Software Testing, Verification and Reliability

WILEY

RESEARCH ARTICLE OPEN ACCESS

Ensuring Syntactic Interoperability Using Consumer-Driven Contract Testing

Georg-Daniel Schwarz | Felix Quast | Dirk Riehle

Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany

Correspondence: Georg-Daniel Schwarz (georg.schwarz@fau.de)

Received: 23 December 2023 | **Revised:** 9 May 2025 | **Accepted:** 30 May 2025

Funding: This work was supported by BMBF (Federal Ministry of Education and Research) Software Campus 2.0 project (01IS17045) and by DFG (German Research Foundation) Research Grants Programme (RI 2147/9-1).

Keywords: action research | consumer-driven contract testing | guidelines | literature review | microservices | testing

ABSTRACT

Integrating services in service-based architectures is a major concern and challenge to their developers. A key problem is that today's compilers cannot ensure syntactic interoperability of web APIs. Without further help, invalid calls surface only at runtime. Microservice-based architectures exacerbate this problem due to their use of polyglot software stacks and independent deployments. As a result, maintaining API compatibility with consumers has become increasingly complex. This study presents a systematic literature review on consumer-driven contract testing, a testing technique that ensures syntactic compatibility between microservices through isolated test execution. We develop a theory on when and how to use consumer-driven contract testing to address the problem of syntactic interoperability. We build out our theory with the insights of an action research study, contributing rare empirical data to the field. Our theory posits that consumer-driven contract testing can ensure syntactic interoperability between microservices and complement the testing strategy of such systems. The action research study confirmed this and revealed that introducing consumer-driven contract testing can promote the design and development of higher-quality APIs and code.

1 | Introduction

Microservices are a popular architectural style for building scalable and robust software systems in the cloud. Unlike monolithic applications, which consist of a single coherent entity, microservices split functionality into multiple independent services [1]. Mature microservice-based architectures facilitate independent deployment of each microservice, allowing for independent development life cycles with teams working in parallel [2].

However, the distributed nature of microservices gives rise to unique challenges, like the reliance on communication over an unreliable network instead of in-process communication. By shifting complexity into the integration layer, integration becomes a more predominant and explicit challenge [3].

While monolithic architectures have compilers to prevent syntactic incompatibilities, no equivalent mechanisms exist for integrating microservices.

Traditional testing techniques fall short of fully discovering breaking changes and preventing incompatibilities from reaching production. On the one hand, testing the compatibility of microservices in isolation requires tests on both sides to test against the same interface specification. Classical unit testing would need tests to be manually adapted on both sides to keep

Felix Quast and Dirk Riehle contributed equally.

them in sync, introducing coupling between the microservices. However, microservices emphasize independence in development through loose coupling [4], making this testing approach unsuitable for microservices. On the other hand, integration and system testing can detect syntactic incompatibilities as they require multiple services to interact with each other. However, using these higher-level tests is more expensive and slower regarding feedback time in continuous integration pipelines [5].

In this article, we build a theory of consumer-driven contract testing (CDCT). A theory consists of abstract knowledge to explain a phenomenon or predict outcomes and is primarily a set of (typically interrelated) hypotheses rooted in data. This article presents such a theory in the format of guidelines on when and how to use CDCT in microservice-based projects, a testing technique used to ensure syntactic compatibility between microservices with isolated test execution.

Instead of testing both sides against the same specification, CDCT divides the test into two isolated phases (see Figure 1). The consumer test produces the compatibility specification, encoding the consumer's expectations towards the API codified as a contract file. The provider test uses this contract file to replay the interactions against the API provider and test the compatibility with the consumer's expectations [6].

In this manner, consumers drive the changes in the API contracts between the consumer (the client) and the provider (the service). By aggregating the consumers' expectations towards the API, the provider can implement changes to satisfy these consumer expectations and ensure that changes don't violate them. By decoupling the consumer and provider via the contract file, the CDC tests don't require running the consumer and the provider simultaneously. CDCT poses an alternative to integration testing to discover incompatibilities between microservices [6]. Popular tools to facilitate CDCT are Pact (https://github.com/pact-foundation) and Spring Cloud Contracts (https://github.com/spring-cloud/spring-cloud-contract) [7].

While CDCT is increasingly being adopted by practitioners, especially in the microservice domain, there are only few empirical studies on the matter. This study aims to summarize the current knowledge body and provide further empirical insights into CDCT:

1. We present a systematic literature review on consumer-driven contract testing, giving an overview of the field.
2. We contribute an action research study to the body of CDCT knowledge, addressing its current sparsity in empirical data.
3. Based on the literature and the action research study, we establish a theory on when to utilize consumer-driven contract testing over other mechanisms to ensure syntactic compatibility.
4. Additionally, we develop guidelines on how to implement consumer-driven contract testing.

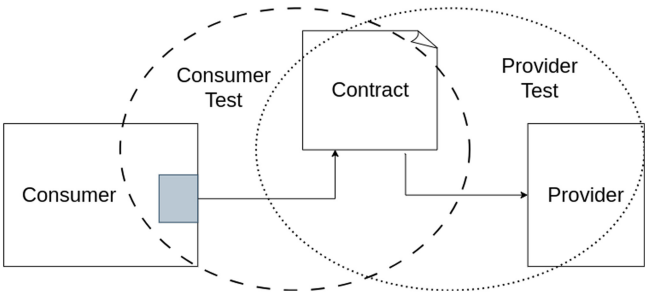


FIGURE 1 | Consumer-driven contract testing (derived from Lehvä et al. [6]).

The remainder of this article is structured as follows: Section 2 positions the article in the related work. Section 3 outlines the applied research design. Section 4 presents the results of the systematic literature review and the action research study. Section 5 discusses the results and outlines future work. Section 6 reflects on the limitations of the applied research methods, and Section 7 concludes the article.

2 | Related Work

In 2006, Robinson [8] published a foundational article on the underlying idea of consumer-driven contracts. Dedicated CDCT tools were introduced several years later, exemplified by the creation of Pact in 2013 or the predecessor of Spring Cloud Contracts called Accurest in 2015. With the rise of the microservice architectural style, the topic was picked up again as integration between microservices is an inherent challenge for these systems. Tools like Pact and Spring Cloud Contracts formed the current understanding of how Robinson's idea can be implemented. Practitioners picked up on the topic, leading to a stream of articles on the topic of CDCT, like the one of Microsoft [9]. Next to the tools' documentation, such articles build the entry door for practitioners to learn and implement CDC tests.

However, the trend has yet to make its way into academia as there are only few articles examining the phenomenon. Contrary to practitioner articles which aim to be actionable and usable, academic studies capture knowledge about a phenomenon with the ambition of avoiding biases, for example, by comparing different views on the topic. Thus, we see merit in aggregating the body of academic knowledge and presenting a cohesive picture of the topic.

