermann[2], Cesare Pautasso[3], and Daniel Lübke[4]

[1] Faculty of Computer Science, Research Group Software Architecture,
University of Vienna, Vienna, Austria
uwe.zdun@univie.ac.at

[2] University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland
{mirko.stocker,olaf.zimmermann}@hsr.ch

[3] Software Institute, Faculty of Informatics, USI, Lugano, Switzerland
cesare.pautasso@usi.ch

[4] innoQ Schweiz GmbH, Cham, Switzerland
ich@daniel-luebke.de

Abstract. Microservice APIs represent the client perspective on microservice-based software architecture design and related practices. Major issues in API design concern the quality aspects of the API. However, it is not well understood today what the established practices related to those quality aspects are, how these practices are related, and what the major decision drivers are. This leads to great uncertainty in the design process. In this paper, we report on a qualitative, in-depth study of 31 widely used APIs plus 24 API specifications, standards, and technologies. In our study we identified six recurring architectural design decisions in two API design contexts with a total of 40 decision options and a total of 47 decision drivers. We modelled our findings in a formal, reusable architectural decision model. We measured the uncertainty in the resulting design space with and without use of our model, and found that a substantial uncertainty reduction can be potentially achieved by applying our model.

1 Introduction

Many approaches have been proposed for designing service-based architectures (see e.g. [15,17,23]). A recent approach which evolved from established practices in service-oriented architectures are *microservices* [14,22]. The microservices approach emphasizes business capability- and domain-driven design, service development in independent teams, cloud-native technologies/architectures, polyglot persistence, lightweight containers, and a continuous DevOps approach to service delivery (see [11,14,22]). When realizing microservices architectures a core task is to design the service contracts or Application Programming Interfaces (APIs). In this context, we focus on the problem to design for, realize, enforce and maintain *quality aspects of the microservice API* – which are of

C. Pahl et al. (Eds.): ICSOC 2018, LNCS 11236, pp. 73–89, 2018.
https://doi.org/10.1007/978-3-030-03596-9_5

key importance as the API is usually the only visible aspect of the microservice from the client's perspective. An API provider has to perform the balancing act of providing a high-quality service in a cost-effective way. Quality of an API has many dimensions, starting with the functionality, but also including many other qualities such as reliability, performance, security, and scalability – sometimes the latter are referred to as Quality of Service (QoS) guarantees. They are usually conflicting with each other, but almost always need to be balanced with economic qualities such as costs and time to market. Many quality measures related to service QoS exist, but only a few of them are directly related to APIs [13].

The main challenge for microservice API designers is to determine the appropriate quality trade-off during the design of the microservice API. Numerous practices exist and have complex relations among each other. Many decision drivers have to be understood and might be conflicting among each other. Therefore, architects orienting themselves and navigating in the microservice API design space usually face a high uncertainty in their decision making, related to finding and assessing the knowledge needed for making an informed decision. Once all required knowledge has been gathered, a high uncertainty on how to combine which practices remains; the impact of these practices and their combinations on the many potentially relevant quality trade-offs is not clear either. This paper aims to study the following resulting research questions:

- **RQ1 (a)** What are established practices to design for, realize, communicate and maintain the quality of a microservice API? **(b)** What are the relations among those practices? **(c)** What are decision drivers of those decisions? **(d)** Which impact do the practices and their combinations impose on the decision drivers?
- **RQ2 (a)** How high is the decision making uncertainty in this design space? **(b)** Can this decision making uncertainty be reduced? If so, how?

This paper makes three major contributions. First, we gather knowledge about established practices, their relations, and their decision drivers in the form of a **microservice APIs design space based on a qualitative study** of 55 knowledge sources (including 31 widely used APIs). Our second contribution is the codification of this knowledge in form of a **reusable Architectural Design Decision (ADD) model** which we formally modelled based on a UML2 meta-model. We also described newly documented patterns using pattern templates (which can be found in another publication along with technical details [19], whereas this paper focuses only on decision modelling aspects). In total we documented six decisions in two contexts with 40 decision options and 47 decision drivers. Please note that we limited our scope to message representations in the interface contracts, and excluded e.g. the architectural decisions required in service implementations (which were addressed in our earlier works [23]). Finally, we **estimate the decision making uncertainty** in this design space, calculate the uncertainty left after applying the guidance of our ADD model, and compare the two. Our model shows a potential to substantially reduce the uncertainty not only by documenting established practices, but also by organizing the knowledge in a model.

The remainder of this paper is organized as follows: In Sect. 2 we compare to the related work. Section 3 explains the research methods we have applied in our study. Then Sect. 4 describes our reusable ADD model. Section 5 provides the uncertainty reduction estimation. The findings are discussed in Sect. 6, and Sect. 7 concludes.

2 Related Work

Quite a number of studies on services focus on QoS aspects (see e.g. [13,18,21]). In microservice-specific studies related to quality, topics like the increased operations qualities when combined with DevOps (see e.g. [1]), qualities in service decomposition [7], or specific qualities like trade-offs in self-adaptive architectures [8] are studied. The specific quality aspects of the API, which is of high practical relevance as the API is the only part of the microservices visible to the client, is not yet a major focus of study.

A number of approaches study microservice patterns and best practices: The microservice patterns by Richardson [17] address microservice design and architecture practices. Another set of patterns on microservice architecture structures has been published by Gupta [6], microservice best practices are discussed in [11], and similar approaches are summarized in a recent mapping study [15]. So far, none of those approaches has been combined with a formal model and API quality is not a major focus.

Decision documentation models (examples are those covering service-oriented solutions [23], service-based platform integration [12], REST vs. SOAP [16], and big data repositories [5]) promise to improve the situation, but the focus on this kind of research is not yet on API design. The model developed in our study can be classified as a reusable ADD model [23]. Other authors have combined decision models with formal view models [9]. We apply those techniques in our work, but also extend them with a modelling approach and a detailed uncertainty reduction estimation. Exploiting uncertainties has been used in software architecture traceability research before [20].

