Creation of a Framework and a Corresponding Tool Enabling the Test-Driven Development of Microservices

Christian Daase*, Daniel Staegemann, Matthias Volk, Klaus Turowski
Institute of Technical and Business Information Systems, Faculty of Computer Science, Otto von Guericke University Magdeburg, Germany.

* Corresponding author. Email: christian.daase@ovgu.de
Manuscript submitted September 5, 2022; revised November 23, 2022; accepted January 23, 2023.
doi: 10.17706/jsw.18.2.55-69

Abstract: Microservice architectures have emerged as counter design to traditional monolithic applications. While monoliths are single executable applications, microservice architectures consist of several smaller units. Advantages of microservice architectures are their development speed, lower costs of change, and dynamic scaling ability. However, this pattern requires an adaptation of quality assurance measures. In this research article, test-driven development is investigated in context of microservices that are developed according to established practices. Based on the systems development research methodology, recommendable practices and testing strategies are examined with a translation of that knowledge into an extensive artifact, enabling test-driven microservice development on local systems. Five design principles could be identified, including focusing small services, domain-driven design, striving for low-complexity networks, avoiding cyclic dependencies, and aiming for high connectivity performance. Integration, component, and contract tests could be integrated for automatic execution, showing that test-driven development for microservices is feasible, although with room for improvements.

Keywords: Design science research, microservice, software engineering, test-driven development.

1. Introduction

In software development, delivering best quality products is one main goal of many projects, leading to an overarching importance of sufficient test planning and execution [1–3]. On the two sides of the spectrum, test-driven development (TDD), where tests are written before the actual code is produced, and test-last development (TLD), where tests are written after a new functionality is implemented, have been established as recognized approaches [4]. Since the designation *test* describes a wide range of quality assurance practices on different levels (i.e., complexity and the amount of functions to be tested at once), especially TDD is often limited to unit tests in the literature, meaning the automation of test routines for the smallest pieces of software [4, 5]. Since testing is such an important part of software development and the term test-*driven* suggests that the whole development cycle is shaped by the careful integration of constantly executed quality assurance measures, it is worth to take a closer look on opportunities to expand this definition to larger portions of code and more complex emerging software structures and architectures.

While a unit test is usually designed to test the smallest piece of testable functionality [6, 7], meaning a set of code lines [1] of which any software product usually consists, TDD on the lowest level of functionality testing is not a new phenomenon. In fact, TDD was initially proposed in the late 1990s as a key part of the *extreme programming* development philosophy by Kent Beck [8], but the general mentality on how to build

complex software products has changed since then. For a long time, the predominant approach for software development were so-called *monoliths*, single executable applications whose modules share the resources of one machine (i.e., computation power, storage, and file system) [9]. This traditional practice provides a few positive characteristics. Especially small applications can be quickly deployed and bought to market [10, 11] without considering, for example, overly complex architectural challenges in component communication. However, a downside shows when these monoliths grow over time and thereby evolve into nearly impossibly maintainable and changeable projects since everything inside the system is tightly connected or interdependent [12]. The implementation and testing of new features becomes increasingly difficult as every action may introduce regression defects [10], thus slowing down the overall development process [13]. In particular, rapid innovation and agile delivery is severely restricted due to monolithic architectures [14].

As an alternative design approach, *microservice architectures* (MSAs) propose a decentral system concept [15] based on the idea of splitting an application into independently executable and isolated functional units (i.e., applications) called microservices [9]. These services can be developed individually by different teams, in varying languages, with respective preferred frameworks and technologies [10, 12, 13, 16]. In order to construct a complete system, the services should be able to communicate over lightweight messages through defined interfaces [9, 10, 15]. Since it is generally considered a good strategy to keep highly interdependent business capabilities together [10, 15], a basic microservice design practice has been coined as "*loose coupling and strong coherence*" [16]. Multiple major companies already adopted MSAs for their services, including Twitter, Facebook, Netflix, and Amazon [14, 17, 18].

The testing of individual components such as code blocks and single services in MSAs and the automation of these procedures have already been considered in the literature [6, 14, 16]. However, no publication including a comprehensive framework for the TDD of MSAs with an evaluated instantiation of an operational development environment as a proof of concept could be found in preparation of this scientific investigation. Therefore, the contribution of this paper consists of two theoretical and one practical part. First, peculiarities of MSAs are investigated in order to recommend a common development approach. Second, software testing strategies are reviewed and evaluated regarding their suitability with respect to a conceivable TDD approach for MSAs. And third, these findings are combined into one piece of software (i.e., an artifact) to prove the feasibility of the elaborated TDD strategy for MSAs in one specific use case with the possibility of extension in future research and engineering endeavors. The focus for the usefulness of the artifact to be created is placed on a user with minimal technical capabilities to make its design more tangible to a broader audience. This work therefore aims on answering the following research questions (RQs):

RQ1: *Which design principles of microservices should be considered in a development and automated testing environment based on established best practices for MSAs derived from the literature?*

RQ2: *Which test types beyond unit tests could be implemented in an environment for the TDD of microservices and how could such an environment be designed?*

2. Methodology

Research efforts revolving around engineering and systems development offer a wide range of methodologies [19]. One branch within this field is called *design science research* (DSR) in which the main outcome is considered to be a usable and testable artifact in the form of a construct, a model, a method, or an instantiation [20]. Hevner et al. define the term *design* in the context of DSR as "*the purposeful organization of resources to accomplish a goal*" [20] and remark that it should be understood as a twofold phenomenon, a process in the form of a set of activities as well as a product in terms of an innovative artifact. Furthermore, the duality of this type of research manifests itself in the split between theoretical foundations and the subsequent practical integration of these findings into a new development, which requires two different

kinds of methods for these aspects [22]. In a more recent version of this understanding, vom Brocke, Hevner, and Maedche note that the goal of DSR is "*to extend the boundaries of human and organizational capabilities*" [23]. Since this research aims to specifically address an organizational problem (i.e., the trade-off between quality assurance and productivity) that arises from the characteristics of the development approach of TDD and a software architecture style in the form of MSAs, the research conducted here is suitable for applying a DSR methodology. In this section, the *systems development research methodology* (SDRM) introduced by Nunamaker *et al.* [19] is outlined as the adopted specific DSR methodology for this research.