We did not find a systematic review that devotes itself to consumer-driven contract testing in detail. While broader systematic literature reviews have addressed related topics, they have not extensively explored the testing technique itself. For example, Ghani et al. [10] systematically reviewed microservice testing approaches and briefly mentioned contract testing as a broader concept without providing detailed insights. Similarly, Waseem et al. [11] mapped the microservice testing literature and briefly discussed contract testing without delving into its specifics. Bogner et al. [12] conducted expert interviews and reviewed grey literature on the evolvability assurance of microservices, where they showcased the relevance of CDCT by presenting it as an evolvability pattern.

Our study extends these higher-level reviews by focusing specifically on the testing technique of consumer-driven contract

testing. By delving into the intricacies of CDCT, we aim to contribute a comprehensive understanding of this specific testing approach.

In addition, the amount of empirical studies on the topic is limited. Ayas et al. [13] performed a post-mortem qualitative analysis of software repositories, shedding light on testing architectures of open-source microservice projects that employed CDCT. Koschel et al. [14] presented CDCT within the context of a comprehensive testing strategy in an example system. Lehvä et al. [6] conducted a case study on consumer-driven contract testing and how it complements an existing testing setup in a microservice-based system.

The action research part of this article takes the same line as the mentioned articles, contributing rare empirical data to the field. However, our study differs by integrating relevant literature to build a scientific theory alongside describing our findings.

3 | Research Design

To build a theory of consumer-driven contract testing, we conducted a systematic literature review (SLR) to answer the following research questions:

RQ1: When to apply consumer-driven contract testing?

RQ2: How to apply consumer-driven contract testing?

The SLR allowed us to gather relevant information from existing studies and identify gaps in the current understanding of the subject. After establishing the theoretical foundation, we applied the initial theory to a software project using participatory action research. This approach allowed us to build out the theory by obtaining real-world insights and feedback. The following subsections provide a detailed explanation of the two methods involved in this research design.

3.1 | Systematic Literature Review

We followed the guidelines of Kitchenham [15] to plan and execute our systematic literature review. Figure 2 depicts an overview of the SLR process. The remainder of this section details the search and selection process. The replication package [16] contains the literature search and selection step results.

3.1.1 | Search Strategy

We employed a search strategy on multiple electronic databases, including Google Scholar, IEEE Xplore, ACM Digital Library, and Scopus to gather scientific articles for our review.

We refined our search strategy through exploratory literature searches. Ultimately, we performed three parallel searches on each data source using the following logical queries (S) in May 2023:

S1: 'consumer-driven contract testing'

S2: 'consumer-driven contract test'

S3: 'cdct' AND 'microservice'

We compiled the search results from each data source and consolidated them into a common list. Throughout this process, we removed duplicate articles, resulting in a pool of 68 potentially relevant articles.

3.1.2 | Study Selection

To select the relevant literature for our review, we established specific inclusion criteria (IC):

IC1: The article must be peer-reviewed academic literature published at a journal, conference, or workshop.

IC2: The article must be accessible in full text to the authors.

IC3: The article must be available in English.

IC4: The article must speak about the advantages, disadvantages, or guidelines of consumer-driven contract testing, or report about experiences with consumer-driven contract testing

We considered only those articles that met all of the inclusion criteria, resulting in a literature pool of ten relevant articles for further analysis. Figure 2 details how each inclusion criteria narrowed down to the final literature pool.

In addition, we conducted one iteration of forward snowballing (by using Google Scholar's 'Cited By' feature) and one iteration of backward snowballing (by looking into the references of the selected articles) to discover literature that may have been missed [17]. We considered articles with relevant titles by applying the

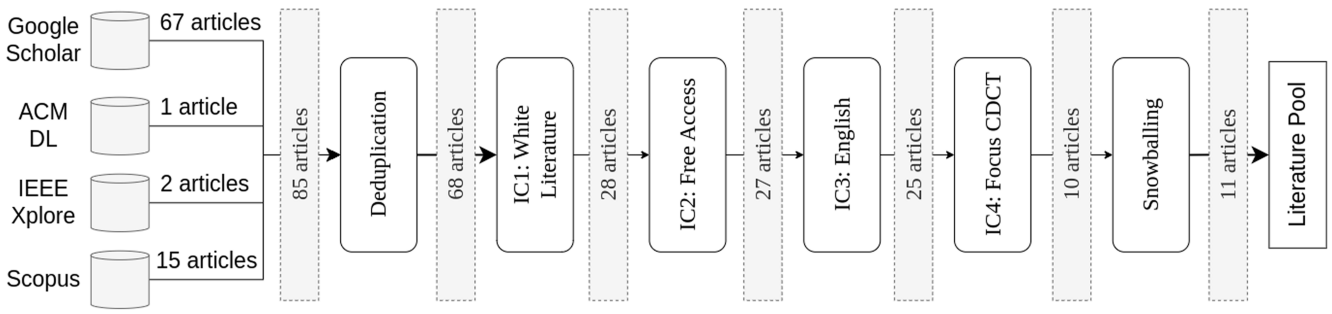


FIGURE 2 | Systematic literature review - process overview.

eligibility criteria to the potentially relevant articles identified through snowballing. We included one additional article in the literature pool.

3.1.3 | Study Quality Assessment

Considering the limited number of articles in our literature pool, we did not conduct a formal quality assessment. However, we considered the research method employed in each article during the data extraction process, as it indicates the reliability of the findings. If conflicts arose during the data synthesis, we considered this information to assess the credibility of conflicting statements.

3.1.4 | Data Extraction

For data extraction, we utilized a structured form to gather the following information for each article:

- Title
- Authors
- Publication year
- Publication outlet
- Publication outlet type (conference, journal, workshop)
- Research method (inferred if not explicitly mentioned)
- Topic

By systematically extracting these details, we aimed to organize and analyse the articles efficiently.

3.1.5 | Data Synthesis

We performed a thematic analysis to distil the qualitative information from the selected articles. The researcher takes an active role in generating codes from the qualitative data guided by the research question [18]. In this analysis, the researcher actively generates codes that capture relevant features of the data and aggregates them into themes. We followed the six-step process outlined by Braun and Clarke [19] for thematic analysis:

1. *Familiarize with the data:* We actively read the primary materials to gain a deep understanding of the data.
2. *Generate initial codes:* We annotated data segments with preliminary codes, ensuring a detailed and contextualized representation.
3. *Search for themes:* We examined the list of codes and explored how they could be combined into cohesive themes. We considered the relationships between codes and themes, organizing them hierarchically.
4. *Review the themes:* We revisited the themes and codes to ensure they accurately represented the dataset. We considered distinguishing criteria for each theme and discussed any ambiguous ones.
5. *Define and name themes:* We gave each theme a working title and carefully examined their relevance to the research question. By incorporating sub-themes, we ensured that the themes were not overly complex or broad.