3 Research Method

This paper aims to systematically study the established practices in the field of microservice API quality aspects. A number of methods have been suggested to study established practices. A classical method is pattern mining (see e.g. [3]) which starts with the authors' own experiences, searches systematically for other known uses in real-life systems, and then applies a series of feedback loops to improve the pattern. A number of techniques have been suggested for improving this research method. Hentrich et al. [10] define a pattern mining method as a form of qualitative research resembling methods like Grounded Theory (GT) [4]. Like GT, we studied each knowledge source in depth. We followed a similar coding process, as well as a constant comparison procedure to derive our model. In contrast to classical GT, our research began with initial research questions, as in Charmaz's constructivist GT [2]. Whereas GT typically uses textual analysis, we used textual codes only initially and then transferred them into formal UML models and text in pattern templates.

Our knowledge mining happened in many iterations. That is, we searched for one or a few new knowledge sources, applied open and axial coding [4] to identify candidate categories, and compared with the so-far-designed model continuously. We improved this model incrementally. A crucial question in GT is when to stop this process; here, theoretical saturation [4] has attained widespread acceptance in qualitative research: We stopped our analysis when 5 to 7 additional knowledge sources did not add anything new to our understanding of the research topic. As a result of this very conservative operationalization of theoretical saturation, we studied a rather large number of knowledge sources in depth (55 in total, summarized in Table 1, see [19] for more details on the sources), whereas most qualitative research often saturates with a much lower number of knowledge sources. In addition to 31 APIs and their documentations, our search led us to 24 additional knowledge sources (specifications, standards, and technologies) directly related to the design of a number of APIs. Our search was based on our own experience, APIs we have access to and worked with. We also used on major search engines (e.g., Google, Bing) and online API directories and topic portals (e.g., ProgrammableWeb, InfoQ) to find known uses and validate intermediate results. We included knowledge sources, if they were about widely used APIs (i.e., many more users than just the original authors), use modern service technologies, and follow at least some of the microservice tenets summarized in Sect. 1. Note that not always those APIs are labelled as microservice APIs, but as microservice tenets have been in use long before the microservice term was coined, and as some RESTful HTTP APIs share the same technological underpinnings, we have also considered them in our study.

Table 1. Knowledge sources included in the study

APIs studied	31	Amazon EC2 API, Amazon S3, AWS Lambda, Cloud Convert API, Confluence REST API, Facebook Graph API, File Transfer Service API, Finance Industry Web Service API, GitHub API v3, GitHub API v4, Google Calendar API, Google Compute Engine, JIRA Cloud API, LinkedIn API, Microsoft Azure, Microsoft Dynamics CRM, Microsoft Graph API, Open Weather Map, Optimizely, PayPal API, Quandl API, Salesforce API, Singlewire, Stripe API, SWIFT, Swiss Bank API, Swiss Federal Administration registry of companies web service API, Swiss Insurance API, TMForum REST API, Twitter API, YouTube Data API
API-related specifications, standards, technologies studied	24	Adidas API Spec, Amazon API Gateway, apistylebook.com, Basic Authentication, CHAP, EAP, EC SLA Guidelines, HTTP/1.1: Conditional Requests, JSON API Spec, Kerberos, LDAP, MuleSoft API Manager, OAuth, OpenID Connect 1.0, OWASP REST Security, Play2 Guard, REST API design book, RESTful Web Services Cookbook, RFC 7519, SAML, SLA Best Practices, SLA Whitepaper, Suggested REST Practices, TM Forum Applications Framework 3.0

4 Reusable ADD Model for API Quality

In this section, we report on the reusable ADD model that resulted from our study. Figure 1 shows an overview of the reusable decisions and their relationships, as well as their major decision contexts. Our model contains all kinds of *decision contexts in API design* in a separate domain model of which we only use a small part in this paper: Most decisions on API quality have to be made for *combinations of API clients and the API* those clients access. Many such decisions can be made for large groups of those combinations, such as all clients with freemium access or all clients accessing a specific API. One decision needs to be made at the level of *operations* in the API. Note that in Fig. 1 the decisions inherit those contexts from their (sub-)categories. Below we describe our findings for each of the decisions in detail. Note that all elements of our reusable ADD model are instances of a meta-model (with meta-classes such as *Decision*, *Category*, *Pattern*, *AND Combined Group*, and so on), which we introduce in this paper implicitly in the text with the uses of the meta-classes in our model. Please note that in this paper we focus on the decision models and their decision drivers. For space reasons, it is not possible to provide all technical details. Detailed patterns explaining the decision options used in this paper, including examples, known uses, and detailed discussions of decision drivers can be found in another publication describing the patterns using pattern templates [19]. The patterns are part of a larger pattern language effort started and summarized in [24].