2.1. Systems Development Research Methodology

The SDRM consists of five research steps building on the four research strategies of *systems development* and, revolving around that, *theory building*, *experimentation*, and observation. The (I) *construction of a conceptual framework* serves to justify the significance of the pursued RQs within a project which is ideally "*one that is new, creative, and important in the field*" [19]. The conceptual framework leads to theory building in terms of understanding the domain under investigation and envisioning first ideas and new approaches. This step is performed by means of a two-part systematic literature review (SLR) on microservices and MSAs in the first part, and software testing techniques and strategies in the other one. The outputs of this activity are the theoretical foundations needed to compile a set of presumably best practices of microservice development and a list of common test routines in ascending order by their degree of complexity. A visual representation of the interplay of the capabilities a test environment (i.e., the artifact) should satisfy and the developer concludes this phase.

The (II) *development of a system architecture* is then conducted by relating the principles to implementable technical components a TDD environment could integrate. The methods mainly used in this phase can be labeled as *conceptual engineering*, *knowledge synthesis*, and *logical reasoning*. As a result, the abstract connections between MSA principles and realistic capabilities of a TDD environment can be highlighted. Subsequently, the (III) *analysis and design of the system* are presented as a connector between the abstract knowledge from (II) and the phase of (IV) *building the system*. In (III), the verifiable characteristics of the two subject areas (i.e., MSAs and TDD) as well as the synthesized knowledge gathered in (II) are merged into a practical system design. To complete the explanations, essential parts of the aspired artifact are depicted. This step encompasses the methods of (II) and additionally a *graphical visualization*.

Building the artifact requires the active development and realization of its architecture. Therefore, the method of *prototyping* is adopted in this phase, which can also be viewed as the core element of any DSR endeavor since knowledge and understanding of the design problem and its solution can be primarily generated by building and ultimately applying the artifact [20]. The usable TDD environment for locally developed microservices can subsequently be set in operation to demonstrate its feasibility and to gather experience regarding the advantages when utilizing it [19]. These insights from active application furthermore serve as a basis for a possible redesign of the system in general or individual components. The DSR approach closes with the (V) *observation and evaluation of the system* by testing through the method of *experimentation* and *simulation in an artificial scenario*. By defining a potential scenario corresponding to a developer's assumed workflow when following the identified best practices, qualitative aspects of the artifact can be assessed. However, since the main purpose of this research work is to provide theoretical engineering knowledge and to propose *one* solution focusing on its functional capabilities rather than on its performance, attributes such as response times are not considered because of the dependency on the underlying hardware. Fig. 1 illustrates the explained research process according to an adapted version of the SDRM of Nunamaker *et al.* [19]. Each activity is placed in context with the methods employed in it and the aspired outcomes. Phase (II) and (III) as well as (IV) and (V) are combined respectively as their executions and results are strongly connected.

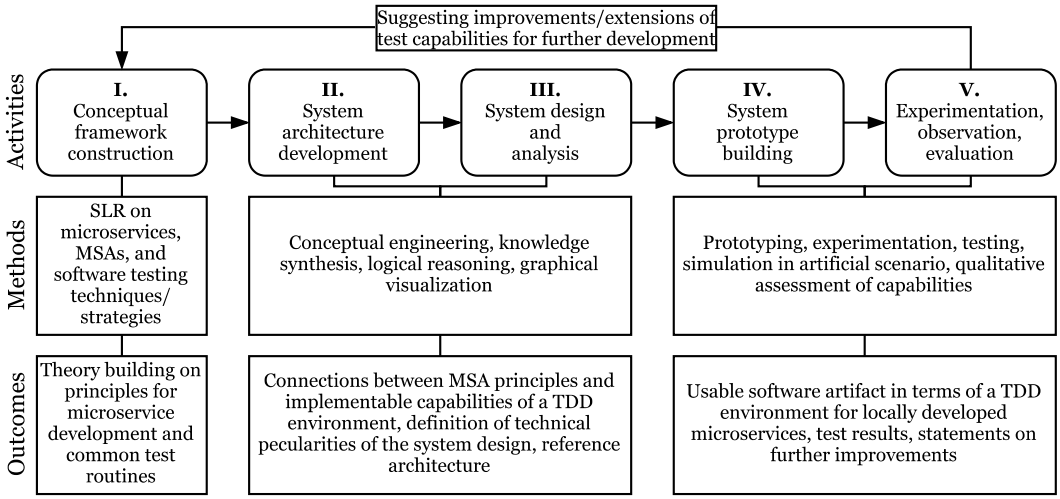


Fig. 1. Adapted research process according to the SDRM.

2.2. Systematic Literature Review

The structure of the SLR is presented to ensure a high degree of comprehensibility. To this end, this section serves as a review protocol that is synthesized from the guidelines of Kitchenham and Charters [24], who adapted guidelines derived from the medical research area for software engineering projects, and Okoli [25], whose remarks complement the methodology with a closer connection to information systems research. The protocol includes statements on databases, search terms, review stages, and inclusion and exclusion criteria.

The databases Scopus and SpringerLink are chosen for the literature search on microservice development principles and software testing techniques. Scopus, on the one hand, is an abstract and citation database linking a variety of different scientific full-text databases such as IEEE Xplore, ACM Digital Library, and ScienceDirect [26] that furthermore claims to be the largest database of its kind [24]. SpringerLink, on the other hand, provides additional access to its own high-quality and peer-reviewed journals that might be not retrieved when using Scopus exclusively [24]. Therefore, these two databases are employed as complementary parts in this SLR to improve the overall quality of the final literature body.

Kitchenham and Charters [24] suggest the adoption of a multi-part search term structure, which consists of three elements in its most basic variant. First, the *population* describes the domain, actors, organization, or technology under investigation. Since the SLR is structured into two fields of interest, microservice development practices and software testing techniques, this criterion has to be two-fold as well. Test searches revealed that the number of retrieved articles in Scopus and SpringerLink is already sufficiently limited when simply adopting the phrase *microservice* as *population* for the former area. For the latter area, the phrases *software*, and *microservice* are established as *population*. As the second structure element, the guidelines state that the *intervention* can be understood as "*the software methodology / tool / technology / procedure that addresses a specific issue*" [24]. Hence, this element can be regarded as an action that is carried out in the domain, respectively on the *population*, and that effects its properties. For the first review part, the *intervention* is defined with the terms *develop** and *implement**, as these two terms usually aptly describe the creation of a new piece of software in computer science. The asterisks are used as a wildcard character, thus including similar terms. The *intervention* of the second search (i.e., on testing) consists of the term *test** in conjunction with *technique* or *strateg**. The *outcome* represents the last element of the search phrases. According to the definition of Kitchenham and Charters, the "*outcomes should relate to factors of importance to practitioners*" [24]. In other words, this part studies the effects that the *intervention* assumingly has on the

population. In case of the microservice related search, the *outcomes* revolve around the issues of *best practices*, *principles*, and *patterns*, whereby the last term is an addition found to be useful at the time of trying different test phrases. The *outcomes* of the second search can be stated as *quality*, *success**, and *complex**. Table 1 summarizes the used search phrases and the queried databases with an additional identifier. Based on test searches and to sufficiently limit the number of retrieved publications, the three components of each query are entered in different search fields, as shown in the table.