We utilized the MaxQDA (https://www.maxqda.com/) software to facilitate the coding and theme-building process to maintain the traceability of codes and themes to their sources. The replication package [16] contains exports of the code system. This software helped organize and analyse the qualitative data effectively.

3.2 | Participatory Action Research

The application of action research in this study follows the guidelines provided by Baskerville [20]. By employing action research, the study aims to bridge the gap between academia and industry by putting theoretical concepts into practice [21]. This approach enables active participation in the project, allowing the researchers to introduce changes, observe the effects, and utilize the gained knowledge to adapt both the theory and actions.

The process of the study followed the well-established cyclical model introduced by Susman and Evered [22], which consists of the following phases:

- **Diagnosing:** Identify and choose a significant and relevant problem.
- **Action planning:** Consider alternative interventions for solving the problem.
- **Action taking:** Put the intervention into action and observe its effects.
- **Evaluating:** Study the consequences of the intervention.
- **Specifying learning:** Reflect on the evaluation and identify general findings.

As Kemmis et al. [23] point out, in reality, the phases of this cycle might overlap, and the process is more responsive to change as plans might change due to organizational constraints or new insights. Data from these 'failed' cycles are still valuable as they may improve our understanding of the context in which actions can be applied successfully and the circumstances in which they may not be as effective.

As context for the first action research case, we chose an open-source software project of our research group to gain a deeper understanding of testing microservices and enhance our theory. Section 4.2.1 details the context of the project as a suitable microservice project. The project was developed and maintained by three core developers, the lead author of this study being one of them. The action research study was carried out by a fourth developer, who is also a co-author of this article. This executing developer implemented all the interventions under the close supervision of the core developers. An exception was some CDC tests for dedicated interactions that the core developers implemented towards the end of the study. We captured their experience and opinions with exit interviews. Section 6 discusses the limitations of the convenience sample and potential role conflicts.

3.2.1 | Data Collection

We collected the following three different kinds of data to triangulate the findings:

Developer interviews and observation: We conducted semi-structured interviews with each of the three core developers. We let them implement CDC tests for the same HTTP-based integration for comparability and asked them to share their impressions in interviews. We compared their implementations and their procedures to discover similarities and differences. We provided them with the same materials and documentation and instructed them equally. We created an interview guide with all the relevant questions that helped us to stick to the topic of interest. The interview had four phases: initial training materials, CDCT setup, CDCT implementation, and CDCT in general. Appendix A details the interview guide.

Defect seeding: We applied defect seeding to evaluate how well the CDC tests capture incompatibilities as Lehvä et al. [6] applied it as well. We implanted integration defects into the code and evaluated if the CDC tests discovered them. We considered the following aspects regarding changes that may introduce defects:

- Change on the consumer or provider side
- Change on request, reply, or event
- Change on query parameter (HTTP), path parameter (HTTP), or payload attribute
- If payload change: change on mandatory or optional attribute
- Removal, renaming, addition, increase of value range, or decrease of value range
- Regarding HTTP: status codes, headers, HTTP method, URL changes

We removed impossible changes, for example, changing query parameters of an event, because our messaging mechanism AMQP did not support any query parameters. For all remaining changes, we individually identified if and under which circumstances the changes can produce an incompatibility.

Experiential learnings: By iteratively conducting an action research study, we gained practical experience with our theory. This experience is based on participation in designing and implementing consumer-driven contract testing in the JValue project and contributes to the overall understanding of the topic.

4 | Results

The result of our study is a theory, abstract knowledge aimed at explaining a phenomenon or predicting outcomes, primarily a set of (typically interrelated) hypotheses rooted in data. We present this theory in the format of guidelines on when and how to use CDCT to approach the challenge of syntactic interoperability of microservices.

The first 2 subsections detail the data on which we base our insights. Section 4.1 presents the selected literature while Section 4.2 presents the action research case. Section 4.3 gives an overview of the different advantages, disadvantages, and challenges of CDCT and infers when CDCT is a suitable testing technique to ensure syntactic interoperability. Section 4.4 complements the results with eleven guidelines on how to apply CDCT.

4.1 | Selected Literature

We followed the procedure described in Section 3.1.4 to identify relevant literature for further analysis. Table 1 gives an overview of the eleven selected studies. None of the authors of this study authored any of the selected articles. Figure 3 illustrates the statistics of the selected literature. Figure 3a depicts the distribution of publication years. The publication dates range from 2018 to 2022, with an increase in the number of relevant publications in more recent years. However, the literature sample is too small to infer a trend over time. Figure 3b displays the distribution of publishing outlets for the selected articles. The outlets of the selected literature are very diverse, indicating that CDCT is a topic appealing to a variety of publication outlets. Conference articles dominate the literature pool, accounting for 7 out of 11 articles. Additionally, three articles were published in journals, and one was a workshop paper. Figure 3c provides an overview of the topics covered in the selected articles. Four articles focus on testing in microservice-based systems in general, and another four specifically address contract testing. Two articles by the same author explore architecture visualization, and one explores evolvability aspects.

4.2 | Action Research Procedures

We decided to build out the theory by incorporating participatory action research, as the selected literature contained limited empirical data. The remainder of this section describes the project context and the implementations and observations of tested interactions and defect seeding.

4.2.1 | Project Context

The JValue Open Data Service (ODSv2) project offers ETL-like functionality for open data. The microservice-based architecture was cut according to the different steps of the ETL functionality (Figure 4). The *Datasource Service* is responsible for data extraction from a data source, the *Pipeline Service* for data transformations, the *Query Service* for loading the data into a sink, and the *Notification Service* for notifying clients of newly arrived data. Each microservice has a separate database to govern its data. The monolithic user interface (UI) accesses the backend API via HTTP calls. Traefik (https://www.traefik.io) serves as a facade to the backend, hiding the complexity behind it by forwarding the requests to the suiting microservice based on a fixed set of routing rules. Asynchronous events facilitate communication between microservices. RabbitMQ (https://www.rabbitmq.com) acts as a message broker with publish-subscribe functionality via the AMQP protocol. Instead of sending out events directly, every microservice instance writes into

TABLE 1 | Selected literature.

