A Systematic Evaluation of Microservice Architectures Resulting from Domain-Driven and Dataflow-Driven Decomposition

ILIE SEBASTIAN MIHAI, University of Twente, The Netherlands

At some point in their lifecycle, monolithic applications can reach a threshold where their continuous deployment, integration, and scalability processes become problematic to handle. Tackling this, the Microservice Architecture (MSA) is advocated to compile individually executable services, each of whom is distinctively deployed and serves a unique functional segment of the system. This paper addresses two of the most general and systematically applicable decomposition techniques, namely Domain-Driven Design (DDD) and Dataflow-Driven Development (DFD), which we have used to break down a medium-sized, actively maintained monolithic application. The most suitable metrics derived from relevant literature, and suitable for DDD and DFD, have been compiled and assessed. A metric-based comparison of the two approaches has been performed, evaluating each decomposition technique individually. The aim of this paper is to assist architects in further understanding the general applicability of the aforementioned decomposition methods, given the limited number of comparative studies between the two, and the plethora of evaluation concerns arising from domain-specific influence factors.

Additional Key Words and Phrases: Microservices, Domain, Dataflow, Architecture, Event, Monolith, Destructuring, Metric, Evaluation

1 INTRODUCTION

Microservice Architectures (MSA) is a concept introduced in 2014, advocated to tackle the concerns of legacy systems regarding rising complexity, high dependency, and high coupling. This type of architecture directly targets the limitations of monolithic applications, by providing a unique way of service-oriented structural design, which splits the functionality of a system into microservices [1, 31]. The interaction between microservices can be achieved via "lightweight" processes, usually decentralized or via brokers, following rule-based communication interfaces [1, 26]. Since the introduction of MSA, plenty of structural algorithmic ways of achieving microservice-based architectures have been presented. However, choosing the most suitable decomposition method for an application is a complex task, as the business models and functions for each such system can highly differ, thus interfering with the destructuring processes, which can lead to biased evaluation outcomes or unsuitable architectural designs [14, 31, 41, 43]. Approaches to handle this concern had also followed, yet it was shown that extensive inquiries regarding such domain-related criteria do not invariably offer better results [4, 6]. Our literature analysis concluded that the most suitable and standardized decomposition techniques, given the highly extensive profile of this paper, are Domain-Driven Design (DDD) and Dataflow-Driven Development (DFD). With regards to a direct comparison, based on qualitative metrics between DDD and DFD, there are no systematic, comparative studies between the two [34, 41]. The goal of this paper is to fill in this gap by providing a thorough analysis of the two techniques that could further help designers understand their applicability. The research questions that were answered in this paper are:

- What set of qualitative metrics is suitable for evaluating microservice-based architectures, and out of these, which are most suitable for DDD and DFD?
- How do DDD and DFD compare to each other with regard to their relevant set of qualitative metrics?

To answer the first question, our analysis consisted of a literature review, which resulted in a set of qualitative criteria applicable to DDD and DFD. The second question is answered based on the individual evaluation of each architecture, with regard to the compiled metric-based evaluations. The diagrams, although relevant only for the context of the paper, can be used as a tool to visualize the rationale of decomposition techniques and benefits advocated by MSA.

The following sections are organized as follows: Section 2 presents the common metric-based concerns regarding static analysis and the drawbacks of runtime-dependent evaluation criteria, by means of literature analysis. Section 3 briefly addresses the background and concerns that current monolith decomposition techniques encounter. Section 4 presents an analysis of DDD and DFD techniques, as well as considerations for the most appropriate representatives for each, by briefly assessing criteria such as domain boundaries, granularity, and modularity. Section 5 presents the models we considered in the research steps, such as the domain and dataflow concerns of DDD and DFD, selection aspects for the appropriate monolith candidate, and influencing factors that can lead to biased analysis. Section 6 presents the extracted set of metrics, alongside their related evaluation procedure steps, and addresses their applicability for DDD and DFD. Section 7 discusses the processes and steps that were followed to destructure the monolith and describes the yielded diagrams. Section 8 refers to the results of Section 7 to evaluate each resulting architecture by means of the compiled metric set. During such a process, it objectively addresses contextual gaps and drawbacks of both architectures, while assessing the results of the metrics.

2 RELATED WORK

The benchmarking processes presented by Bjørndal et. al [5] resemble, to a great extent, the rationale of our paper. In this work, a systematic metric study was presented, gathered by means of a literature review, and then assessed by designers. Although presenting promising results, the decomposition process is not detailed, and the environment on which the metrics were benchmarked is highly dependent on cloud-based deployment aspects such as pods or CPU allocation, whose set-up processes were proven problematic to handle. A similar approach is followed by Taibi et. al, whose paper discusses similar metrics, with the rationale of function-based

TScIT 37, July 7, 2023,

analysis. However, the data collected to evaluate such metrics are runtime-based, which can be problematic to set up and whose environment specifications are hard to objectively evaluate in different contexts [45].

Another paper, presented by Vera-Rivera et. al [47] follows the same decomposition process and assesses the result, given a set of metrics similar to ours. The "backlog" presented in the paper is said to be compiled from a literature review, yet the queries or methodologies of such review are not thoroughly detailed. The granularity specification is said to be "intelligent" and follows a machine-learning-based model. The decomposition processes specify DDD and Service Cutter, yet the latter approach has a considerable outlier: one microservice contains 10 services, while the previous one, has only 2. This may yield biased results, given the vector-based metric evaluation methodology of the paper.

Another broadly applicable framework, proposed to "align with industry requirements" [14], presents a similar set of metrics. The selection is backed by a broad survey, yet the contextual use is intended for "semi-automatic decomposition" techniques. The paper continues to contextualize DDD and DFD, and although most of the metrics are said to be suitable for both, domain use and limitations are not mentioned. Some metrics, such as granularity, are evaluated given LoC, a criterion that was proven to be rather "informative" than reliable. Furthermore, other metrics are evaluated without using proper weights for their parameters, such as "number of operations".