Table 1. SLR Query Specifications

ID	Database	Search string specifications
Q1	Scopus	*Title*: microservice *Abstract*: (develop* OR implement*) AND ("best practice" OR principle OR pattern)
Q2	SpringerLink	*Title*: microservice *All of the words*: development *At least one of the words*: practice principle pattern
Q3	Scopus	*Title*: (software OR microservice) AND testing *Abstract*: testing AND (technique OR strateg*) AND (quality OR success* OR complex*)
Q4	SpringerLink	*Title*: microservice *All of the words*: testing *At least one of the words*: technique strategy

The SLR is divided into four stages in which appropriate inclusion and exclusion criteria are applied: Stage 0 relates to the automatically retrieved articles, Stage 1 to the phase of reading the titles and abstracts, Stage 2 to reading the introductions and conclusion, and Stage 3 to reading the full texts in detail. The criteria applied are divided into a general part and one specific part for each of the two subjects under study. The literature review was conducted in March 2022. The time frame, however, is limited to 2021, so that all published articles up to this date are included. Thus, all stated criteria were applied to the original body of literature. According to the employed guidelines, an initial set of criteria was tested in preparation and iteratively refined during the search process [24]. In the case of subsequent adjustments, the existing criteria were checked again retroactively to see whether they still applied to the previously evaluated articles. For the definition of the inclusion and exclusion criteria, the recommendations of Okoli [25] are considered. Table 2 summarizes the inclusion and exclusion criteria with their corresponding review stage in parentheses.

Table 2. SLR Inclusion and Exclusion Criteria

Queries	General criteria	Inclusion criteria	Exclusion criteria
Q1, Q2 (microservice development practices)	(0) Published between 2018 and 2021 (0) Written in English language (0) Conference paper or journal article (0) Completed and finalized research (1) No duplicates	(1) Main topic is microservice development (1) General use case scenario for microservices	(1) Microservices in active use, not development (1) Other context of *principles*
		(2) Variety of principles in context (2) Focus on recommendable practices	(2) No semantic duplicate (2) Only scant overview
		(3) Practical assessment of principles	
Q3, Q4 (testing techniques and strategies		(1) Testing and test types as topic, not analysis (1) Software-related domain	(1) Unsuitable context of *test* (1) Experience report (1) Specific product test
		(2) Examination of several different applicable software test types	(2) Outside of the context of local development
		(3) Details on investigated tests	(3) Insufficient depth

2.3. System Development and Evaluation

The design and construction of the artifact follows the methodological guidelines presented by Nunamaker et al. [19]. The practical research phases encompass the steps II to V shown in Fig. 1, beginning with the rather abstract creation of the design and its subsequent refinement to the actual development, presentation, and evaluation. Regarding the guidelines for *design evaluation* by Hevner et al., it is stated that the "*evaluation*

includes the integration of the artifact within the technical infrastructure of the business environment" [20], which refers in the context of this work to the workstation on which the software for the TDD of microservices is developed. Therefore, the artifact itself is already constructed within the context of its intended usage. The relevant quality attributes are functionality, usability, and appropriateness for the problem domain rather than quantifiable performance metrics that depend on the underlying system. From the proposed methods, experimentation is the most suitable, as the determination of the solvable issue is more of a range of problems within a problem domain summarized under the term *test-driven development*. Therefore, an exemplary use case is constructed, which is consistent with the two sub-methods of *controlled experiment*, meaning the evaluation in a supervised environment, and *simulation*, which is testing the artifact with artificial data.

3. Quantitative SLR Results

Entering the queries Q1 to Q4 according to the definitions in the review protocol into the databases Scopus and SpringerLink with additional filters defined for the review stage indexed with 0 yielded 84 articles for Q1, 81 articles for Q2, 133 articles for Q3, and 63 articles for Q4. For the subsequent analysis, the retrieved publications from the review paths Q1 and Q2 as well as Q3 and Q4 were examined in combination. Thus, the investigation of recommendable practices for microservice development is based on 165 publications while the elaboration of software testing techniques and strategies is based on 196 scientific contributions. Fig. 2 illustrates the completed review process, including information about how many articles were excluded due to which criterion. In total, 16 publications remained after the strict execution of the review.