Ref	Title	Year	Outlet	Outlet type
[7]	Testing for Event-Driven Microservices Based on Consumer-Driven Contracts and State Models	2022	APSEC	Conference
[13]	An Empirical Analysis of Microservices Systems Using Consumer-Driven Contract Testing	2022	SEAA	Conference
[24]	Version-based and risk-enabled testing, monitoring, and visualization of microservice systems	2022	Journal of Software: Evolution and Process	Journal
[14]	On Testing Microservice Systems	2021	FTC	Conference
[6]	Consumer-driven contract tests for microservices: A case study	2019	PROFES	Conference
[25]	Consumer-Driven API Testing with Performance Contracts	2018	Workshops of ESOCC	Workshop
[26]	Design, monitoring, and testing of microservices systems: The practitioners' perspective	2021	JSS	Journal
[27]	Using service dependency graph to analyse and test microservices	2018	COMPSAC	Conference
[12]	Industry practices and challenges for the evolvability assurance of microservices: An interview study and systematic grey literature review	2021	EMSE	Journal
[28]	Research on Microservice Application Testing System	2020	ICISCAE	Conference
[5]	A Test Concept for the Development of Microservice-based Applications	2021	ICSEA	Conference

an outbox table in their database. A separately deployed outbox container sends all events in the outbox table at least once to the message broker. This implementation ensures consistency between different network interactions (writing into a database and sending out events). This pattern is known as the transactional outbox pattern (https://microservices.io/patterns/data/transactional-outbox.html).

All microservices share one common code repository. A core team of three employed developers led the project, with several students participating and supporting the project over time in the form of theses or university projects and seminars. Table 2 gives an overview of the project members at the time of the action research study. As is typical for university projects, the employed core developers allocated varying amounts of time to the project. Student 1 was the executing researcher in this part of the study, who actively participated in the project and introduced CDCT. The three core developers all participated in implementing a subset of the consumer and provider tests and shared their experiences and opinions in the exit interviews. Student 2 did not actively participate in the study but worked on DevOps topics like the deployment to Kubernetes in parallel.

The project facilitated a continuous integration (CI) pipeline with three phases. Phase 1 lints, unit tests, and executes isolated black-box tests on every microservice validating functional requirements. Phase 2 executes functional system tests against a complete backend deployment with the API facade. Phase 3 publishes a Docker container image per microservice. A docker-compose file allows easy deployment.

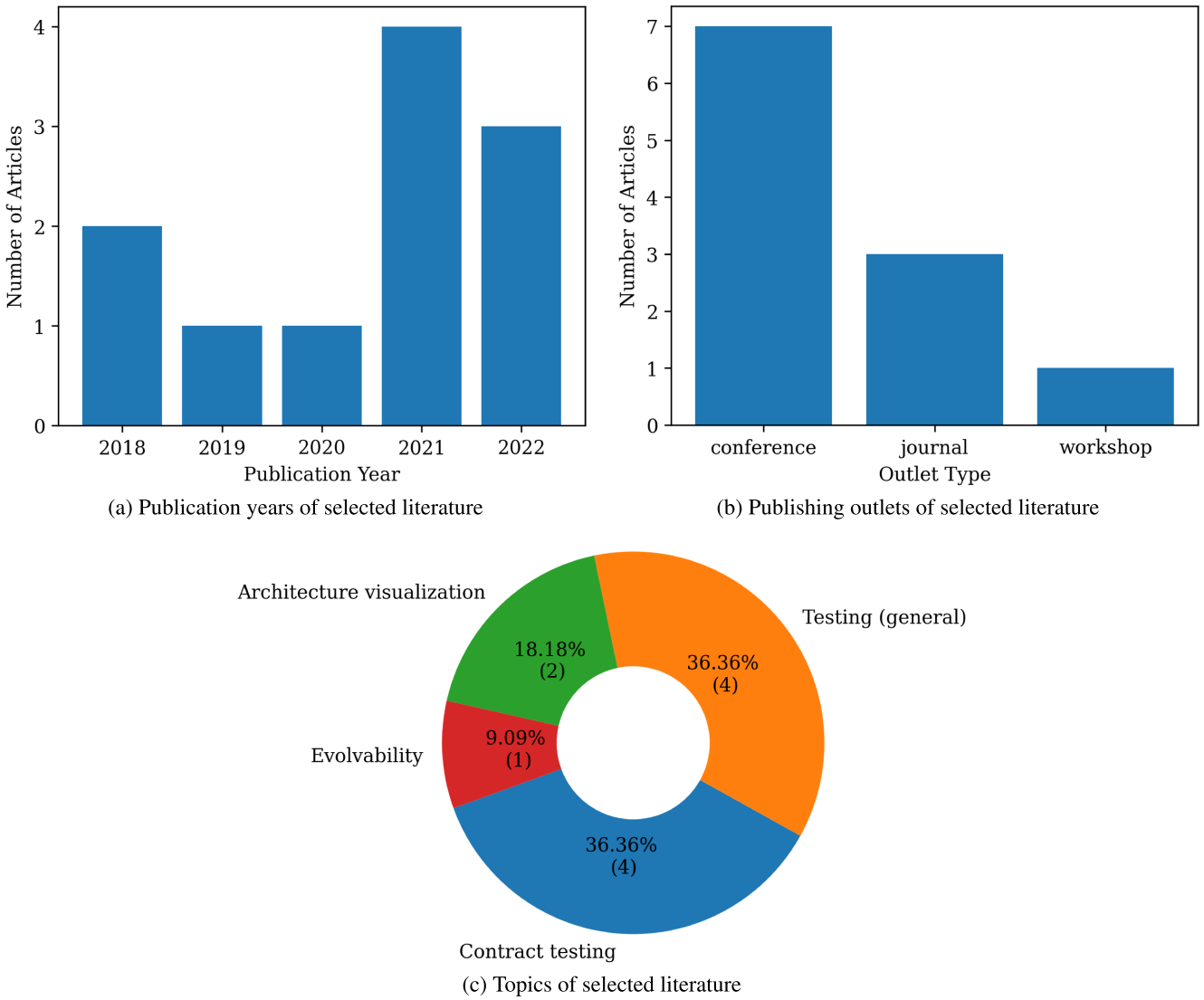
4.2.2 | Iterations

We applied nine iterations over a period of three months. Table 3 describes the underlying problem and the planned action of each iteration. The replication package [16] contains code diffs of each action research iteration. The executing researcher and one or more core developers determined each intervention in a joint decision in regular subject-specific meetings. The executing researcher advised and mainly executed the interventions. Each successful iteration that solved the initial problem of the iteration ended with a pull request.

4.2.3 | Implementation

We used the Pact (https://pact.io/) library to implement consumer-driven contract testing, integrating it into our existing DevOps workflow. We aligned the test execution to the existing testing setup by providing npm scripts to execute consumer and provider tests. Additionally, we added the consumer tests to the build steps in the CI pipeline (Figure 5). We introduced the provider tests to the CI pipeline on the same level as the system tests, allowing parallel execution and maintaining pipeline efficiency.

This integration reflects a practical application of DevOps principles by ensuring fast feedback cycles, continuous validation, and automation of quality checks across service boundaries. To validate the provider API against the consumers' expectations, we had to transmit the encoded expectations - the contract files that are created by the consumer tests



(a) Publication years of selected literature

(b) Publishing outlets of selected literature

(c) Topics of selected literature

FIGURE 3 | Statistics of selected literature.