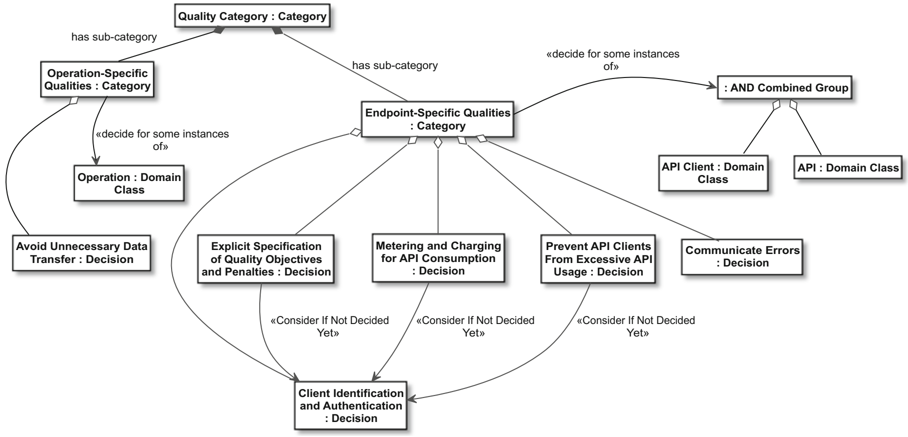
Quality Category : Category
has sub-category
has sub-category
Operation-Specific Qualities : Category
«decide for some instances of»
Operation : Domain Class
Avoid Unnecessary Data Transfer : Decision
Endpoint-Specific Qualities : Category
«decide for some instances of»
: AND Combined Group
API Client : Domain Class
API : Domain Class
Explicit Specification of Quality Objectives and Penalties : Decision
Metering and Charging for API Consumption : Decision
Prevent API Clients From Excessive API Usage : Decision
Communicate Errors : Decision
«Consider If Not Decided Yet»
«Consider If Not Decided Yet»
«Consider If Not Decided Yet»
Client Identification and Authentication : Decision

Fig. 1. API quality: reusable ADD model – overview of required decisions and categories

Reusable Decision: Identification and Authentication of the API Client. Identification and authentication are important for API providers that are paid for or use freemium models: the API provider can grant authorization based on the API client's proven identity. This is key to ensure security, but also impacts many other qualities; e.g., if unknown clients can access the API without control or known clients can make excessive use of the API, the performance of the system can degrade, reliability can be in danger, or costs (e.g., for used cloud resources) can rise. The typical decision to be made here is shown in Fig. 2. The simplest option is to chose *no secure identification and authentication needed*, which is suitable only if the a number of clients is limited and if the risks with respect to abuse or excessive use are low. The obvious alternative is to introduce an *Authentication* mechanism for the API (which includes identification). An *API Key* that assigns each client a unique token that the client can present to the API endpoint for identification is a minimalistic solution. If security is an issue, *API Keys* are not enough. In conjunction with an *additional secret key* that is not transmitted, *API Keys* can be used to securely authenticate a client. Another secure alternative are *authentication* or *authorization protocols* such as OAuth, SAML, Kerberos, or LDAP.

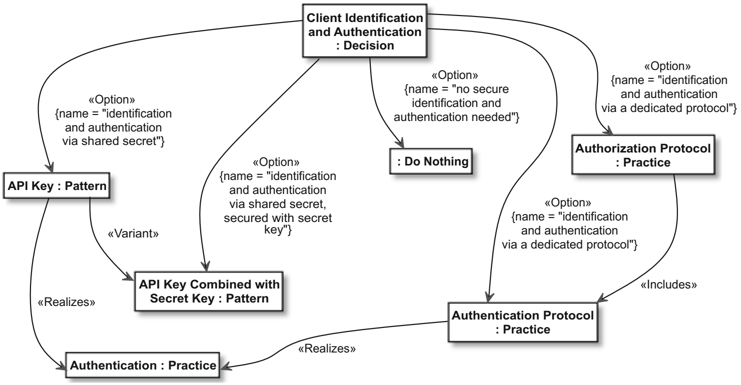
Client Identification and Authentication : Decision
«Option» {name = "identification and authentication via shared secret"}
«Option» {name = "identification and authentication via shared secret, secured with secret key"}
«Option» {name = "no secure identification and authentication needed"}
«Option» {name = "identification and authentication via a dedicated protocol"}
«Option» {name = "identification and authentication via a dedicated protocol"}
API Key : Pattern
: Do Nothing
Authorization Protocol : Practice
«Variant»
API Key Combined with Secret Key : Pattern
«Realizes»
«Includes»
Authentication Protocol : Practice
«Realizes»
Authentication : Practice

Fig. 2. Client identification and authentication decision

There are a number of decision criteria that need to be considered in this decision. First of all, the *level of required security*, as outlined above. In addition, *API Keys* are only a slight degradation in terms of *ease of use for clients* compared to doing nothing; the other options are less easy to use as they require dealing with more complex protocol APIs and setting up the required infrastructure. In addition, the *management of user account credentials* required in *authentication and authorization protocols* can be tedious both on client and provider side; this is avoided in all options using *API Keys*. With regard to the performance of the solution, doing nothing has no overhead. *API Key* options have a slight overhead for processing the key(s). *Authentication and authorization protocols* tend to have more overhead as they also offer additional features. The *API Key* options also *decouple the client making an API call from the client's organization*, as using the customer's account credentials would needlessly give system administrators and developers full account access.

Please that in our models we use the term practice as a superset of patterns and other established practices; only those practices that have been described in pattern form (e.g., in our related publications [19]) are denoted with the stereotype *Pattern*, all other existing practices are denoted as *Practice*.

Reusable Decision: Communicate Errors. A common quality concern for APIs is how to communicate errors as this has direct impacts on qualities like avoiding and fixing defects, costs of defect fixing, robustness and reliability problems due to unfixed defects, and so on. Of course, one option is not to handle the error at all, but this is usually not advisable – at least, for production APIs. A common solution, if only one protocol stack is used (e.g., HTTP over TCP/IP), is to use *Protocol-level Error Codes*, e.g., status codes in HTTP, but this does not work if error reporting needs to work across multiple protocols, formats, and platforms. In such cases the *Error Reporting* pattern should be used so that replies include in addition to machine-readable error codes, also a textual description of the error is provided for the API developer. Also such error messages can carry parameters and constants in order to allow internationalization of error messages when reporting the to the user of the client application.