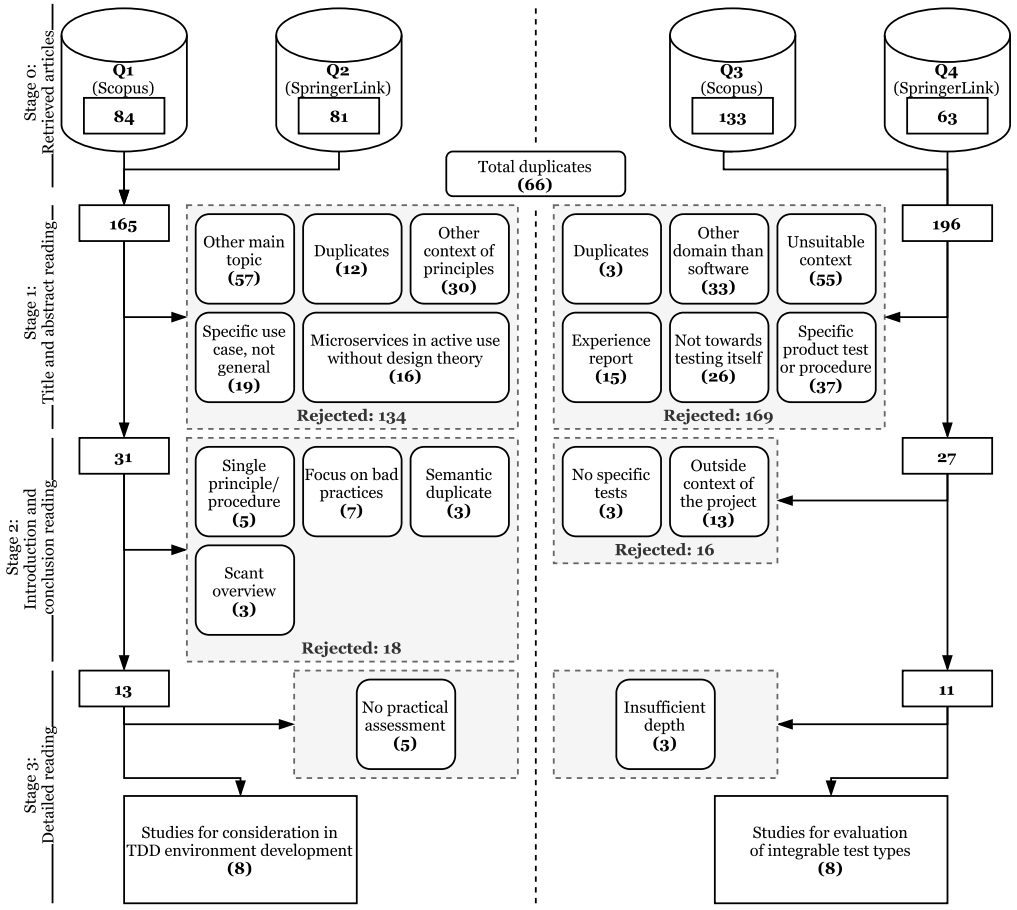


Fig. 2. Literature process visualization.

4. Conceptual Framework

Since microservices allow for an incremental transition strategy [13] and evolutionary deployment due to the internal independence of services [12], MSAs can be employed for continuous integration/continuous delivery (CI/CD) pipelines [10, 11, 18, 27]. General advantages reach a lot further when the services are appropriately designed. The small functional size can speed up the development process while decreasing the costs [12] with the additional characteristic that the system might be easier to scale in times of heavy exposure [28]. Combined with a standardized communication through messages [9], this offers a facilitator to quickly onboard new employees without having them to understand complex dependencies and program codes of the whole system [28]. This would be in particular problematic, since MSAs can contain thousands of services and therefore might grow very large [14],[15]. The flexibility to choose dedicated technologies and languages for individual microservices [6, 11, 12] also increases the openness to new approaches, the reusability, and the potential diversity of talents in the team [12]. However, there are some factors that militate against MSAs in certain contexts. First, organizations need to understand the potential benefits compared to monolithic approaches [13] to avoid issues regarding testing and code analysis [6, 15], insufficient time-to-market because of an overall limited project scope [10], and staff shortage due to an inconsiderate integration of too many different programming languages with only a few experts involved [28]. Still, MSAs are expected to attract more attention in the future with more matured technologies such as Kubernetes [16] for large-scale architectures, justifying further investigations on how to assure quality.

Testing a system requires at least four activities: planning and implementing a test, as well as executing it with a subsequent evaluation of the results [1]. When the tests are written only before the functionality is implemented, this approach equates with the TDD concept. Microservices, with their assumed advantage of boosting the development speed up to CI/CD standards [10, 18, 27], can form a powerful symbiosis with TDD, when it is considered that although TDD may lower productivity, it increases code quality [5], which cancels out this possible disadvantage. From the reviewed literature, different testing strategies and techniques can be derived from which a set for realization in the TDD environment has to be determined as implementable in order to construct a conceptual framework.

The first identified and most basic test type (indexed with T1 in the summarizing table) are *unit tests*. A *unit* can be commonly described as the "*smallest part of the application*" [7] or the "*least possible set of lines of code which can be tested*" [1]. Since microservices themselves are independently developed components able to run isolated and communicate with each other via defined APIs [9, 10, 15], these software units can be viewed simply as applications which are usually of a much smaller scale than monolithic applications according to common understanding. Therefore, unit testing can be performed in the same manner for both architectural styles. Integrated development environments (IDEs) such as Microsoft Visual Studio or PyCharm often have built-in tools or at least the option to be extended with capabilities to perform unit tests directly. Hence, the aspired TDD environment does not need to integrate special capabilities to perform unit tests, so the development of this artifact can focus on enabling more complex test types for which no trivial solutions exist. In this context, the second test type, (T2) *integration tests*, imposes specific requirements on a TDD environment for MSA. This type of test is used to evaluate how modules (i.e., in this case the microservices themselves) perform in combination by verifying their connectivity in terms of interfaces, the transport of data, and the correct formats [1, 2, 6, 7, 14, 16]. Regarding the intended TDD capabilities, integration testing in this context is considered from a purely technical perspective, meaning the question of whether a microservice requested through a particular provided interface returns a valid response and not a failure that appears to be a server error. The semantic correctness is explicitly not checked by this test type in the final construction of the artifact. However, it is not claimed that this definition is universally accepted, so other software testers might evaluate the semantics of inputs and outputs here as well. The next test

strategy (T3) is designated either as *single microservice test* or *component test*, depending on the source. T3 focuses on the functioning of a service's internal logic and not just on its external communication [14]. The services are tested isolated from other services that are to be integrated in the same MSA in a later development stage and external dependencies, whereby necessary inputs are simulated and replaced with mock objects [6],[16]. For this type of test, the developer needs to know which output can be expected as a response to a certain input. *Contract tests* (T4) can be considered as a combination of integration testing and multiple component tests. This test system occasionally uses consumer-driven contract testing that mimics a customer's perspective by transmitting the required data and interaction details through an available interface of the system or the part of the system responsible for the desired functionality [6, 14]. Thus, the collaboration between services is evaluated and the logic of a larger portion of the system is verified by comparing the actual output with the expected one. The functional tests with the largest scale that could be identified in the available literature are (T5) *system tests* or *end-to-end tests*. As the name suggests, this test helps evaluating the entire system as a unit [1],[6]. Real-world requirements are imposed on the system from an external perspective, without touching internal details, to assess the extent to which the system meets the user's expectations during a process execution [7, 14, 16]. The penultimate identified strategy, (T6) *performance testing*, evaluates the efficiency of the system and its components, especially under extreme conditions and heavy workloads [16, 17]. Moreover, this technique helps to reveal weaknesses in the design and to examine how the system uses the available resources. The last test type, which is usually applied when the system is used as in production mode, is the (T7) *acceptance test*. After being handed over to the users, this test's purpose is to evaluate whether the system meets the user requirements and intended specifications [1, 7]. Alpha testing (i.e., making the software available to involved stakeholders) can be viewed as internal acceptance testing, while beta testing is an early form of external acceptance testing [2].