3 DECOMPOSITION TECHNIQUES BACKGROUND

This section presents the wide range of decomposition alternatives that have already been proposed [43], with a systematic search for "Service discovery" and "Data management" techniques, which are the two most recurrent approaches to identifying microservice candidates [42].

Richardson et al. [38] introduced a novel concept of decomposing monoliths based on "business capabilities" which was later proven [21, 32] to need outsourced involvement, and therefore a systematic approach cannot be followed. A decomposition method proposed by Baresi et al. [4] presents a monolith system decomposition method using "API specification analysis". However, this approach needs clearly determined boundaries with regard to the interfaces used [21, 32]. Another novel proposal uses transactional contexts [35], a principle that advocates against using domain interdependency factors to decompose a monolith, and instead promotes the "aggregation of domain entities". However, the analysis assumes explicit "business logic" distinction and a clear predefined assessment of its implications in the decomposition process, aspects that were found crucial in our metric analysis. One algorithmic process, proposed by Zhamak Dehghani et al. [16], provides a set of steps to decouple elements of the business context, thus minimizing dependencies. However, this was proven to be inefficient, lacked the measurement steps, and the decomposition process was found tedious and complex. [29, 41]

Out of all the popular techniques, including the previous ones, it is advocated that Domain-Driven Design is the most common and objective design choice for the decomposition of processes [14, 46]. DDD decomposition yields modular microservice candidates, with a high degree of generality, thus making it suitable for a wide range of applications [20, 23, 27]. Dataflow-Driven Development (DFD) yields microservice candidates with a high degree of granularity (granular, "rational, and understandable" candidates), as opposed to other data-related decomposition methods, which involve database segmentation steps, dependency graphs, or facade adaptations [28]. Both DDD and DFD are highly popular decomposition techniques, preferred by many designers given their wide applicability range [20, 27, 34] or granular resulting candidates. For these reasons, they represent suitable representatives for the context of this paper and will be further detailed in the next section.

4 DECOMPOSITION TECHNIQUES

4.1 DDD techniques

First introduced by Evans in 2003 in his book "Domain Driven Design" [18], the method encapsulates "organizational units" into "domain contexts". This laid down the foundation of the Microservice architectures later on, as the concept of "bounded contexts" can easily compile service processes and interfaces, which fit MSA. The boundaries of the business functions are clearly extracted and the bounded contexts are in direct relation to the domain models of the organization [14, 18, 48]. One technique that leverages the concepts of DDD is proposed by Brito et. al [10] and makes use of Topic Modelling techniques. The threats to the validity of this approach were considerable, as the reliability of the used measures was not checked for correctness, and the scope of the approach was narrowed down to a too-small list of applications. Another DDD-related approach was proposed by Jin et al. [27], whose performance and reliability metrics were not considered, and instead, the approach focused on leveraging "functionality and evolvability" [23, 27]. Another notorious DDD application is Context Choreography Domain-Driven Design [24]. It was however stated that the implementation is based upon DDD patterns and UML profiles [37]. The most representative DDD method within the context of this paper appears to be the UML Profile proposed by Rademacher et al. [37]. This method introduces UML design choices into the microservice architecture context. It further proposes semantics and domain choices with regard to the UML profile for Domain-driven MSA Modelling (DDMM), while making use of already-existing contributions in the area of UML modeling and its characteristics/design units. This approach provides a solid and general ground base for domain model representations and is highly suitable given its "selective abstraction of conceptual knowledge" approach and concepts that leverage Ubiquitous Language to join "domain experts and technologists together" [18].

4.2 DFD techniques

Leveraging the design choices of DFD [13], a "semi-automatic decomposition" technique was proposed [32], introducing the novel concept of "process-datastore" for DFD (DFDPS) that oversees the business functions. The authors demonstrated that the method introduced by Chen et al. is indeed efficient, and validates the high granularity of the resulting microservice candidates. However, the main drawback is that DFDPS is useful only if the system architecture is so cluttered that it cannot be contained within a single resulting DFD. Another DFD approach was proposed by Taibi et al. [44], who introduced a decomposition framework that leverages the concept of "process mining". One concern with regard to this method is that the tool used to evaluate the effectiveness could not be proven to yield objective and unbiased results [2]. The most representative DFD method is the "Purified/Decomposed DFD" approach, proposed by Chen et al. [13]. This paper introduces rules and patterns that have been proven to significantly reduce the complexity of decomposition processes, based on constructing and evaluating "purified" and "decomposed" dataflow diagrams, while the resulting microservice architectures consist of "rational, and understandable" candidates [21, 28, 32].

5 RESEARCH METHODOLOGY

The first step of the research was to find appropriate decomposition methods. Our search aimed at finding highly applicable, generalizable, and popular approaches, that would not greatly influence the impact of very specific domain models on the final analysis. Several alternatives were analyzed, and DFD and DFD were selected. For each of these approaches, the most representative technique was chosen, given the same criteria, and a brief analysis of each technique was conducted. Techniques that were highly dependent on database communication, highly dataflow dependent (which could only be analyzed dynamically), or whose results were highly dependent on the domain models have been ignored. Following this, the monolith was chosen using the following criteria:

(1) Domain considerations: the structure of the chosen monolith had to contain a clear overview of its domain models. The separation of concerns should have clearly defined boundaries, thus an MVC architecture was regarded as highly appropriate.
(2) Dataflow considerations: the flow of data throughout the components of the system needed to be easy to identify, in order to gain a systematic overview needed for later destructuring processes.
(3) The number of files & commits: These 2 criteria (and the LoC) of each application were analyzed and only applications with 1000+ commits were considered. For these, we looked for a proportional number of files with a (median) factor of around 40 times the number of commits.

Given these three factors, a set of 14 applications were selected as suitable candidates for the monolith. Lower outliers had approximately 24000 commits & 100 files, whereas the highest one had around 275000 commits & 2000 files. A median candidate, which also matched the domain and dataflow considerations, was chosen to be a popular library called Healthchecks [22]. It has a Django-based MVC architecture for cron monitoring tools, with scheduled tasks, a web dashboard, API integrations, authentication, and team management features, with a REST-based communication protocol.