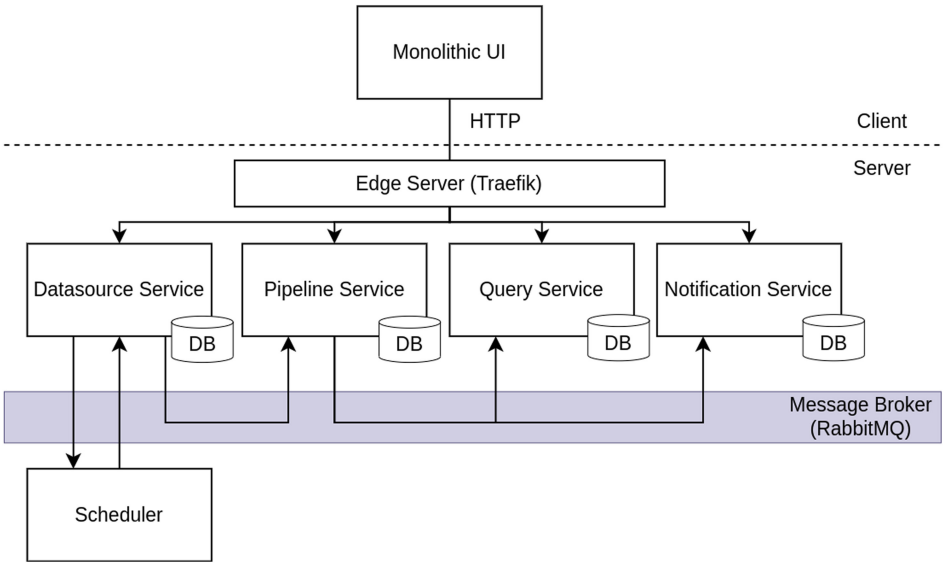


FIGURE 4 | JValue ODS architecture.

- to the provider tests. This step was especially challenging and required two attempts to find a satisfying solution. First, we added the consumer contracts to the version control via automated commits. Later, we moved to a CI feature called *artefacts*in the CI pipeline itself, making the consumer contracts available in later CI steps. This approach relies on all microservices being maintained in one common repository. We also considered using the Pact Broker as a contract broker. Ultimately, we decided against it to keep the solution as simple as possible.

TABLE 2 | Project members at the time of the action research study.

#	Education level	Weekly time in project
CoreDev 1	Master's degree	25h (PhD candidate)
CoreDev 2	Master's degree	15h (Developer)
CoreDev 3	Bachelor's degree	5h (Student worker)
Student 1	Bachelor's degree	30h (Master thesis)
Student 2	Bachelor's degree	30h (Master thesis)

4.2.4 | Tested Interactions

HTTP-based Interactions: We tested three HTTP-based integrations: UI and Pipeline Service, UI and Query Service, and UI and Notification Service. In total, we discovered four existing

TABLE 3 | Action research iterations at JValue.

#	Problem	Action
1	Missing CDCT setup	Introduce minimal example, automated commits for consumer contracts (for CI)
2	HTTP interactions between UI and Pipeline Service untested	Write corresponding CDCTs
3	AMQP interactions between Notification and Pipeline Service untested	Write corresponding CDCTs
4	CDCT setup not mature	Add convenience scripts, CI artefacts instead of automated commits to pass contracts to providers
5	HTTP interactions between UI and Query Service untested	Write corresponding CDCTs
6	AMQP interactions between Pipeline and Query Service untested	Write corresponding CDCTs
7	CDCT Docker containers download CDCT dependencies with every build	Optimize Docker image by reordering the layers
8	Optional attributes are not covered	Optional attributes must be at least once present and once missing
9	Inconsistencies of status codes and response payload on update and delete endpoints	Include changed or deleted resource in response, make status codes consistent

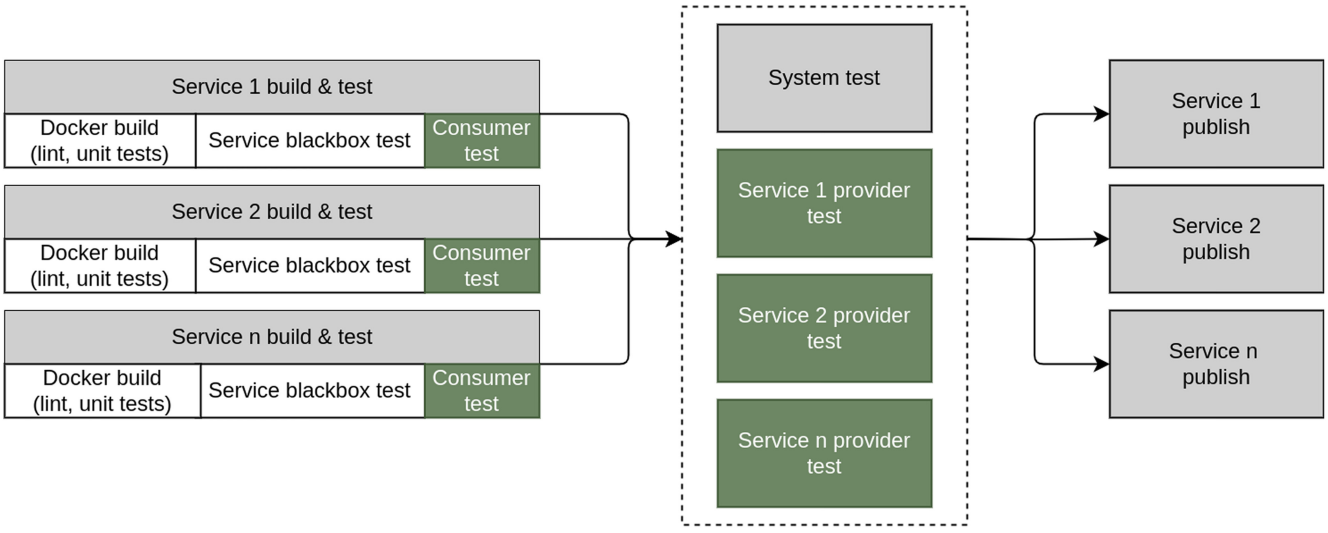


FIGURE 5 | JValue ODS continuous integration with CDCT (added steps are coloured green).

integration issues during the implementation. The consumer and provider assumed different value ranges on some JSON attributes in two of the four integration issues. The other two issues were based on discrepancies if some JSON attributes were optional or mandatory.

AMQP-based Interactions: Pact provides a generic API for testing asynchronous communication that enables the extension of the CDC tests to event-based messaging. However, due to the transactional outbox pattern, we could not use the library, leading to a more complex test setup. We had to start the database, the outbox service, and the AMQP broker to create the contracts. We tie this complexity to other architectural design decisions like the transactional outbox pattern rather than to testing message-based interactions per se. We discovered no integration issues.