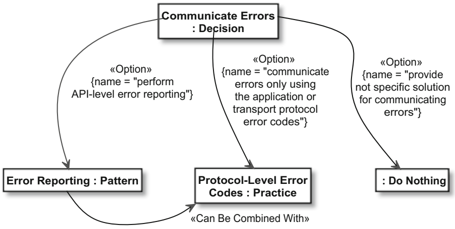
Communicate Errors : Decision
«Option» {name = "perform API-level error reporting"}
«Option» {name = "communicate errors only using the application or transport protocol error codes"}
«Option» {name = "provide not specific solution for communicating errors"}
Error Reporting : Pattern
Protocol-Level Error Codes : Practice
: Do Nothing
«Can Be Combined With»

Fig. 3. Communicate errors decision

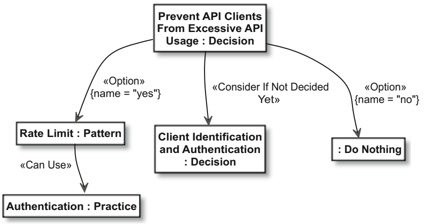
Prevent API Clients From Excessive API Usage : Decision
«Option» {name = "yes"}
«Consider If Not Decided Yet»
«Option» {name = "no"}
Rate Limit : Pattern
Client Identification and Authentication : Decision
: Do Nothing
«Can Use»
Authentication : Practice

Fig. 4. Prevent API clients from excessive API usage decision

The main decision drivers (see Fig. 3) to introduce any kind of error reporting are *help in fixing defects* and increased *robustness* and *reliability*. Error reporting leads to better *maintainability* and *evolvability*, and the more it explains errors and thus reduces the effort in the task of finding the cause of a defect, the more effective it is; thus the *Error Reporting* pattern performs better in this regard than simple error codes. *Error Reporting* is also better performing with regard to *interoperability* and *portability* as it better enables supporting protocol, format, and platform autonomy. However, the more elaborate error messages can reveal information that is problematic with regard to *security*, as revealing more information about system internals opens up attack vectors. *Error Reporting* requires more work, if *internationalization* is required, as the more detailed information needs to be translated.

Reusable Decision: Preventing API Clients from Excessive API Usage. Excessive use by a few clients can significantly limit the availability of the service for other clients. Thus preventing excessive API usage by clients is needed.

Assuming API clients can be identified as previously discussed, their individual usage of the API can be monitored for billing purposes. If offsetting the expense of operating the microservice to its clients is not enough to limit their traffic (e.g., using a *Rate Plan*, see next decision), an explicit *Rate Limit* can be introduced to safeguard against API clients that overuse the API. The limit can be expressed in number of requests per period of time. If this limit is exceeded, further requests can either be declined, be processed later or with lower priority.

The decision is shown in Fig. 4. The major decision criteria to be considered in this decision are: A certain level of *scalability and performance* needs to be maintained by the provider, but could be in danger if clients abuse the API. Means for supporting *client awareness* of *Rate Limits* are required so that clients can find out know how much of their limits they have already used up. Establishing *Rate Limits* helps the provider to support qualities such as *resilience, reliability, and fault tolerance* as they make it hard for clients to abuse the API in a way that puts those qualities at risk. All these potential benefits must be contrasted to the *impact and severity of risks of API abuse* and *economic aspects*. Introducing *Rate Limits* produces costs and can be seen critically by clients as well as additional complexity if clients are allowed to negotiate their limits.

Reusable Decision: Metering and Charging for API Consumption. If the API is a commercial offering, the API provider might want to charge for its usage. Thus a means for identifying and authenticating clients is required (see decision above). Then the provider can monitor clients and assign a *Rate Plan* which measures API usage e.g. on a per-call level and is used to bill API clients, advertisers, or other stakeholders accordingly. As shown in Fig. 5, we can alternatively not meter and charge the client. In the context of a *Rate Plan* sometimes a *Rate Limit* is used to ensure fair use. Figure 5 also illustrates possible variants of the *Rate Plan* pattern: Pricing can be based on actual usage, on

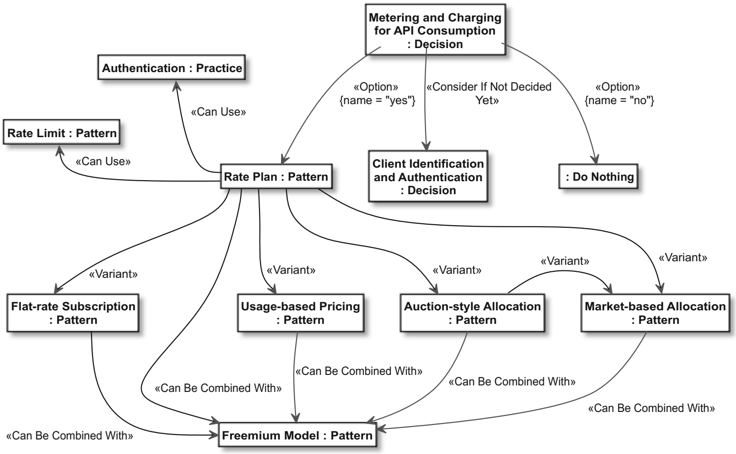

Fig. 5. Metering and charging for API consumption decision

market-based allocation (or with its sub-variant based on auctions), or on flat-rate subscriptions. All those variants can be combined with a freemium model.

The major drivers for this decision are usually *economic aspects*, such as pricing models and selecting a variant of the pattern that suits the provider or the consumer business model best. The benefits of applying the pattern need to be contrasted to the efforts and costs required to meter and charge customers. *Accuracy* is central as API clients expect to be billed only for the services they actually have consumed. Accurate metering requires an adequate *meter granularity* to be defined. As information about metering and charging contains sensitive information about clients, e.g. indications of how well they do in their markets, it needs extra protection with regard to *security*.