The following step was to select appropriate metrics for comparing DDD and DFD. This stage involved analyzing pre-existing case studies and evaluating their results. The chosen metrics had to already be evaluated for effectiveness, with a strong pre-existing technical ground. The main criterion was that the case study results had to be applicable to both DDD and DFD. Moreover, the analysis considering these metrics should be doable without extensive help of tools or highly dynamic analysis, evaluating runtime data, or anything related to deployment/integration, such as PaaS or orchestration mechanisms. In order to yield an unbiased result, the analysis of such metrics requires a complex environment and careful resource allocation [5], aspects that could not be objectively assessed due to time limitations.

The next step was to decompose the chosen monolith using the DDD and DFD techniques. Initially, the domain models and the source code of the application were analyzed. Afterwards, the two chosen techniques were followed step by step, the appropriate design considerations were applied, and multiple diagrams representing the resulting architectures were produced. Having compiled all the results up to this point, each chosen metric was analyzed, alongside their applicability degree and domain considerations. The analysis focused on qualitative evaluation, and no runtime data or orchestration elements were considered.

6 METRICS

The criteria that are chosen with respect to metric evaluation follow an objective and rational structure (no relative, tool-based, or highly dynamic attributes). Most of the existing criteria in microservice architecture case studies (as well as automated tools) rely on the runtime quantitative data [4, 7, 33] and some are proven to be highly dependent on the environment, and thus hardly generalizable.

6.1 Granularity

The granularity degree of a service can be perceived as a "trade-off between size and number of microservices" [4]. Determining the size of service is a complex task and the current literature proposes a wide range of, sometimes, contradicting criteria [9, 12, 33]. One proposed model is called "MM4S for maintainability considerations" [9], which sees criteria in a pyramid-like hierarchy, where just below maintainability are “service properties”. It also proposes a granularity metric called Weighted Service Interface Count (WSIC) which accounts for "the number of exposed interface operations of service S" and applies weights to each of them depending on their number of arguments [9], while also taking implementation into consideration. The exact weight assignment methodology is not unanimously agreed upon and is up to the designer, and additional papers suggest that besides the number of parameters, the granularity of such operations also needs to be considered [15]. Extending on the WSIC definition, Bogner et. al [8] propose that in order to find an unacceptable (too large) granularity level, each service’s WSIC should be compared to the general WSIC of the architecture, using Formula (1) for WSIC.

$$WSIC_{AVG}(Y) = \frac{\sum_{S \in Y} WSIC(S)}{|Y|} \tag{1}$$

Another size-related metric, introduced again by Bogner et. al, is "Total Response for Service (TRS)" [8], where RFO is the number of operations that can be called in response to an incoming request for operation O, given its specific interface. Formula (2) is the general TRS formula presented in Bogner et. al [8].

$$TRS(S) = \sum_{O \in SI_S} RFO(O) \tag{2}$$

A literature review [36] mentions another set of popular criteria to evaluate Size and Complexity: for Size, they mention "number of synchronous cycles", "distribution of synchronous calls" or "average size of asynchronous messages"[17]. Moreover, size itself is said to be more helpful in finding outliers across candidates, instead of being a qualitative metric per-se [14], as there is no unanimously agreed range. One of the most popular quantitative evaluation metrics is Lines of Code (LoC) [30], which builds upon granularity. However, this metric is not objective since it is conditioned by coding styles, scaffolding, boilerplate code, or programming language [14].

6.2 Database Connectivity

Monoliths that are highly reliant on data flow might yield complex issues with regard to data connectivity and consistency [11]. Two suitable methods proposed are CQRS (Command Query Responsibility Segregation), and "Role Separation (RS)", used to separate databases and redirect read/write operations to core and replica database instances, based on the "actor use of available [REST] operations" [11], where "queries can be efficiently performed" [39].

6.3 Event Sourcing and Async Messaging

Event sourcing is a pattern [39] that can leverage responsibility segregation by chaining the states (of objects, entities) as a "sequence of events". One solution mentioned in the literature is "Gateway Aggregation" [39], which creates aggregated access calls. When it comes to messaging, Faustino et. al proposes that the size of the event should increase as much as the cache permits, and the developers need to carefully consider the threshold such that the data is actually used [19]. Furthermore, in cases where calls affect multiple services, services cannot hold "ACID properties between them". Another suitable metric is the "number of interface calls" and its corresponding database-sharing optimizations.

6.4 Structural Coupling

Suitable coupling metrics are the "number of clients that invoke at least one operation to the service", "the number of other services that a service depends on" or "the number of service pairs bidirectionally dependent on each other" [40]. Other metrics are [9] Absolute Dependence of the Service (ADS), and Services Interdependence of the System (SIY) or Absolute Importance of the Service (AIS) [8]. Automatic validation frameworks tend to only count "outward dependencies" for an emergent candidate, in order to form the dependency graph without the risk of cyclic dependencies [15]. The result of SCC should be null, and ideally, each service should present one outward link and no cycles [15]. The same framework also proposes cohesion-degree criteria such as evenly distributed methods (without duplicated entities), the "responsibilities composition" (all services should have a similar weighted purpose, and advocates for a "single responsibilities principle"), or "semantic similarity", which advocates that all methods/classes should be functionally related to each other [15].

6.5 Cohesion

Given its "semantic" implications, literature review [14, 36] showed that the most frequent metrics for Cohesion are "the similarity of the parameters data types", and "used operations per client". Another study [9] showed the following metrics are relevant: Service Interface Data Cohesion (SIDC) (operation's similarity of passed data, resulted from dividing the number of “common data types” by the number of "discrete data types", whose result should preferably be 1), or Service Interface User Cohesion (SIUC), based on the user similarity actions, whose value is calculated by the number of "operation calls" made by users divided by total call possibilities [8, 9]. An additional metric is "Total Service Interface Cohesion (TSIC)" [8], which is the weighted system average of SIDC and SIUC, and whose purpose is to be compared against all individual services.