4.2.5 | Defect Seeding

We implanted integration defects into the application to evaluate the effectiveness of the written consumer-driven contract tests. Table 4 shows the possible consumer-side integration defects we systematically identified. We artificially added such defects to check whether CDCT would be able to catch those defects. Out of the nineteen seeded consumer-side defects, the tests identified eleven. Five more of those could have been identified if we had additional consumer-side interactions that took such a case into account, increasing the potentially revealed defects of this category to 16. CDCT could not reveal the remaining three defects, which revolved around a decreasing value range in JSON attributes.

Table 5 shows the integration defects we systematically identified on the provider side. We artificially added such defects to the code and observed which of them CDCT can reveal. 14 of the 23 seeded provider-side defects could be detected, while the remaining nine could not be accommodated with the tests. Here as well, changes to the value ranges of attributes and parameters constitute the cases that CDCT was not able to reveal. For many of the other interactions, an assumption we made was that at least one consumer uses the corrupted query parameter or JSON attribute.

TABLE 4 | Systematically derived consumer-side integration defects used for defect seeding.

			Operation	CDC
HTTP request	Query param	opt	Rename	✓
			Incr value range	✓
	Path param		Incr value range	✓
	JSON attr	opt	Rename	✓
		required	Delete	✓
			Rename	✓
			Incr value range	✓
HTTP response	JSON attribute	optional	Rename	✓
			Add	✓
			Decr value range	✗
		required	Rename	✓
			Add	✓
			Decr value range	✗
Async message	JSON attribute	optional	Rename	✓
			Add	✓
			Decr value range	✗
		required	Rename	✓
			Add	✓
			Decr value range	✓

✓ at least one test case revealed the defect
✓ an artificially added consumer-side interaction revealed the defect
✗ the defect was not revealed

Table 6 summarizes further integration defects we investigated in the context of HTTP-based interactions, focusing on higher-level structural changes such as changes to the HTTP headers or URL paths. The CDC tests were able to reveal all those eleven defects, although frequently under the assumption that consumers and providers pay attention to headers and HTTP endpoints that are actually consumed.

In total, we seeded 53 defects. The CDC tests detected 41 of the overall 53 seeded defects (77%). 11 of the 12 undetected defects have in common that they involve changes to the value range of attributes, query, or path parameters. We ground this observation in the fact that consumer-driven contract tests only spot-check single values in the value range. From the presented quantitative data, we derived the insights presented in the following subsections.

4.3 | Advantages, Disadvantages, and Challenges of CDCT

To better understand when to use consumer-driven contract testing, we analysed the selected literature and the primary materials from the action research study for advantages (A), disadvantages (D), and challenges (C) of CDCT:

A1 **Test isolation:** The consumer and provider part of the test can be executed in isolation, leading to more stable and deterministic test environments and faster test execution with a faster feedback cycle [5, 6, 13, 28].

All three interviewed developers of the action research study confirmed a faster test execution than the service black-box tests and a faster feedback cycle.

A2 **Disclosure of interface incompatibilities (replacing integration tests):** CDC tests examine if contents provided by an API provider conform to the expectations of consumers. API changes can be evaluated in this fashion if they break consumers' expectations, exposing incompatible interfaces [5–7, 12, 14, 25, 26, 28].

Due to a similar scope, CDCT can (partially) replace integration tests [5, 6].

The defect seeding of the action research study backed up this advantage with quantitative assessment: 41 of 53 seeded incompatibilities were caught by the implemented CDC tests.

A3 **Awareness of consumers:** If applied consistently throughout the project, each microservice is aware of all its consumers [6, 25]. This knowledge simplifies coordination and impact analysis of API changes.

TABLE 5 | Systematically derived provider-side integration defects used for defect seeding.

			Operation	Precondition	CDC
HTTP request	Query param	optional	Delete	>= 1 consumer uses the query param	✓
			Rename	>= 1 consumer uses the query param	✓
			Decr value range	>= 1 consumer uses the query param	✗
	Path param		Decr value range		✗
	JSON attribute	optional	Delete	>= 1 consumer uses the JSON attribute	✓
			Rename	>= 1 consumer uses the JSON attribute	✓
			Decr value range	>= 1 consumer uses the JSON attribute	✗
		required	Delete		✗
			Rename		✓
			Add		✓
			Decr value range		✗
HTTP response	JSON attribute	optional	Delete	>= 1 consumer expects the JSON attribute	✓
			Rename	>= 1 consumer expects the JSON attribute	✓
			Incr value range	>= 1 consumer expects the JSON attribute	✗
		required	Delete	>= 1 consumer expects the JSON attribute	✓
			Rename	>= 1 consumer expects the JSON attribute	✓
			Incr value range	>= 1 consumer expects the JSON attribute	✗
Async message	JSON attribute	optional	Delete	>= 1 consumer expects the JSON attribute	✓
			Rename	>= 1 consumer expects the JSON attribute	✓
			Incr value range	>= 1 consumer expects the JSON attribute	✗
		required	Delete	>= 1 consumer expects the JSON attribute	✓
			Rename	>= 1 consumer expects the JSON attribute	✓
			Incr value range	>= 1 consumer expects the JSON attribute	✗

✓ at least one test case revealed the defect
✗ the defect was not revealed

TABLE 6 | Derived HTTP-based integration defects used for defect seeding.

	Change	Precondition	CDC
Consumer-side	Change the HTTP response status code		✓
	Delete an HTTP header	Provider expects the header	✓
	Rename an HTTP Header	Provider expects the header	✓
	Change URL of an API call		✓
	Change the HTTP method of API call		✓
Provider-side	Change the HTTP response status code	Consumer expects the response status code	✓
	Delete an HTTP header	Consumer expects the header	✓
	Rename an HTTP header	Consumer expects the header	✓
	Delete an HTTP endpoint	Consumer uses the HTTP endpoint	✓
	Change URL of an HTTP endpoint	Consumer uses the HTTP endpoint	✓
	Change the HTTP method of an HTTP endpoint	Consumer uses the HTTP endpoint	✓

✓ at least one test case revealed the defect
✗ the defect was not revealed

A4 **Provider evolution based on actual consumer needs:** CDC tests inform API providers how each consumer uses their API. This knowledge enables API providers to drive the API by the actual business needs of consumers [6].

A5 **Contracts as communication tools between teams:** CDCs can serve as a communication medium between different development teams making consumer expectations explicit [6].

A6 **Improved API and code design:** The developer interviews of our action research study highlighted that adopting CDCT can lead to a better API and code design by taking the consumers' perspective. We found evidence for that in one developer reporting the redesign

of editing operations not to require a fetching operation afterward and another reporting the refactoring of the consumer for better testability via dependency injection. Two developers mentioned adopting the robustness principle [29] by ignoring all fields in transmitted models that the consumer does not need.