Table 3. Testing Strategies for the TDD Artifact

ID	Strategy/technique	Description
T1	Unit test	Testing of smallest piece of testable functionality [6, 7]; "least possible set of lines of code which can be tested" [1]
T2	Integration test	Verifying the operability of the combined modules regarding the interfaces and communication between them, especially recommendable for distributed systems [1, 2, 6, 14, 16]
T3	Single microservice test/component test	Validating whether a microservice provides the intended functionality/expected output [6, 14]; isolating a single service from external dependencies or replacing them with mock objects [16]
T4	Contract test	Testing the collaboration between microservices [14]; verifying validity on every application programming interface (API) change [6]
T5	System test/end-to-end test	Testing the whole system as a unit [1, 6]; validating whether the system meets the user expectations under real-world conditions [14]
T6	Performance test	Testing under extreme conditions and heavy load [16, 17]
T7	Acceptance test	Evaluating the system from the user's perspective for satisfaction [1, 7]

Table 3 summarizes the identified strategies with brief descriptions beginning with the most basic testing level at the top. The gray shaded rows contain techniques which are not considered in the following construction of the framework and the artifact. Unit tests are usually strongly connected to the used programming language and environment. Therefore, this technique is only considered at the stage of developing a service before being visible for the TDD environment. System tests, on the other hand, require the whole architecture being viewed as a unit. Although this can be possible with the intended artifact for small assemblies, this could surpass the capabilities and visual traceability inside the environment. Performance tests would require the system to be deployed within the final realistic scenario. Since the

artifact is designed for locally developed microservices which could be migrated later, it cannot fulfill the requirements for this type of test. Lastly, acceptance tests would necessitate an unbiased user perspective to evaluate whether the system satisfies the client's interest, which is not existent. The techniques selected for implementation are integration tests to verify the connectivity of services, single microservice/component tests to check the semantic functionality with artificial inputs, and contract tests to inspect the collaboration of microservices that depend on each other's results.

Five principles that are considered when designing the TDD environment could be identified. The most remarkable publication originates form Engel *et al.* [15], thus forming a basis for refinement in the following elaborations. The first one is to concentrate on (P1) *small, specialized services* with limited functionality [15]. Thus, an optimal scalability can be achieved, so that only a very dedicated capability of the overall system has to be replicated in order to increase the availability. Since every service possesses its own data management in contrast to the traditional monolithic approach, an issue might occur regarding the necessity to receive data simultaneously from multiple services for a specific further processing [10]. Consequently, an MSA can be a highly fragmented and decentral organized system [29], leading to the requirement for the TDD environment to be capable of managing connection details and properties of many different services.

The second extracted principle, (P2) *domain-driven design* [15, 27, 30], partly contradicts the first one since it aims for a higher cohesion of business capabilities to construct services within *bounded contexts* [29, 31] and, in turn, for lesser dependencies between the individual services. From an organizational perspective, Bozan *et al.* [13] state that "*business needs drive which services will be isolated and developed independently*". Aside from this possible type of division, it is also recommended to structure the services with respect to their required data access [28]. The attempt to balance out these first two principles corroborates the thesis, that no MSA can support every microservice tenet well at once [32]. For the TDD environment, however, the possibility of testing multiple services interdependently should be considered, which also corresponds to the most complex of the targeted test levels, contract testing.

The third principle can be stated as ensuring a (P3) *manageable, clear, low complexity network* [15]. Since an MSA and especially testable sections should contain as few connections as possible, a general representability within the 2D graphical user interface (GUI) of the TDD environment is reasonably aspired. Another question regarding the productive application of an MSA is how to make the system accessible for external users. Direct connections to every service would yield the issues of high coupling, difficulties with deploying, and the additional threat of damaging the infrastructure as a result of altering API endpoints [32]. Apart from this, security risks can arise if access to services is not strictly regulated. The only way to access functionality should be through the provision of dedicated interfaces [29], which, however, increases the complexity of development if every service is to be directly accessible from the outside. A better solution might be a common entry point such as an API gateway [18, 28]. Considering that in the test phase of a system it is unlikely that every component is in place, a management entity as a placeholder seems justified, orchestrating the interaction between services and able to interact internally with the whole microservice system. The issue that Rossi [33] refers to as the "*chain of calls*" would thus be significantly mitigated.

A fourth principle to avoid a termination of the system (e.g., due to deadlocks) is to include (P4) *no cyclic (synchronous) dependencies* [15]. Especially in highly granular MSAs with many interdependencies, cascading service calls may lead to vulnerability to failures [30], since cycles may not be obvious when there is a high degree of service connectivity. Therefore, potential cycles should be highlighted in the TDD environment. Generally, asynchronous messaging wherever possible should be preferred as services that are waiting for an response to a call could process other incoming requests in the meantime [18].

The fifth recommended practice is to keep in mind the (P5) *connectivity performance*, also simply shortened as performance in the literature [15], which is measured using the length of synchronous call traces and the

average size of messages. Although the strategy of performance tests is not included in the artifact's capabilities, these parameters independent from the performance of the run time environment can be tracked. Logging is an integral part of testing and debugging, but since MSAs consist of many components, it must be defined which entries from which services should be written to the log file, or whether only the overarching input and output parameters should be stored. Distributed tracing is a method to record traces on distributed components such as services participating in the fulfillment of a request [28]. As with the third principle, a central entity in terms of an API gateway can help to gather all feedback from every intermediate step of a procedure and to store them into a unified log file [32]. Engel et al. [15] remark that the architectural model must contain certain information to assess an MSA in regard of the principles. This includes information on the implemented microservices and parameters of their interfaces, the type of calls to understand communication dependencies, and an option to logically cluster the microservices. This leads to the assumption that microservices should be registrable and reusable to store the first part of the needed information. In addition, the definitions of the individual test sequences should contain information about the calls to be executed. Clustering is accomplished so that a setup consisting of multiple services registered in the TDD environment with a set of tests should be capable of being stored as a retrievable unit. Logging moreover helps to implement and evaluate the other principles within a microservice and ultimately helps to impose appropriate dependency and communication path restrictions as well as to avoid unnecessary data transfers [27, 29]. Table 4 summarizes the extracted principles for microservice development that are considered in the construction of the TDD environment, thus serving as an answer to RQ1.