6.6 Dynamic Similarity Index

Andrade et. al defines similarity as the "distance between domain entities", where the dependency tree between bounded contexts is outlined using graph theory elements [3]. Designers can compile similar domain entities within the same bounded context, should their similarity measurement be low, which can highly reduce the number of interface calls between services. Such similarity index is based on the number of "domain entities that are frequently accessed in sequence". Formula (3) presents the "sequence similarity measure" between two entities e1 and e2, proposed by Andrade et. al [3], where "**sumPairs**" represents the "number of consecutive accesses" between the two entities, and "**maxPairs**", the longest sequence of graph calls for the same entities.

$$sm_{sequence}(e1, e2) = \frac{sumPairs(e1, e2)}{maxPairs} \tag{3}$$

6.7 Evolvability

Sometimes considered a sub-criteria of maintainability, "evolvability" is referred to as the "degree of effectiveness and efficiency with which a system can be adapted or extended" [7]. The most common characteristics are "very decentralized [...] very autonomous teams vs centralized governance" and "varying degree of team autonomy" [7]. The survey of [7] further suggests that designers should also account for test coverage values, yet its results can be problematic as one can find a workaround to surpass linting settings/use fake coverage.

7 MONOLITH DESTRUCTURING

7.1 DDD Destructuring Process

Following the paper of Rademacher et al. [37], the following DDD destructuring processes revolve around domain models and the concept of "Bounded Context", which encapsulates each emergent microservice. The textbook definition of the process, as the paper has it, is leveraging "informal UML class diagrams to express domain models". An initial UML class diagram was constructed, based on the classes in the models, database schema, and the source code of the application. Figure (1) presents the microservices architecture resulting from DDD decomposition, by means of an ER diagram, providing a detailed visualization of the implemented concepts. First, the domain considerations are outlined. This defines boundaries to separate the result into domain models, which were used to define a "ubiquitous" language to leverage the initial UML elements and provide a common communication ground. The following constraints were applied, based on the constraint specification methodology and notation presented in the paper [37]: AggregatePart C1 and C2 for [accounts-project, accounts-credential, auth-group, and api-ping], AggregatePart C3 for [api-ping, api-notification], C4 for [api-ping, api-notification, auth-group]. AggregateRoot C5 and C6 for [auth-user, api-check, and accounts-project]. Entity C7 for all entities. Repository C10 to [Channel, Group, and Project repositories]. Service C11 and C12 to all services. Spec C14 to [Payment and Notification Specs], Spec C15 to all specs. SideEffectFree C21 to Payment validation function. BoundedContext C25 to Auth context. As paper [37] mentions, one has to define proper "shared model interfaces" between bounded contexts, denoted by Value Objects (User Shared, Project Shared).

The decomposition process resulted in an architecture with three services: Accounts, Auth and API. The communication between them can be accomplished using REST interface calls, which are needed to expose entities "as instances of" Value Objects [37] (in our context, User Shared and Project Shared). Furthermore, as mentioned in Section 8.2, each microservice can incorporate a separate database, consisting of "all parts of the domain model". Horizontal scalability is easily achievable for the resulting architecture, as bounded contexts can be developed, deployed, and replicated independently, and the "distribution of synchronous requests provided by the exposed interfaces" [14] can be handled effectively, given the clearly outlined REST interfaces.

7.2 DFD Destructuring Process

Initially, a traditional DFD diagram (figure 2) was created. The application source code was checked, and based on the resulting logging, the processes were determined. Tthe external entities and data stores were determined, given their link to the processes resulting from the call graph tree. The traditional diagram's purpose is to showcase the data flow from a "business function" perspective [13]. The main difference between DFD and DDD is that DFD only uses processes that "manipulate data" and therefore, elements such as "auth-group-permissions" which do not manipulate data are not taken into account as in DDD. Afterward, the purified DFD (whose purpose is to hide the information like "data store and external entities") is used to construct the decomposable version "DDF" (figure 3), by "combining the same operations with the same type of output data" [13]. This layer is said to improve maintainability, metric that was previously found crucial in the literature. Finally, the candidates (so-called modules) are selected, by aggregating "individual modules of operation and its output data" [13]. API checks operations were abstracted into a facade that encapsulated all types, given that they have related output data. Although the service-finding algorithm suggests that each operation resulting after applying the decomposed DFD rules should be a separate service alongside its relevant data, the paper mentions that the designers need to consider their contextual use as well.

The decomposed architecture consists of five services: Auth, Payment, Project, User, and API. The communication between them can be asynchronous, using a central buffer, or by means of gateway aggregation [19, 39]. Data Stores, which are repositories "of data manipulated by operations" [13], can be used as databases for the resulted microservices architecture, optimized by introducing database replicas with different interfaces for the User and Admin agents.

8 RESULTS

8.1 Granularity

As the DDD and DFD-based decompositions result in different numbers of services, we calculated the average WSIC instead of the individual WSIC, so as to not bias the results. Empty parameters and constructor methods were not considered, as the weight cannot be accurately estimated. Tests were also excluded, as the authorization service needs extensive testing, thus unbalancing the result. Each parameter gets the weight of one. Table (1) gives an overview of the method count (including decorators and Django-specific methods) based on the parameter size, for each service resulting from DDD decomposition.

Service	0/1 parameter	2 parameters	3 parameters
Accounts	38	33	24
Auth	33	47	21
API	152	44	26

Table 1. The number of service methods based on parameter size, for the microservices architecture resulted from DDD decomposition

$$WSIC_{avg}(Y) = \frac{38 * 1 + 33 * 2 + 24 * 3 + ... + 44 * 2 + 26 * 3}{3} = 228 \tag{4}$$

We see that the Accounts (190 WSIC) and Auth (176 WSIC) services are close to each other. However, the API service is an outlier. The methods within API are more concerned with external integration, thus a lot of boilerplate and external integration needs to be set up. Although non-granular, the weights can be deceiving as that data is not all transported through the other interfaces, and thus the async events' data structure size is smaller.