A7 **Adapter testing of UIs:** We infer from the experience in our action research study that using the user interface as a test driver for integration and system tests in an automated way is expensive and, thus, not commonly adopted. CDCT, however, enables simplified testing of the compatibility of the user interface with backend services as the consumer test can be implemented similarly to a unit test to run in isolation (see A1).

D1 **No test of functional behaviour:** CDC tests cannot validate the functional behaviour of consumers. Only those parts of consumer code are tested that execute the API calls [6].

Two interviewed developers of the action research study reported that CDCT only validates interoperability but no functionality. They could not replace the service black-box tests but only complement them.

D2 **No application to external systems:** CDCT requires coordination with the integration counterpart [6]. If the integration counterpart is an external system without control over it, implementing CDCT is not possible [13]. Waseem et al. [26] report the extensive use of third-party resources in microservice systems as an impeding factor for adopting CDCT.

C1 **Lack of adoption:** Bogner et al. [12] list CDCs as the third most common evolvability pattern, while Waseem et al. [26] found that CDCT is used sparingly. The tooling ecosystem seconds their perception with the frequent mention of only two tools: Pact (https://pact.io/) and Spring Cloud Contract (https://spring.io/projects/spring-cloud-contract).

As an implication, the restricted choice of tooling might be a hurdle for the adoption of CDCT.

C2 **Learning curve as entry barrier:** Lehvä et al. [6] report that the majority of time in adopting CDCT was spent learning and researching the testing technique. Learning another technology might pose an entry barrier for CDCT.

Two interviewed developers of the action research study stressed an initial training effort. The same two developers estimated the effort to write CDC tests larger than unit tests but smaller than service black-box tests, while the third developer found it the most effort of the three.

C3 **Effort majorly on consumer side:** The CDCT implementation on the consumer side takes more effort than on the provider side [6]. This imbalance has to be considered in resource planning when adopting CDCT.

C4 **Rising complexity with the increasing number of services:** Checking the test results with an increasing number of inter-service connections becomes inconvenient and prone to error [24].

C5 **Long-running transactions across multiple services:** Long-running transactions over multiple microservices make CDCT more challenging and complex [7].

C6 **Communication obstacles with larger team sizes:** Waseem et al. [26] identified the communication obstacles caused by an increased team size per microservice as one of the reasons for the lower adoption of CDCT in their examined sample. We logically infer that larger team sizes might pose a challenge to CDCT. However, fully understanding this challenge requires further empirical data.

C7 **Contract exchange:** The transmission of consumer contracts to the provider tests is an inherent challenge to CDCT [6].

The action research study showcases that this challenge applies not only to multi-repository setups. Two developers listed the contract exchange as a challenge, mainly since solving it might introduce additional complexity, like operating a contract broker.

C8 **Testing different code versions:** We infer from the open point of Schneider et al. [5] to enable CDCT on different branches that testing different code versions and keeping track of the version compatibilities is a further challenge.

C9 **Uncovered changes in validity ranges of parameters:** The defect seeding results of our action research study show that changes in the validity range of attributes or parameters are challenging to discover with consumer-driven contract tests.

These findings constitute a significant part of the theory we built of CDCT and support us in generating new hypotheses. Among others, we can set the disadvantages and challenges in relation to the advantages to reason about when CDCT should be applied.

Answer to RQ1: When to apply CDCT?

Use consumer-driven contract testing to address the aforementioned problems when

a) the pool of API consumers is limited,

b) the API consumers and their needs should drive the APIs of the providing microservices (instead of API providers specifying the API by themselves),

c) isolated tests are favoured over integration tests, and

d) resources are given to learn, set up, and apply the testing technique consistently.

4.4 | CDCT Guidelines

Similarly, we derived guidelines from the systematic literature review and the action research findings. We refrain from giving in-depth guidance on how to apply consumer-driven contract testing on a technical level, as the technology used significantly

impacts how consumer-driven contract tests might be written. Instead, we highlight generic guidelines that apply in broader project contexts independent from the tool used to implement consumer-driven contract tests. We arrange the guidelines into the categories of adoption guidelines (GA), implementation guidelines (GI), and coordination guidelines (GC). Please note that we did not collect enough data on the coordination guidelines to assert their usefulness in the action research study.

Answer to RQ2: How to apply CDCT?

GA1 **Adopt CDCT incrementally:** Adopt CDC tests in an iterative way to discover impediments early. Start with a prototype and incorporate process decisions into the adoption, like how to get the coordination between teams right [6].

The action research findings confirm the usefulness of this guideline. The nature of our study design iteratively led to an incremental adoption of CDCT. Before implementation in its full breadth, we devoted the first four iterations to designing a vertical prototype. This prototype included the test setup with CI pipelines and examples for all communication technologies used between services (HTTP and AMQP).

GA2 **Embed CDCT into the testing architecture:** Complement CDC tests with other established testing measures to build a comprehensive test strategy or architecture [6, 13, 26, 28].

The action research findings confirm the usefulness of this guideline. We added consumer-driven contract tests to the existing testing setup. The CDC tests took a complementary role by focusing on service interoperability, while the service black-box tests focused on functionality, and the system tests acted as smoke tests.

GA3 **Use the adoption of consumer tests to improve the code quality of the consumer:** Leverage consumer tests to test the consumers' adapter logic to increase test coverage and improve code quality. With traditional testing approaches, such adapter tests on consumers are frequently overlooked due to the high effort required for integration testing. This guideline is based on the observations and experiences of the action research study.

GA4 **Use the adoption of consumer tests to enhance the API quality:** Writing consumer tests implicates taking the perspective of consumers and allows to detect inconsistencies in the API design. Adopt the robustness principle, also known as 'Postel's Law' [29], by curating the parsed content of API payloads to the content that the consumer really uses. This guideline is based on the observations and experiences of the action research study.

GI1 **Use a contract broker to decouple the test execution from the location of the microservice's repository:** A contract broker is the most advised solution to the challenge of exchanging contract files (C7) [5–7, 13, 25, 27]. Utilizing a hosted contract broker makes the mechanism to exchange contracts independent of whether all microservices share or are distributed over multiple code repositories.

The action research findings confirm the usefulness of this guideline. We decided against using a contract broker due to the mono-repo setup. We experienced some pain points with this setup, for example, the inefficiency of repeating test execution, although the contract did not change. With this experience made, we recommend using a contract broker instead. The Pact ecosystem provides an Open Source contract broker.

GI2 **Add CDCT to continuous integration pipelines:** CDCT execution should be part of the Continuous Integration pipeline [12, 28]. In combination with GA2, the priority of different types of tests should be discussed based on the testing pyramid [13]. Schneider et al. [5] showcase a non-trivial CI setup and elaborate on how a contract broker (GI1) is embedded in this process.