Table 4. Principles in the Conceptual Framework

ID	Principle	Description	Consequence for TDD environment
P1	Small, specialized services	- Optimal scalability by splitting the system into very dedicated modules	- Manage multiple microservices' details - Ensure easy reusability of microservices in different scenarios
P2	Domain-driven design	- Keeping business capabilities together - Structuring by data access	- Capability of concatenating microservices for contract testing
P3	Manageable, clear, low complexity network	- Few intermediate connections - Restricting external access	- Ensuring representability of scenario in 2D GUI - Using a common entry point
P4	No cyclic (synchronous) dependencies	- No cycles of services depending on each other	- Highlight potential cycles/dead locks
P5	Connectivity performance	- Short call traces and small messages - Log intermediate results	- Central entity (API gateway) for logging intermediate results and test information

Three technical layers for the artifact can be set up. First, in preparation of the tests, the developer needs to be able to define which microservices are to be tested, which purpose the tests should fulfill, and how the microservices should work together. These *component definitions* combine the necessary requirements to meet P1 and P2 from Table 4 by ensuring the reusability of individual components and allowing for the construction of complex test assemblies. The definition of the components is part of the GUI that the developer also uses in the second layer by observing the *dynamic test area*. A gateway service forwards the requests defined in a single test to the respective microservices and receives the intermediate results. This procedure continues until the complete scenario consisting of multiple tests with different services involved reaches the endpoint, or until one part of the setup fails to respond with the expected outcome. The third technical layer is the underlying *persistent storage* where the gateway service stores scenario configurations and log files of results. Fig. 3 illustrates the conceptual framework with the layers for a test-driven development approach of microservices in whose design the identified principles are considered.

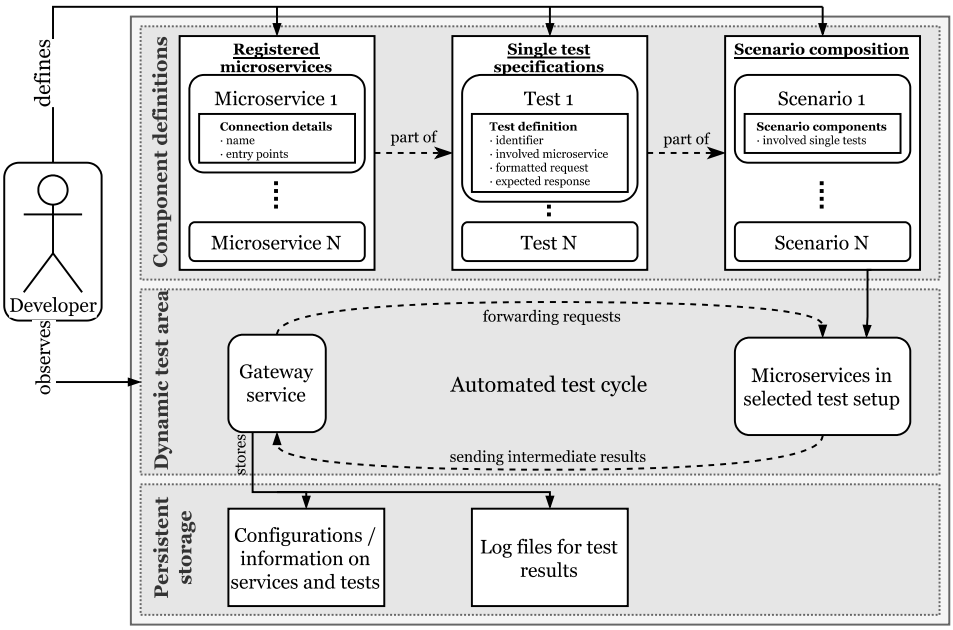


Fig. 3. Visualized conceptual framework for the TDD environment.

5. Practical Development and Evaluation

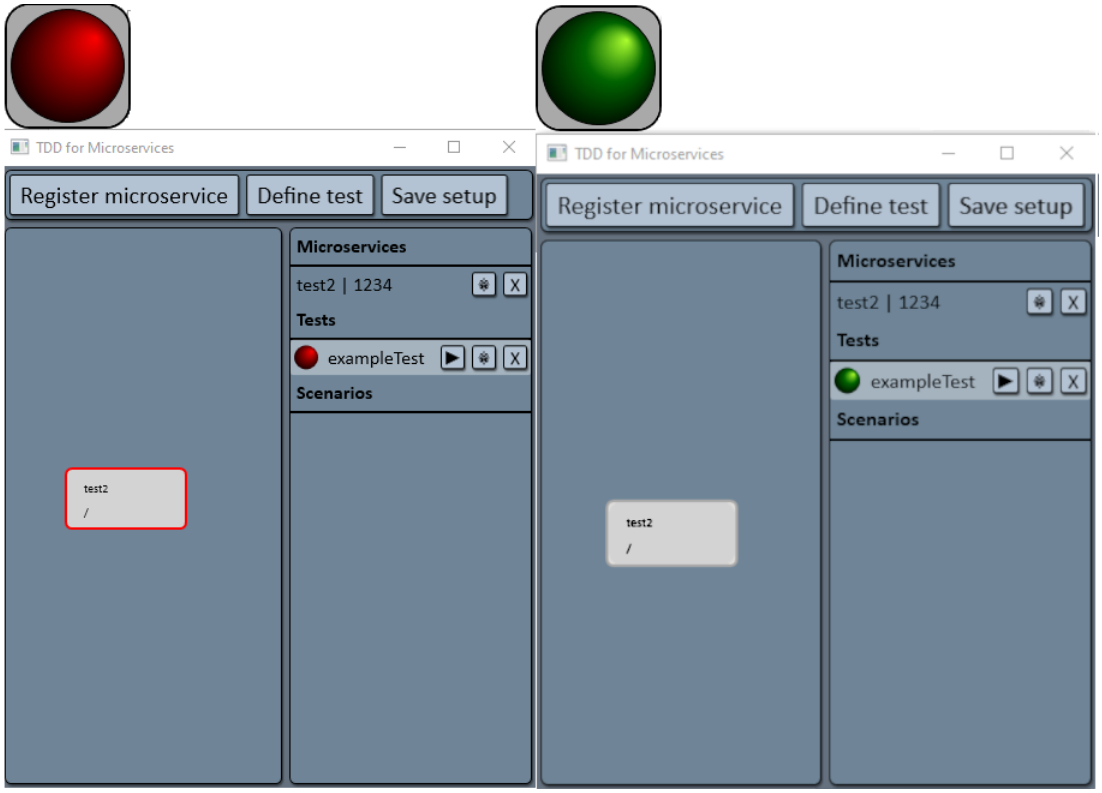
The GUI should contain the fundamental elements for managing components such as buttons for adding and editing registered microservices, tests, and scenarios. Moreover, displays on the status of test procedures are necessary to evaluate the state of the services. One distinctive feature of microservices is that they can usually be containerized. *Containerization* involves packaging an application, along with its dependencies and all required components into a lightweight, executable container image so that it can be transferred to and run on any system that provides the basic technological infrastructure [16]. Since this approach is also pursued in the creation of the artifact, the registration of testable microservices via the GUI also defines which services must be containerized again in case of code changes, because otherwise a containerized application is isolated from external developments (i.e., the further editing of the underlying source code). The second control element next to a button for registering microservices is a button for defining specific tests. A third control element is considered to store a complete scenario for later reuse. A *scenario* hereby includes the entirety of currently registered microservices in the TDD environment as well as the defined tests.