With regards to DFD, Table (2) presents the service method count, for each service, based on parameter size:

Service	0/1 parameter	2 parameters	3 parameters
Auth	25	35	17
User	38	30	21
Payment	8	12	4
Project	29	12	7
API	123	33	22

Table 2. The number of service methods based on parameter size, for the microservices architecture resulted from DFD decomposition

Again, the Auth and User services have similar WSIC. The API management is broken down, as now the focus is on the data flow. The outlier introduced is Payment management. The payment and

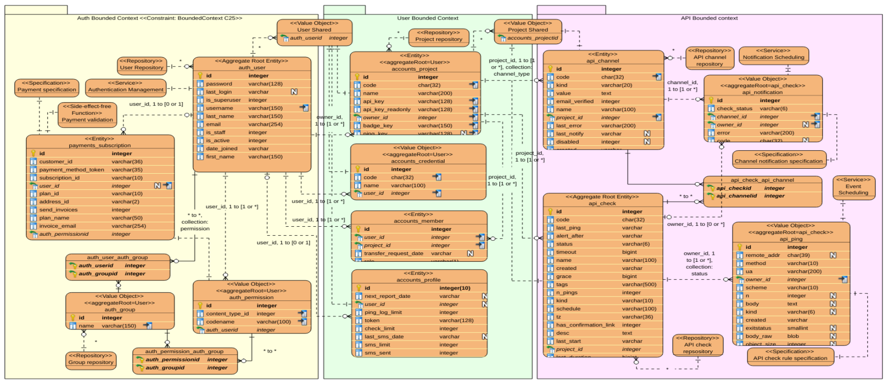
Fig. 1. Final ERP diagram for the microservices architecture resulted from DDD decomposition

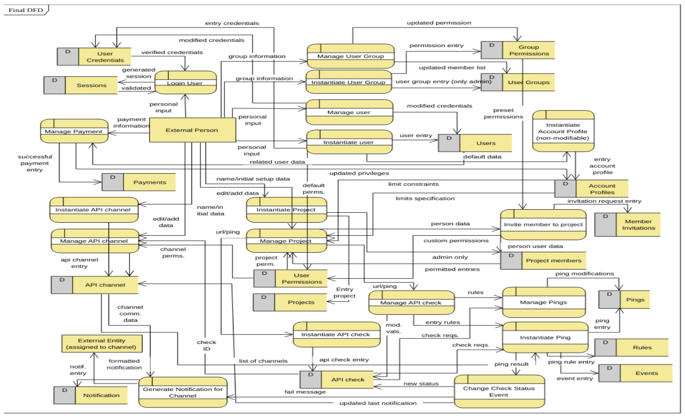
Fig. 2. Traditional DFD diagram of the original monolith

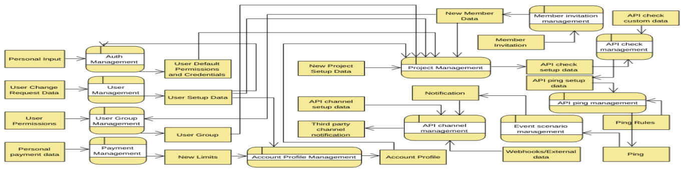
Fig. 3. Final decomposed DFD diagram of the microservices architecture after the rules have been applied

member registration operations were separated for the Auth Management service and User Management service respectively.

When it comes to TRS, DDD is considerably more granular. As seen from the diagrams, the relatively small number of interface interactions indicate low coupling. For example, between User and API services, the connection is done via two projects and channel metadata. The shared Value Object of project data can be designed to yield as little information as possible for the channel, which can now be set up independently on the project dashboard, while all API processing is done separately of Account context. This is not the case for DFD however, as the interface needs higher complexity (and thus higher TRS) to ensure data consistency. However, the WSIC index has a better value for DFD. With regard to the other metrics, the number of interfaces is smaller for DDD. However, in the DFD case, the Project service is separated, thus the distribution and size of asynchronous calls can be spread out more efficiently. Our DDD-resulted architecture is "the core basis for all granularity-based decisions" [25], and as every service is meant to be encapsulated within a bounded context, while the granularity is automatically based on domain "functionality" and not on other metrics such as LoC [25].

8.2 Database Connectivity

In our resulting DDD architecture, a designer can store "all parts of the domain model [...] in the same database" [19], such that the "need to share databases between microservices" decreases [39], and all operations/function calls between the entities can be done via interface calls. Such interface calls can be minimized by correlating a "database access index" to "domain entities exported to other modules" [19]. Moreover, the bidirectionally related services are now combined, as the literature suggests [40]. Such a step removed the old monolithic interdependencies between entities yet added "indirect relations between databases" [19]. For the resulting DFD architecture, optimized replicas were in front of the interface for faster accessibility. If applied at the "functional level", this method facilitates scalability and avoids maintainability issues. In order to maintain consistency across calls, the write calls of the interface should be the last in the chain [19]. This is especially relevant when such a state is being deserialized and validated, and as Faustino et. al point out, states can be consistent and yet still yield deserialization errors. The deserialization step is however more problematic for DFD, as opposed to DDD, as the data state has to be made consistent. A big factor in this problem is the nature of the DFD architecture, given the high flow of pings, API checks, and async notifications, especially in the API service. Thus, a "centralized service" can be hard to attain, making the CQRS metric more suitable for DDD instead. Different from RS, the read accesses are all redirected toward the replica database, optimized for such operations. However, one drawback could be inconsistent accesses or deadlocks, and such a method requires low coupling systems with high data consistency. In order to handle this, our DFD implementation would need a centralized separate service for updating the optimized replicas, which introduces overhead [39]. As an optimization aspect, in our context of DDD, the architecture is configured to have different request contexts (user, admin) separated, and a unique interface is exposed for each. Replicas of such databases can thus be created, cached, and optimized for each set of operations, such as put/delete accesses for admins, with prioritized and more efficient access calls [39]. This technique was proven to provide better results for RS [19].