Reusable Decision: Explicit Specification of Quality Objectives and Penalties. Quality objectives are kept implicit and vague for many APIs. If the client requires (or even pays for) stronger guarantees or the provider wants to make explicit guarantees (e.g., to differentiate from competitors), an explicit specification of quality objectives and penalties can be considered. This can be done by introducing a *S*ervice Level Agreement (SLA) which is an extension of the API description detailing measurable *Service Level Objectives* (SLOs) and penalties in case of violation. Any *Rate Plan* and *Rate Limit* should refer to the SLA if these patterns are used (and vice versa). SLAs require means for identifying and authenticating clients; usually authentication practices have to be used. There are a number of typical variants of the pattern: SLAs only used for internal use, SLAs with formally specified SLOs, and those with only informally specified SLOs, e.g., with natural language.

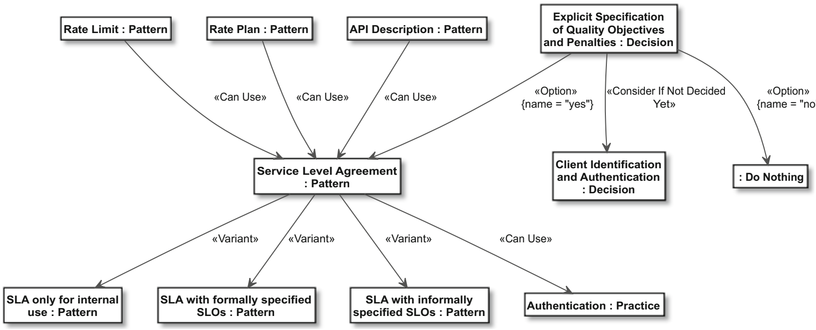
Rate Limit : Pattern
Rate Plan : Pattern
API Description : Pattern
Explicit Specification of Quality Objectives and Penalties : Decision
«Can Use»
«Can Use»
«Can Use»
«Option» {name = "yes"}
«Consider If Not Decided Yet»
«Option» {name = "no"}
Service Level Agreement : Pattern
Client Identification and Authentication : Decision
: Do Nothing
«Variant»
«Variant»
«Variant»
«Can Use»
SLA only for internal use : Pattern
SLA with formally specified SLOs : Pattern
SLA with informally specified SLOs : Pattern
Authentication : Practice

Fig. 6. Explicit specification of quality objectives and penalties decision

As shown in Fig. 6, the main decision drivers are: *Attractiveness from consumer point of view* can be higher if guarantees about qualities can be made. However, this must be contrasted to possible issues related to *cost-efficiency and business risks from a provider point of view*. Some guarantees are required by *government regulations and legal obligations* like those related to personal data protection such as the EU General Data Protection Regulation (GDPR). If a provider intends to make any guarantees about the quality of its service (typical candidates concern the microservice *availability*, *performance and scalability*, or *security and privacy*), then such qualities become decision drivers for this decision. Finally, the decision relates to *business agility and vitality* as the business model of a client might rely on the above named qualities of a service.

Reusable Decision: Avoid Unnecessary Data Transfer. The decision described in this section contains four patterns addressing different situations in which unnecessary data is transferred by the operations of an API. Note that in contrast to the prior decisions, this one needs to be made per operation as only a detailed analysis of the individual needs of the clients can indicate whether the data transfer can be reduced or not.

It can be hard for API providers to design operations that provide the required data exactly: the needs of clients might not be predictable and/or differ from each other. One solution is to let the API client provide a *Wish List* in the request enumerating all desired data elements so that the API provider can deliver only the desired elements in the response. A *Wish List* is not always easy to specify, e.g., if only certain fractions of nested or repetitive parameter structures are required. An alternative that works better for complex parameters is to let the client send a more expressive *Wish Template* that mirrors the structure of the desired responses (but contains dummy data) in its request.

If multiple clients repeatedly request the same data, which seldom changes, unnecessary data transfer can be avoided through a *Conditional Request*. To make requests conditional, they contain additional metadata parameters so that the provider may only process the request if a condition is met; e.g., in RESTful HTTP APIs, the provider could provide a *fingerprint*, which the client can then include in subsequent requests to indicate the latest known version of the resource that the client already has retrieved.

Another scenario is when one client makes multiple related requests that form logical batches. If the provider receives and replies to all requests individually, *performance* and *scalability* may suffer. This can be avoided by defining a *Request Bundle* as a container message that assembles multiple individual requests and is accompanied by metadata such as number of and identifiers of individual requests. Exchanging a single large message is usually more efficient than transferring multiple short messages. This comes at a price of increased effort for request processing on the provider side.

Sometimes no data transfer reduction is possible or wanted for the target operation(s); no action has to be taken in that case. Alternatively, unnecessary data transfer can be avoided through the patterns explained above. A combination of *Conditional Request* with either *Wish List* or *Wish Template* can be useful to indicate which subset of changed data is requested. *Request Bundle* can be combined with any of the prior alternatives, but combining multiple of the patterns increases the *complexity of the API*.

The main decision driver for this decision as illustrated in Fig. 7 is the *individual information needs of a client* which need to be analyzed to find out which of

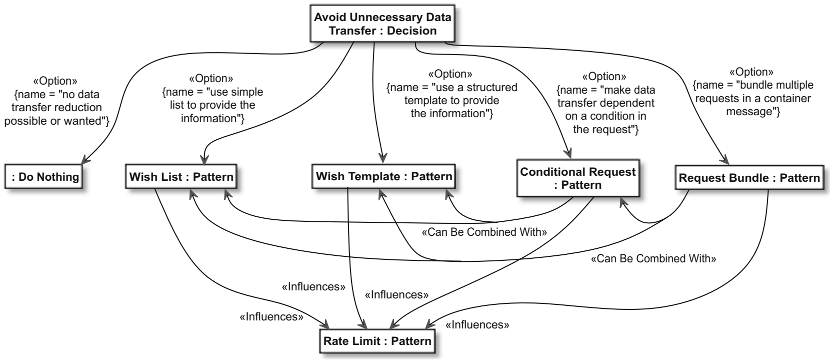
Avoid Unnecessary Data Transfer : Decision
«Option» {name = "no data transfer reduction possible or wanted"}
«Option» {name = "use simple list to provide the information"}
«Option» {name = "use a structured template to provide the information"}
«Option» {name = "make data transfer dependent on a condition in the request"}
«Option» {name = "bundle multiple requests in a container message"}
: Do Nothing
Wish List : Pattern
Wish Template : Pattern
Conditional Request : Pattern
Request Bundle : Pattern
«Can Be Combined With»
«Can Be Combined With»
«Influences»
«Influences»
«Influences»
«Influences»
Rate Limit : Pattern