Another important element of the GUI is a panel for the lists of services, tests, and available scenarios. Clicking an *edit button* of a service or test in the corresponding list should open a pre-filled form with the currently valid definition of this component. Since the scenarios are based on the entirety of the available components at that moment, it is not necessary to allow manual editing here as well. Here, pressing a button to remove a scenario and saving them again is sufficient. The information presented in the panel includes headings for each category of components, names, ports, and the status of tests and scenarios. The *ports* are the microservices' entry points for internal communication, meaning that they do not need to be accessible from the outside or the operating system. The *status* of a test should be visually assessable.

The core element of the user interface is the panel for graphical visualization of the currently selected test and its detailed status with respect to each involved service and the intermediary connections. Unique designations of the services are necessary to identify which part of the test causes problems in case of a failure. If an error occurs within a service, this could also be made clear visually and thus support the execution of *component tests*. However, *contract tests* represent a special case. If a test consists of a concatenation of several services, the specification of expected intermediate results should be optional in the TDD

environment. Although exact localization of errors might be more difficult in this way, it is within the realm of possibility that within this concatenation the intermediate results may vary, so it might be impossible to cover all existing cases. Nevertheless, to evaluate whether a contract test has been passed or failed, the status in the list of tests must change its color. If one service refers back to another test that was already part of the test before, the visual concatenation should be displayed numerically. Different height levels can be used to indicate skipping one or more services when going forward again after going back.

Besides this main window of the TDD environment, a secondary window is needed that permanently indicates the status of the environment. This window is called *check window* in the following and consists of just one visual information about whether (1) *all* functions are running without errors and *all* tests have passed their last execution without a rebuild being triggered in the meantime; (2) *at least one* malfunction has occurred, regardless of whether it comes from the TDD environment itself or from an implemented test; or (3) *at least one* action such as testing or rebuilding a container is currently in progress, so no definite result can be given. The check window must be visible at all times, because a software developer usually works in the IDE and would not notice any errors if the check window was not visible above all other applications.



(a) (b)

Fig. 4. Utility evaluation of the artifact when (a) a test fails and (b) all tests succeed.

When many services are invoked through synchronous APIs, the MSA inevitably grows into a *distributed monolith* at some point, where the outage of a particular service can cascade to affect the entire architecture and lead to a total failure [31]. In contrast, a central service (i.e., an API gateway) could connect the services by routing requests and serving as a single point of entry [32]. A tightly coupled MSA that contradicts the principle of *loose coupling and strong cohesion* [15] can lead to a construct that Rossi [33] calls a "*microservice death star*". The counter design to this in the form of a gateway service could take the appearance of an asterisk with the gateway service at its center and all available services surrounding it. When a user sends a

request, the gateway service is responsible for receiving the message with the data it contains and forwarding it to the first service in the chain of calls. After the service does its part and sends a response to the gateway service, the processed output is again forwarded to the next service in the sequence. The artifact evaluation is conducted by following the TDD approach in an exemplary scenario. First, a new microservice is added to the docker-compose file and subsequently registered in the TDD environment. A new test is then defined to request a specific function of the microservice. Since the service is newly created, it has no functionality yet, so the corresponding test status icon is colored red. After changing the source code of the microservice to implement the functionality, a process to rebuild the Docker image and the application's container is triggered. When the test is then executed again, the status icon turns green to symbolize that the test has been performed successfully. Fig. 4 displays the result of the exemplary evaluation. On the left side, the interface right after registering the new test involving an also newly defined service is shown. The check window immediately turns red just as the indicator at the entry in the test list and the particular service on the 2D canvas. After a short phase of rebuilding and containerizing the service after the functionality is implemented, the interface changes to the appearance on the right side. This proved the ability of the artifact to enable test-driven microservice development on a local workstation. However, as this is only one possible solution based on the MSA development practices studied, other software solutions are also conceivable.

6. Conclusion and Future Research

Using a clear methodological approach guided by the principles of design science research and systematic literature reviews, this study was able to identify five main principles of MSA development. First, small and specialized services are recommended to enable optimized scalability. Second, a domain-driven design should be followed, where closely related business capabilities are bundled into single services. Third, networks between services should strive for a manageable and clear structure with low complexity and few interconnections. Fourth, networks should also avoid cyclic synchronous dependencies to prevent potential deadlocks. And fifth, generally high connectivity performance should be aimed for in terms of short communication paths and small message sizes. The three test types considered as suitable to be integrated into the artifact were integration testing, component testing, and contract testing. Although the characteristics of microservices differ from those of traditional monolithic software architectures, it can be stated that a test-driven approach can be conceptualized, which was demonstrated on an abstract level by developing a comprehensive artifact that was then evaluated by applying it to a typical use case on the topic.

Reflecting the progress made, two relatively new software engineering concepts have been investigated and successfully combined. The main contribution of this work lies in the provision of a theoretical framework on best practices for microservice development, suitable test techniques on diverse levels of complexity, and a solid technical foundation for developing a corresponding environment for the TDD of MSAs. As generalizable design knowledge, the insights presented can be adopted for the development of more elaborate TDD environments for microservices to achieve a more efficient and higher quality result.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Christian Daase performed the theoretical research and technical development. Daniel Staegemann and Matthias Volk provided the resources for development and methodological guidance. Klaus Turowski supervised the research and reviewed its progress.