8.3 Event Sourcing and Async Messaging

When it comes to event sourcing, decoupling databases using chained events in the context of DFD was shown to improve database access times and lower the number of interfaces calls [19, 39]. However, this poses the problem of data consistency, already mentioned before. Our resulting DFD architecture can, however, introduce asynchronous messaging between components using a broker and a central buffer [39]. The downside would be the overhead added by tools such as Kafka or using MACH-related worker thread pools. However, such an approach needs to account for sender/receiver identification and data deserialization [19]. On the other hand, an advantage of our resulting DDD architecture is that there is no need for gateway aggregation, given the principles of shared Value Objects between bounded contexts. However, for DFD, such a gateway (usually for Network Address Translation) introduces security and availability overheads, and eventual complications of reverse proxies.

In event-driven architectures, the event data is "transferred in the form of Data Transfer Objects (DTO)" [19], which is generally encoded in a JSON format, and which should hold as little information about sender and receiver microservices as possible. The Event Sourcing pattern argues that the "state of a business entity is persisted as a sequence of events" [39]. In order to optimize interconnectivity, one needs to think about the trade-off between a number of interface calls and the actual payload size that this sequence of events carries, yet as it was previously mentioned, this can sometimes be deceiving for DDD (API bounded context).

8.4 Structural Coupling

When it comes to AIS (and its "sibling metric" ADS [8]), the number of clients calling methods on the service interface is significantly lower in the DDD approach, as previously mentioned. Between the bounded contexts, the User is connected to the Accounts entities only via user-id and owner-id, and the User Shared (Value Object) instance. On the other hand, the rules applied to the decomposed DFD ensure that "operation and its output data" modules promote individual service workflow while having a low dependency degree between each other [13], therefore enforcing loose coupling.

When it comes to SIY, by design, neither architecture has strongly connected components, and thus no bidirectional dependency. When it comes to semantic similarity, DFD yields significantly better results. During the decomposition process, the modules are extracted by an "operation and its output data" identification mechanism, leveraged by the decomposition rules. As already mentioned, this ensures a high cohesion degree, and combined with the $WSIC_{AVG}$ results for the services resulting from DFD decomposition, it can be seen that the methods within such modules are more functionally related to each other, as opposed to the case of DDD decomposition. However, responsibility distribution tends to be lacking in some places of DFD architecture. For example, the Payment service has a lower workload than the rest. The others, as the WSIC metric suggested, are quite even. However, a gateway is sometimes considered a "common practice" [30] for DDD, as the facade can be easily built on bounded contexts, combining its elements to create a "tailored" interface. In our context of DDD, gateways also require abstraction and anticorruption layers [19]. The interfaces also provide minimum data, thus being a testimonial to the low coupling degree of the resulting architecture.

8.5 Cohesion

As already mentioned, the internal similarity of each service is more linear in DFD as the rules applied for the decomposed DFD are used to extract "operation and its output data modules [...] into microservice candidates" [13]. Such rules therefore also ensure a high cohesion degree between the modules. In DDD however, this can be more problematic. In the Auth bounded context, most methods have the self, request, and response parameters. However, in the API-bounded context, the parameters are more diverse: status, message, query set, captcha-challenge, etc. This is easier solvable in DFD, yet in DDD it is hard to overcome as the domain models are the same, yet external integrations need different values. Therefore, SIDC's resulting value is closer to 1 in DFD, given the decomposable DFD rules that aggregate the same operations with their data types. However, in DDD this is hard to achieve automatically and needs further consideration from developers. When it comes to Service Interface Usage Cohesion, DFD again yields more promising results.

By design, the "operations with the same output data" [13] need to be compiled into the same process, which makes the usability index of the resulting services very high. Thus, the "abstract meaning concepts" introduced by the decomposed processing make DFD yield a high value for SIUC. The value of TSIC is the weighted average of SIDC and SIUC, which given the previous results, is generally higher for DFD. In our case, the exact value of this metric is determined by the abstraction degree previously mentioned, given the data model resulting from the purified DFD. The most notable example is within the Project management service, where as shown by the decomposed DFD (figure 3), DDD's contextual operations are reformatted into "Account Profile Management", "Project Management", "Member invitation Management", etc. This value would be even more prominent should there be additional complexity within the User bounded context.

8.6 Dynamic Similarity Index

Although two different approaches, the microservices yielded by DDD and DFD, in this case, tend to be similar in places where abstraction cannot be defined [3]. One of these is, for example, the "project" bounded context. In the context of DFD, we can see that the sumPairs has more instances where entities have considerable consecutive accesses. For example, in the API service, the ping/event/notification entities are closely related in functionality, therefore the similarity metric will be close to 1. On the other hand, DDD provides little to no similar degree, shared between bounded contexts, as all API preprocessing is done inside the API bounded context. The "maxPair" value is proportional to the sumPairs outliers, indicating a correct application of decomposition rules. By the proposed Similarity Index (Formula (3)), it can be easier to evaluate DDD rather than DFD, as using graph calls, the architect can decide whether two domain elements should be in the same bounded context given their frequent interconnection. On the other hand, the previously mentioned high index would improve the performance of interfaces and gateway APIs since it removes complexity from API communication.

8.7 Evolvability

As previously mentioned, DFD sometimes requires a "centralized governance" entity, in order to preserve data states and consistency. On the other hand, the exact comparison factor of "autonomous teams" can be seen in DDD. The other metric is "non-patronizing usage of tools and metrics" [7], which in the case of DFD must be essential to assure data consistency (brokers, messaging queues, etc.). In our context, the clear winner is DFD, although it is also advocated that it is enough to be highly uniform, highly cohesive, and low coupling [14]. Moreover, the correct general application of granularity is in direct relation to this criterion, as a better-sized and low-coupled set of services directly improve maintainability. For our DDD-resulted architecture, maintainability is high as the workload can be decentralized and teams do not have to coordinate to ensure consistency, which imposes additional overhead using various DevOps rules or comprehensive documentation [7].