Fig. 7. Avoid unnecessary data transfer decision

the patterns combinations can provide benefits. Consider situations where data transfer over the network is perceived as a potential bottleneck: *Data parsimony* can further drive the decision as the patterns can help to reduce bandwidth consumption during data transmission. Avoiding unnecessary data transfers can improve *performance*, as transferring all data elements to all clients all the time would harm response time, throughput, processing time, cost, etc. *Security* can be a driver to apply or not to apply the patterns *Wish List* and *Wish Template*: enabling clients to provide options on which data to receive may unwittingly expose sensitive data or open up additional attack vectors. On the other hand, data that is not transferred cannot be stolen and cannot be tampered with. Finally, going from an API with a fixed data representation to an API where clients can dynamically determine what content will be retrieved – which all four patterns do – increases the *complexity of API design and programming.* This is e.g. evidenced in GraphQL which can be seen as an extreme form of *Wish Template*. In addition, the special cases introduced by the patterns cause more *testing* and *maintenance* efforts.

5 Preliminary Estimation of Uncertainty Reduction

Architectural decision making is always tied to the given context; e.g., our ADD model documents decisions in two different contexts $CON = \{API\ \&\ API\ Client,\ API\ Operation\}$. In each context the architect needs to make a set of decisions DEC. For each, $d \in DEC$ there are a number of decision options OPT_d possible to choose for decision d. Finally, there is a set of criteria CRI_d that need to be considered when making a decision d. There are many different kinds of uncertainties involved in making ADDs in a field in which the architect's experience is limited. The obvious contribution of our ADD model is that it helps to reduce the uncertainty whether all relevant, necessary and sufficient elements for making a correct decision have been found (for each of the sets named above CON, DEC, OPT, and CRI). Another kind of uncertainty reduction is the uncertainty reduction our ADD model provides compared to using the same knowledge, but in a completely unorganized fashion. We want to estimate this kind of uncertainty reduction here. Exploiting uncertainties has been used in a similar way in software architecture traceability research before [20]. Our model provides organization to the design knowledge at a number of levels:

- It groups and interrelates decisions; e.g., *Metering and Charging for API Consumption* requires consideration of *Client Identification and Authentication* (see Fig. 5).
- It groups decision options in decisions and interrelates them; e.g., *Perform Error Handling* has three options $OPT_{Perform\ Error\ Handling} = \{$*Error Reporting, Protocol- Level Error Codes, Do Nothing*$\}$. Two of those are related further: *Error Reporting* can be combined with *Protocol-Level Error Codes* (see Fig. 3).
- It associates decision criteria to decisions; e.g., *Metering and Charging for API Consumption* has 4 criteria $CRI_{Metering\ and\ Charging\ for\ API\ Consumption} = \{$*Economic Aspects, Accuracy, Meter Granularity, Security*$\}$ (see criteria explanations for Fig. 5).
- It pre-selects many of those criteria for the options; e.g., in *Metering and Charging for API Consumption*, the criterion *Security* is pre-selected: Using a *Rate Plan* has negative impacts on it; the option *Do Nothing* is preferable as it has no security impact. In contrast, the criterion *Economic Aspects* needs to be investigated further for both decision options in the concrete context and thus cannot be pre-decided.

Here, we estimate the uncertainty reduction both for each individual decision and possible decision combinations in each context (uncertainty reduction estimations are reported as rows in Table 2). We calculate each number both for using our ADD model (denoted with $^{\oplus}$ below) and not using our model (denoted with $^{\ominus}$ below):

- *Number of decisions nodes (ndec):* Our ADD model represents each decision separately. So the number of decision nodes for a single decision d is always $ndec_d^{\oplus} = 1$. Without our ADD model, each design solution (i.e., decision option in the design space) that is not *Do Nothing* is a possible decision node, and it can either be selected or not: $ndec_d^{\ominus} = |OPT_d \setminus \{Do\ Nothing\}|$. Please note that, if a design solution has variants, OPT_d contains the base variant plus each possible variant.
- *Number of required criteria assessments in a decision (ncri):* Our ADD model includes explicit decision criteria per decision. Some of those are pre-decided, others not. Let the functions *decided*() and *undecided*() select them, respectively. If all criteria are decided in a decision, we only require one criteria assessment (assessing the whole vector of decided criteria). If all criteria are undecided, we need to make $|CRI_d|$ assessments of criteria. Often some criteria are decided, others not, so the number of criteria to be decided is in general:

$$ncri_d^{\oplus} = \begin{cases} 1 + |undecided(CRI_d)|, & for\ decided(CRI_d) > 0 \\ |CRI_d|, & for\ decided(CRI_d) = 0 \end{cases}$$

Without our ADD model, we need to assess each criterion for each decision node (as we have no pre-decided choices): $ncri_d^{\ominus} = |CRI_d| \times |ndec_d^{\ominus}|$.