References

[1] Sneha, K., & Malle, G. M (2017). Research on software testing techniques and software automation testing tools. *Proceedings of the 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)* (pp. 77–81).

[2] Galimova, E. Y. (2021). Application of software testing methodology based on quality criteria and expert assessments to mobile applications. *IOP Conf. Ser.: Mater. Sci. Eng.*, *1019(1)*, 2002.

[3] Santos, I., Melo, S. M., Lopes, D. S., P. S., & Souza, S. R. S. (2020). Towards a unified catalog of attributes to guide industry in software testing technique selection. *Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 398–407).

[4] Bissi, W., Serra, S. N., Emer, A. G., & CFP, M. (2016). The effects of test-driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, *74*, 45–54.

[5] Choma, J., Guerra, E. M, Da, S. T. S., Albuquerque, T., Albuquerque, V. G., & Zaina, L. M (2019). An empirical study of test-driven development vs. test-last development using eye tracking. *Agile Methods,* Cham: Springer International Publishing.

[6] Ghani, I., Wan, K., Wan, M. N., Mustafa, A., & Babir, M. (2019). Microservice testing approaches: A Systematic literature review. *International Journal of Integrated Engineering*, *11(8)*, 65–80.

[7] AbuSalim, S. W., Ibrahim, R., & Wahab, J. A (2021). Comparative analysis of software testing techniques for mobile applications. *J. Phys.: Conf. Ser.*, *1793(1)*, 12036.

[8] Beck, K. (2003). *Test-Driven Development: By Example*. Boston, Mass., Munich: Addison-Wesley.

[9] Dragoni, N., Giallorenzo, S., Lafuente, A. L, Mazzara, M., Montesi, F., Mustafin, R. *et al.* (2017). Microservices: Yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, Cham: Springer International Publishing.

[10] Zhelev, S., & Rozeva, A. (2019). Using microservices and event driven architecture for big data stream processing. *Proceedings of the 45th International Conference on Application of Mathematics in Engineering and Economics (AMEE'19)* AIP Publishing.

[11] Railic, N., & Savic, M. (2021). Architecting continuous integration and continuous deployment for microservice architecture. *Proceedings of the 2021 20th International Symposium Infoteh-Jahorina*.

[12] Familiar, B. (2015). *Microservices, IoT, and Azure*, Berkeley, CA: Apress.

[13] Bozan, K., Lyytinen, K., & Rose, G. M. (2021). How to transition incrementally to microservice architecture. *Commun. ACM*, *64(1)*, 79–85.

[14] Li, H., Wang, J., Dai, H., & Lv, B. (2020). Research on microservice application testing system. *Proceedings of the 2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)* (pp. 363–368).

[15] Engel, T., Langermeier, M., Bauer, B., & Hofmann, A. (2018). Evaluation of microservice architectures: A metric and tool-based approach. *Information Systems in the Big Data Era*, Cham: Springer International Publishing.

[16] Koschel, A., Astrova, I., Bartels, M., Helmers, M., & Lyko, M. (2021). On testing microservice systems. *Proceedings of the Future Technologies Conference (FTC) 2020* (pp. 597–609).

[17] Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., & Van, H. A. (2018). A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. I *Software Architecture*.

[18] Valdivia, J. A, Lora-González, A., Limón, X., Cortes-Verdin, K., & Ocharán-Hernández, J. O (2020). Patterns related to microservice architecture: A multivocal literature review. *Program Comput Soft*, *46(8)*, 594–608.

[19] Nunamaker, J. F, Chen, M., & Purdin, T. D. (1990). Systems development in information systems research. *Journal of Management Information Systems*, *7(3)*, 89–106.

[20] Hevner, A., R, A., March, S., T, S., Park, Park, J. *et al.* (2004). Design science in information systems research. *Management Information Systems Quarterly*, *28*, 75–105.

[21] Venable, J. R, Pries-Heje, J., & Baskerville, R. (2017). Choosing a design science research methodology. *Proceedings of the ACIS2017 Conference Proceeding*.

[22] Daase, C., Volk, M., Staegemann, D., & Turowski, K. (2022). Addressing the dichotomy of theory and practice in design science research methodologies. *Proceedings of the 17th International Conference on Design Science Research in Information Systems and Technology*.

[23] Vom, B. J., Hevner, A., & Maedche, A. (2020). Introduction to design science research. *Design Science Research. Cases* (pp. 1–13). Cham: Springer International Publishing.

[24] Kitchenham, B., & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering: Technical report. Version 2.3 EBSE Technical Report.

[25] Okoli, C. (2015). A guide to conducting a standalone systematic literature review. *CAIS*, *37*.

[26] Staegemann, D., Volk, M., Daase, C., & Turowski, K. (2020). Discussing relations between dynamic business environments and big data analytics. *CSIMQ(23)*, 58–82.

[27] Zdun, U., Stocker, M., Zimmermann, O., Pautasso, C., & Lübke, D. (2018). Guiding architectural decision making on quality aspects in microservice APIs. *Service-Oriented Computing*, Cham: Springer International Publishing.

[28] Wang, Y., Kadiyala, H., & Rubin, J. (2021). Promises and challenges of microservices: an exploratory study. *Empir Software Eng*, *26(4)*.

[29] Bogner, J., Wagner, S., & Zimmermann, A. (2019). Using architectural modifiability tactics to examine evolution qualities of service microservice-based systems. *SICS Softw.-Inensiv. Cyber-Phys. Syst.*, *34(2-3)*, 141–9.

[30] Joseph, C. T, & Chandrasekaran, K. (2019). Straddling the crevasse: A review of microservice software architecture foundations and recent advancements. *Softw: Pract Exper*, *49(10)*, 1448–1484.

[31] Oliveira, R. H. F. (2022). Defining an event-driven microservice and its boundaries. *Practical Event-Driven Microservices Architecture*, Berkeley, CA: Apress.

[32] Ntentos, E., Zdun, U., Plakidas, K., Meixner, S., & Geiger, S. (2020). Metrics for assessing architecture conformance to microservice architecture patterns and practices. *Service-Oriented Computing*, Cham: Springer International Publishing.

[33] Rossi, D. (2019). Consistency and availability in microservice architectures. *Web Information Systems and Technologies*, Cham: Springer International Publishing.