9 THREATS TO VALIDITY

The evaluation of the two decomposition approaches has been performed without loss of generality. The most important thread to the validity of this analysis is the business context and business functions of each initial monolithic system. Systematic studies for each individual decomposition process have been conducted given a certain degree of prerequisite business functions. However, this paper had a general applicability approach in mind, thus neglecting the individual business logic side. This can have major implications when it comes to unbiased analysis. However, this concern regarding the correlation between business functions and decomposition methods is widely recognized amongst architects and researchers, which can be a topic of further discussion and research in itself. When it comes to metric-based analysis, given the current state of the art, there are not generally approved and applicable metrics, and frequently the evaluation criteria are contradicting, as mentioned throughout the paper. Therefore, the set of criteria previously compiled is applicable to only a subset of microservice architectures, especially based on either DDD or DFD. Moreover, some approaches were not fully evaluated in an absolute way. For example, the framework proposed by Cojocaru et. al [15] presented some tests with only two microservices, which by the metrics for coupling and cohesion, might lead to biased results.

10 CONCLUSION

This research paper analyzed two microservices architectures, resulting from the destructuration process of a monolithic system using DDD and DFD. The results of the metric-based comparison yield different results, given the contextual use and applicability degree of the two approaches. For instance, DFD yields better results when considering Cohesion, or DDD performs better when it comes to async communication. As the evaluation has been performed in an objective and function-oriented manner, this differentiation can be easily outlined. The related contextual use, and the benefit of one decomposition method over the other, have both been discussed throughout the metric definition and evaluation sections. DDD can help designers understand granularity levels, helping them achieve more granular and modular microservices [25]. DFD can help designers understand data serialization, centralization, and consistency issues. Async messaging concerns were discussed and contextual solutions were proposed. The benefits of introducing bounded contexts were described throughout the evaluation, and formula-based reasoning was given for the corresponding sections, whose benefits tend to outweigh the ones introduced by DFD in numerous cases where data consistency is problematic. Further areas where developer implication is crucial, and method-based evaluation is questionable, such as cohesion, were also analyzed and tackled. A future improvement that could build upon the conclusions of this study can be the dynamic evaluation of the same metrics. Comprehensive runtime-based evaluation can follow, and with it, a more extensive set of metrics could be considered, which could further cover runtime-based or orchestration-based metrics while encompassing both static and dynamic metric-related factors.

REFERENCES

[1] 2014. Microservices. *martinfowler.com* (2014). https://martinfowler.com/articles/microservices.html

[2] Omar Al-Debagy and Peter Martinek. 2021. A microservice decomposition method through using distributed representation of source code. *Scalable Computing: Practice and Experience* 22, 1 (2021), 39–52.

[3] Bernardo Andrade, Samuel Santos, and António Rito Silva. 2022. From Monolith to Microservices: Static and Dynamic Analysis Comparison. arXiv:2204.11844 [cs.SE]

[4] Luciano Baresi, Martin Garriga, and Alan De Renzis. 2017. Microservices identification through interface analysis. In *Service-Oriented and Cloud Computing: 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings 6*. Springer, 19–33.

[5] Nichlas Bjørndal, Luiz Araújo, Antonio Bucchiarone, Nicola Dragoni, Manuel Mazzara, and Schahram Dustdar. 2021. Benchmarks and performance metrics for assessing the migration to microservice-based architectures. *Journal of Object Technology* (08 2021). https://doi.org/10.5381/jot

[6] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. 2022. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access* 10 (2022), 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

[7] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. 2019. Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges. https://doi.org/10.1109/ICSME.2019.00089

[8] Justus Bogner, Stefan Wagner, and Alfred Zimmermann. 2017. Automatically Measuring the Maintainability of Service-and Microservice-based Systems – a Literature Review. https://doi.org/10.1145/3143434.3143443

[9] Justus Bogner, Stefan Wagner, and Alfred Zimmermann. 2017. Towards a Practical Maintainability Quality Model for Service-and Microservice-based Systems. 195–198. https://doi.org/10.1145/3129790.3129816

[10] Miguel Brito, Jácome Cunha, and João Saraiva. 2021. Identification of microservices from monolithic applications through topic modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 1409–1418.

[11] Matteo Camilli, Carmine Colarusso, Barbara Russo, and Eugenio Zimeo. 2020. Domain Metric Driven Decomposition of Data-Intensive Applications. 189–196. https://doi.org/10.1109/ISSREW51248.2020.00071

[12] Roberta Capuano and Henry Muccini. 2022. A Systematic Literature Review on Migration to Microservices: a Quality Attributes perspective. 120–123. https://doi.org/10.1109/ICSA-C54293.2022.00030

[13] Rui Chen, Shanshan Li, and Zheng Li. 2017. From Monolith to Microservices: A Dataflow-Driven Approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 466–475. https://doi.org/10.1109/APSEC.2017.53

[14] Michel Cojocaru, Ana-Maria Oprescu, and Alexandru Uta. 2019. Attributes Assessing the Quality of Microservices Automatically Decomposed from Monolithic Applications. 84–93. https://doi.org/10.1109/ISPDC.2019.00021

[15] Michel Cojocaru, Alexandru Uta, and Ana-Maria Oprescu. 2019. MicroValid: A Validation Framework for Automatically Decomposed Microservices. https://doi.org/10.1109/CloudCom.2019.00023

[16] Zhamak Dehghani. 2018. How to break a Monolith into Microservices. *martinFowler. com, Apr* 24 (2018), 12.

[17] Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. 2018. *Evaluation of Microservice Architectures: A Metric and Tool-Based Approach*. 74–89. https://doi.org/10.1007/978-3-319-92901-9_8

[18] Eric Evans. 2004. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

[19] Diogo Faustino, Nuno Gonçalves, Manuel Portela, and António Rito Silva. 2022. Stepwise Migration of a Monolith to a Microservices Architecture: Performance and Migration Effort Evaluation. arXiv:2201.07226 [cs.SE]

[20] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. 2019. From monolith to microservices: A classification of refactoring approaches. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers 1*. Springer, 128–141.