- *Number of possible decision outcomes (ndo):* Our ADD model already models each decision option separately in $|OPT_d|$ including *Do Nothing*, so $ndo_d^{\oplus}$ usually equals $|OPT_d|$ unless the design space allows explicit combinations of solutions as additional outcomes. For instance, in the decision *Metering and Charging for API Consumption* the variant *Freemium Model* can be combined with the base variant and all four other variants, leading to an additional five outcomes. Let the function *solComb*() return the set of possible solution combinations in the options of a decision; then $ndo_d^{\oplus} = |OPT_d| + |solComb(OPT_d)|$.
  The same is true in principle for the decisions made without our ADD model, but as the decision d is here split into multiple separate decision nodes $ndec_d^{\ominus}$ and without the ADD model no information on which combinations are possible is present, we need to consider any possible combination in $ndec_d^{\ominus}$, i.e., the size of the powerset of the decision nodes: $ndo_d^{\ominus} = |\mathcal{P}(ndec_d^{\ominus})| = 2^{|ndec_d^{\ominus}|}$.

For the context *API Operation*, there is only a single decision (i.e., avoid unnecessary data transfer), but in the context *API & API Client* there are five decisions. It is thus also important to calculate the total uncertainty reduction in this context, where any number of those five decisions can be taken. The combinations of *ndec* and *ncri* in a context $c \in CON$ is with or without our ADD model simply their sum for the decisions d in the context c; let *inCon()* be a function selecting all decisions in a context:

$ndec_c = \sum_{d \in \{dec\, \in\, DEC |\, dec\, \in\, inCon(c)\}} |ndec_d|$

$ncri_c = \sum_{d \in \{dec\, \in\, DEC |\, dec\, \in\, inCon(c)\}} |ncri_d|$

If multiple decisions need to be made, the combinations for *ndo* require us to consider all possible combinations of decision outcomes of each and every decision:

$ndo_c = |\mathcal{P}(\bigcup_{d \in \{dec\, \in\, DEC |\, dec\, \in\, inCon(c)\}} ndo_d)|$.

Table 2 shows the results of the uncertainty reduction estimation. As can be seen without our ADD model in general more decision nodes *ndec* need to be considered, ranging from 0% to 83,33% for individual decisions; and totally 70,59% in the API client/API context and 75.00% in the operation context. For the necessary criteria assessments *ncri* improvements are even higher, ranging from 50% to 97,73% for individual decisions; and totally 90,16% in the API client/API context and 93,75% in the operation context. Here this high improvement is mainly due to the pre-selected criteria, which lead to criteria assessments in whole sets of criteria rather than evaluating each criterion separately. Finally, for the number of possible decision outcomes *ndo*, the improvement in uncertainty reduction for individual decisions ranges from 0% to 81,25%. The large spread is due to the fact that without our ADD model, the number of options rises exponentially: For decisions with larger numbers of decision options the improvement is greater than for those with only a few options. In total we see a 25% improvement in the operation context, as this is just a single decision.

Table 2. Uncertainty reduction estimation

Decision		# Decision nodes *ndec*	# Criteria assessments *ncri*	# Possible decision outcomes *ndo*
Client Identification and Authentication Decision	With design space	1	1	5
	Without design space	4	44	16
	Uncertainty reduction	75,00%	97,73%	68,75%
Perform Error Handling	With design space	1	1	4
	Without design space	2	18	4
	Uncertainty reduction	50,00%	94,44%	0,00%
Preventing API Clients from Excessive API Usage	With design space	1	4	2
	Without design space	1	8	2
	Uncertainty reduction	0,00%	50,00%	0,00%
Metering and Charging for API Consumption	With design space	1	4	12
	Without design space	6	24	64
	Uncertainty reduction	83,33%	83,33%	81,25%
Explicit Specification of Quality Objectives and Penalties	With design space	1	2	5
	Without design space	4	28	16
	Uncertainty reduction	75,00%	92,86%	68,75%
Total in Context API Client / API	With design space	5	12	268435456
	Without design space	17	122	482754917909
	Uncertainty reduction	70,59%	90,16%	99,94%
Avoid Unnecessary Data Transfer = Total in Context Operation	With design space	1	2	12
	Without design space	4	32	16
	Uncertainty reduction	75,00%	93,75%	25,00%

The total for the API client/API context shows a 99,94% improvement; here we use for both cases the same exponential function for calculation, but as individual decisions were performing much better with our model, the resulting total number is much lower than without it.

Please note that the numbers are rough estimates only, not a formal evaluation. They indicate that substantial uncertainty reduction is possible. To harden them, further such estimations in other design spaces are required, which could be the basis for developing a theory. Such a theory could then be validated in empirical studies in realistic cases.

6 Discussion and Threats to Validity

We have studied knowledge on established practices on API quality aspects, relations among those practices, and decision drivers to answer **RQ1** with multiple iterations of open coding, axial coding, and constant comparison to first codify the knowledge in informal codes and then in a reusable ADD model. Some of our decision options were design patterns (documented in [19] and designated as such in our models, see Figs. 2, 3, 4, 5, 6 and 7). Precise impacts on decision drivers of design solutions and their combinations were documented as well; for space reasons we only summarized those in the text and did not show them in the UML models (see [19] for technical details).

The contributions to RQ1 in part already answer **RQ2**, in so far as each of the pieces of knowledge is systematically derived from established knowledge, which helps to reduce uncertainty regarding finding knowledge at all and finding it correctly. In addition, we estimated the uncertainty reduction achieved through the organization of knowledge in our ADD model in Sect. 5. We may conclude that our ADD model (and similar models) have the potential to lead to substantial improvements in uncertainty reduction in all evaluation variables due to the additional organization it provides and pre-selections it makes. For individual decisions, mastering and keeping in short term memory the necessary knowledge for design decision making seems infeasible without the ADD model (e.g., four decision nodes with 44 criteria assessments and 16 possible outcomes for the first decision in Table 2), but quite feasible with our ADD model. Our model also helps to maintain an overview of the decisions $ndec^{\oplus}$ and criteria assessments $ncri^{\oplus}$ in the combined API client/API context. Only the number of possible decision outcomes for the combination of multiple decisions seem challenging to handle, both in the $ndo^{\oplus}$ and $ndo^{\ominus}$ case. That is, despite all benefits of our approach, the uncertainty estimations also show that a limitation of the approach is that when multiple decisions need to be combined in a context, maintaining an overview of possible outcomes and their impacts remains a challenge – even when a substantial uncertainty reduction and guidance is provided as in our ADD model. Further research and tool support is needed to address this challenge. As our numbers are only rough estimates, further research is needed to harden them and confirm them in empirical studies, possibly based on a theory developed based on such preliminary estimations.