[21] Sara Hassan, Rami Bahsoon, and Rick Kazman. 2020. Microservice transition and its granularity problem: A systematic mapping study. *Software: Practice and Experience* 50, 9 (2020), 1651–1681. https://doi.org/10.1002/spe.2869 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2869

[22] Healthchecks. [n. d.]. Healthchecks/healthchecks: A cron monitoring tool written in python amp; django. https://github.com/healthchecks/healthchecks

[23] Benjamin Hippchen, Pascal Giessler, Roland Steinegger, Michael Schneider, and Sebastian Abeck. 2017. Designing microservice-based applications by using a domain-driven design approach. *International Journal on Advances in Software* 10, 3&4 (2017), 432–445.

[24] Benjamin Hippchen, Michael Schneider, Pascal Giessler, and Sebastian Abeck. 2019. Systematic Application of Domain-Driven Design for a Business-Driven Microservice Architecture. *International Journal on Advances in Software Volume 12, Number 3 & 4, 2019* (2019).

[25] Garvit Jain, Urjita Thakar, Vandan Tewari, and Sudarshan Varma. 2021. A Survey on Trending Topics of Microservices. *International Journal* 9, 8 (2021).

[26] Pooyan Jamshidi, Claus Pahl, Nabor Mendonça, James Lewis, and Stefan Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35 (05 2018), 24–35. https://doi.org/10.1109/MS.2018.2141039

[27] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. 2019. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering* 47, 5 (2019), 987–1007.

[28] Justas Kazanavičius and Dalius Mažeika. 2019. Migrating Legacy Software to Microservices Architecture. In *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*. 1–5. https://doi.org/10.1109/eStream.2019.8732170

[29] Holger Knoche and Wilhelm Hasselbring. 2018. Using microservices for legacy software modernization. *IEEE Software* 35, 3 (2018), 44–49.

[30] Martin Lehmann and Frode Eika Sandnes. 2017. A Framework for Evaluating Continuous Microservice Delivery Strategies. In *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing* (Cambridge, United Kingdom) *(ICC '17)*. Association for Computing Machinery, New York, NY, USA, Article 64, 9 pages. https://doi.org/10.1145/3018896.3018961

[31] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. 2016. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. arXiv:1605.03175 [cs.SE]

[32] Shanshan Li, He Zhang, Zijia Jia, Zheng Li, Cheng Zhang, Jiaqi Li, Qiuya Gao, Jidong Ge, and Zhihao Shan. 2019. A dataflow-driven approach to identifying microservices from monolithic applications. *Journal of Systems and Software* 157 (2019), 110380. https://doi.org/10.1016/j.jss.2019.07.008

[33] Lei Liu, Zhiying Tu, Xiang He, Xiaofei Xu, and Zhongjie Wang. 2021. An Empirical Study on Underlying Correlations between Runtime Performance Deficiencies and "Bad Smells" of Microservice Systems. 751–757. https://doi.org/10.1109/ICWS53863.2021.00103

[34] João Lourenço and António Rito Silva. 2022. Monolith Development History for Microservices Identification: a Comparative Analysis. *arXiv preprint arXiv:2212.11656* (2022).

[35] Luís Nunes, Nuno Santos, and António Rito Silva. 2019. From a monolith to a microservices architecture: An approach based on transactional contexts. In *Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings 13*. Springer, 37–52.

[36] Sebastiano Panichella, Mohammad Imranur Rahman, and Davide Taibi. 2021. Structural Coupling for Microservices. arXiv:2103.04674 [cs.SE]

[37] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. 2018. Towards a UML Profile for Domain-Driven Design of Microservice Architectures. In *Software Engineering and Formal Methods*, Antonio Cerone and Marco Roveri (Eds.). Springer International Publishing, Cham, 230–245.

[38] Chris Richardson. 2018. *Microservices patterns: with examples in Java*. Simon and Schuster.

[39] Thatiane Rosa, João Daniel, Eduardo Guerra, and Alfredo Goldman. 2020. A Method for Architectural Trade-off Analysis Based on Patterns: Evaluating Microservices Structural Attributes. 1–8. https://doi.org/10.1145/3424771.3424809

[40] Dmytro Rud, Andreas Schmietendorf, and Reiner Dumke. 2006. R.: Product metrics for service-oriented infrastructures.

[41] Roland Steinegger, Pascal Giessler, Benjamin Hippchen, and Sebastian Abeck. 2017. Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications.

[42] Mehmet Söylemez, Bedir Tekinerdogan, and Ayça Kolukısa Tarhan. 2022. Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review. *Applied Sciences* 12, 11 (2022). https://doi.org/10.3390/app12115507

[43] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. 2018. Architectural Patterns for Microservices: A Systematic Mapping Study. https://doi.org/10.5220/0006798302210232

[44] Davide Taibi and Kari Systä. 2019. From monolithic systems to microservices: A decomposition framework based on process mining. (2019).

[45] Davide Taibi and Kari Systä. 2020. *A Decomposition and Metric-Based Evaluation Framework for Microservices*. 133–149. https://doi.org/10.1007/978-3-030-49432-2_7

[46] Shmuel Tyszberowicz, Robert Heinrich, Bo Liu, and Zhiming Liu. 2018. Identifying microservices using functional decomposition. In *Dependable Software Engineering. Theories, Tools, and Applications: 4th International Symposium, SETTA 2018, Beijing, China, September 4-6, 2018, Proceedings 4*. Springer, 50–65.

[47] Fredy Vera-Rivera, Eduard Puerto-Cuadros, Hernán Astudillo, and Mauricio Gaona. 2020. *Microservices Backlog - A Model of Granularity Specification and Microservice Identification*. 85–102. https://doi.org/10.1007/978-3-030-59592-0_6

[48] Hulya Vural and Murat Koyuncu. 2021. Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice? *IEEE Access* 9 (2021), 32721–32733. https://doi.org/10.1109/ACCESS.2021.3060895