While generalizability beyond the 55 knowledge sources we studied is possible to a large extent, our results are limited to those sources and to a lesser extent to very similar APIs. Most of the 55 source were public, Internet-wide APIs; we have studied a few in-house APIs as well. This mix might have introduced bias or left to the omission of important in-house practices in commercial enterprises. We could only study the API quality aspects addressed in those sources. Thus, we do not claim any form of completeness. Our results are only valid in our set scope. In the various coding process and review stages of our research method, each finding was checked in at least five iterations by different members of our author team. However, possible misinterpretations or biases of individual researchers or the whole author team cannot be fully excluded and might have influenced our results. As the authors have many years of experience in the field (gained both in industrial projects and in education of students and practitioners), we are optimistic that this threat to validity is mitigated in our study to a large extent.

7 Conclusions

We have performed a qualitative study in which we have studied microservice API quality aspects in 55 unique sources. Our study led to the identification of six architectural design decisions with in total 40 decision options and in total 47 decision drivers modelled in a formal ADD model. In our uncertainty reduction estimations we were able to indicate that the knowledge organization in our ADD model can lead to a significant reduction of uncertainty where multiple decisions need to be combined. In our future work, we plan to combine our ADD model with other aspects of API design, apply the results in case studies e.g. in different verticals or industries, and build and empirically validate a theory based on the preliminary uncertainty reduction estimations.

Acknowledgements. This work was partially supported by Austrian Science Fund (FWF) project ADDCompliance and FFG project DECO (no. 864707).

References

1. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016)
2. Charmaz, K.: Constructing Grounded Theory. Sage, Thousand Oaks (2014)
3. Coplien, J.: Software Patterns: Management Briefings. SIGS, New York (1996)
4. Glaser, B.G., Strauss, A.L.: The Discovery of Grounded Theory: Strategies for Qualitative Research. Aldine de Gruyter, New York (1967)
5. Gorton, I., Klein, J., Nurgaliev, A.: Architecture knowledge for evaluating scalable databases. In: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015), pp. 95–104 (2015)
6. Gupta, A.: Microservice design patterns (2017). http://blog.arungupta.me/microservice-design-patterns/
7. Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O.: Service cutter: a systematic approach to service decomposition. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) ESOCC 2016. LNCS, vol. 9846, pp. 185–200. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44482-6_12
8. Hassan, S., Bahsoon, R.: Microservices and their design trade-offs: a self-adaptive roadmap. In: Proceedings of the SCC, pp. 813–818 (2016)
9. van Heesch, U., Avgeriou, P., Hilliard, R.: A documentation framework for architecture decisions. J. Syst. Softw. **85**(4), 795–820 (2012)
10. Hentrich, C., Zdun, U., Hlupic, V., Dotsika, F.: An approach for pattern mining through grounded theory techniques and its applications to process-driven SOA patterns. In: Proceedings of the 18th European Conference on Pattern Languages of Program, pp. 9:1–9:16 (2015)
11. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term, March 2004. http://martinfowler.com/articles/microservices.html
12. Lytra, I., Sobernig, S., Zdun, U.: Architectural decision making for service-based platform integration: a qualitative multi-method study. In: Proceedings of WICSA/ECSA (2012)
13. Menascé, D.A.: QoS issues in web services. IEEE Internet Comput. **6**(6), 72–75 (2002)

14. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly, Sebastopol (2015)
15. Pahl, C., Jamshidi, P.: Microservices: a systematic mapping study. In: 6th International Conference on Cloud Computing and Services Science, pp. 137–146 (2016)
16. Pautasso, C., Zimmermann, O., Leymann, F.: RESTful web services vs. big web services: making the right architectural decision. In: Proceedings of the 17th World Wide Web Conference (WWW), pp. 805–814, April 2008
17. Richardson, C.: A pattern language for microservices (2017). http://microservices.io/patterns/index.html
18. Rosenberg, F., Celikovic, P., Michlmayr, A., Leitner, P., Dustdar, S.: An end-to-end approach for QoS-aware service composition. In: IEEE International Conference on Enterprise Distributed Object Computing Conference (EDOC 2009), pp. 151–160. IEEE (2009)
19. Stocker, M., Zimmermann, O., Lübke, D., Zdun, U., Pautasso, C.: Interface quality patterns - crafting and consuming message-based remote APIs. In: Proceedings of the 23nd European Conference on Pattern Languages of Programs, EuroPLoP 2018 (2018)
20. Trubiani, C., Ghabi, A., Egyed, A.: Exploiting traceability uncertainty between software architectural models and performance analysis results. In: Weyns, D., Mirandola, R., Crnkovic, I. (eds.) ECSA 2015. LNCS, vol. 9278, pp. 305–321. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23727-5_26
21. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: Proceedings of the 12th International Conference on World Wide Web, pp. 411–421. ACM (2003)
22. Zimmermann, O.: Microservices tenets. Comput. Sci. - Res. Dev. **32**(3), 301–310 (2017)
23. Zimmermann, O., Koehler, J., Leymann, F., Polley, R., Schuster, N.: Managing architectural decision models with dependency relations, integrity constraints, and production rules. J. Syst. Softw. **82**(8), 1249–1267 (2009)
24. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U.: Interface representation patterns: crafting and consuming message-based remote APIs. In: Proceedings of the 22nd European Conference on Pattern Languages of Programs, EuroPLoP 2017 (2017)