The action research findings confirm the usefulness of this guideline. As described above, we added the CDCT execution to the CI pipelines. The mono-repo setup allowed us to facilitate the contract exchange via a cache feature between different CI steps. Using CDCT as regression tests allowed us to detect breaking API changes with every code change integration.

GI3 **Develop a strategy to uncover missing contract tests:** We advise developing a strategy to identify test cases systematically. Lehvä et al. [6] report using HTTP status codes to distinguish interactions under test for HTTP-based interactions, while Wu et al. [7] use a state model in the context of an event-driven architecture.

The action research findings confirm the usefulness of this guideline. For HTTP-based interactions, we treated every status code as a different interaction. In some cases, we tested multiple interactions per status code, mainly to test the use and omission of optional attributes. For AMQP-based interactions, we treated every event topic as an interaction. This strategy provided us with a consistent and helpful way of identifying test cases.

GC1 **Communicate consumer needs via contracts:** Use consumer contracts to suggest API changes. Only proceed with the consumer change once the provider test ensures compatibility [6].

GC2 **Communicate breaking provider changes to consumers:** Establish a coordination process to announce changes in the provider API if the API provider has to make a breaking change [6]. While this guideline might be of general use to microservice-based projects, it is essential for CDCT. Consumers can adapt their contracts accordingly before the provider evolves as planned.

GC3 **Visualize CDCT results in a service graph:** The existing information on API consumers and providers can be utilized to visualize a service dependency graph (SDG) [6]. Ma et al. [24, 27] present approaches to model test results in SDGs to efficiently detect compatibility anomalies.

GC4 **Communicate changes in the value range of attributes and parameters or cover them by functional tests:** Changes in the value range of attributes or parameters might not be caught by CDC tests, especially if the robustness principle is not adopted correctly by consumers. This guideline is based on the observations and experiences of the action research study, especially the defect seeding.

5 | Discussion

Sections 4.3 and 4.4 outline the CDCT theory we have built in this study. The primary focus of this theory is to address the initial research questions: (RQ1) when to apply CDCT and (RQ2) how to apply CDCT. Practitioners can use the results of this study for an easy and fast evaluation of whether CDCT fits their context and can accomplish their goals. If they adopt CDC testing, they can use the actionable guidelines to successfully apply CDCT, avoid common pitfalls, and reap its benefits. Researchers can use the SLR presented in this paper to get an overview of the field. Further, they can use the empiric insights from the action research study and the theory itself to build new hypotheses, extend the work, and evaluate it.

The theory is still in its early stages. We built a first version of the theory using a systematic literature review. Subsequently, we augmented this theory through a participatory action research study aimed at strengthening and broadening the findings from the literature. Action research is a well-suited research method to build out an early theory in an evolving research field. Notably, our base of existing publications was sparse, which underlines the importance of triangulation via action research. We intend to conduct follow-up work utilizing case study research, an approach well-suited for refining theories before progressing to comprehensive (albeit costly) hypothesis testing.

The limited academic literature we found during our systematic review indicates that CDCT is still a niche topic, a very new topic, or both.

The recency of publications in 2021 and 2022, as illustrated in Figure 3a, points to the nascent stage of CDCT adoption. The number of papers is too small to speculate about a possible growth trajectory. However, we observed that 24 out of the 85 articles were student theses, indicating that multiple research groups are currently exploring the topic. Further, we noticed increasing grey literature on the topic, such as blog articles. While these sources can contribute to the popularity of CDCT, it is imperative to acknowledge their limitations from an empirically founded research perspective. Hence, we advocate for future research using appropriate empirical research methods. We suggest to further building out our theory, for example, with the following research designs:

- qualitative surveys and interview studies involving experienced CDCT practitioners,
- repository mining to delve into the technical facets of CDCT,
- case studies and action research on CDCT-adopting projects to evaluate the findings, and
- theory validation studies with questionnaires or controlled experiments.

Our utilization of participatory action research facilitated such an acquisition of additional empirical data, enabling triangulation with findings from the literature. The resulting theory is open to further augmentation and evaluation through subsequent action research studies and case study research.

The theory we present is partial by nature, as it is centred around addressing the research questions posed. When presenting a theory in the form of guidelines, an important quality criterion is to ask whether they are mutually exclusive and collectively exhaustive. Our guidelines achieve mutual exclusivity through disciplined qualitative analysis. However, due to the early stage of the theory, the guidelines are not yet exhaustive. While incremental thematic analysis led to a reduction in code system changes, we cannot claim theoretical saturation, resulting in gaps within the theory. Consequently, our analysis suggests the need for more extensive data collection in future studies, particularly concerning whether and how contracts can be used as communication tools between teams.

Further, this study brought CDCT into relation to DevOps practices in several places. For instance, the action research procedure in Section 4.2 illustrates how continuous integration pipelines can accommodate consumer and provider tests. An explicit DevOps challenge is the exchange of contract files between consumer and provider tests. This can be facilitated by passing contract files between CI pipeline steps in simple scenarios like our action research case or by operating a contract broker in more sophisticated projects (see guideline GI1). CDCT may contribute to DevOps' observability practices, such as providing insights into integration visualizations (see guideline GC3). We encourage future research to investigate the relation to DevOps practices. Particularly, the topics of how complex DevOps workflows, such as deployment to multiple environments, can be supported by CDCT and how agile methods might play together with the consumer-driven API evolution combined with consumer-driven contracts as a communication medium remain aspects for future work.

Future work may also research how to cover changes in value ranges of attributes or parameters beyond actively communicating those (see guideline GC4). Inspirations from other testing techniques, such as property-based testing, automated boundary value analysis, and parameterized tests, might be able to enhance CDCT or complement the testing strategy in this regard.

Additionally, we propose constructing rival theories for other approaches addressing the syntactic interoperability challenge posed by microservices. One such approach uses client code generators, utilizing API specifications such as the OpenAPI Specification (https://spec.openapis.org/) for generating code across multiple programming languages, enabling the technological independence of microservices. A broader theoretical framework could discern differences between approaches towards the syntactic interoperability challenge, their potential

coexistence within projects, and guidelines for practitioners to select the most suitable approach based on their project context.

6 | Limitations

While this study significantly contributes to the understanding of CDCT in research, we acknowledge certain limitations in our approach.

First, the selected literature may be subject to publication bias, only including peer-reviewed publications in English. While relying on such high-quality data is a strength of our approach, this might have led to missing relevant work. To mitigate this, we triangulated the findings with an action research study, incorporating firsthand empirical insights complementing the analysed literature.

Second, using a single case in the action research study limits the generalizability of the findings. We mitigate this by using the action research study as an evaluation to refine the theory we first built on the literature, not as the only source of evidence. Further, our theory constitutes a set of hypotheses we draw from the data. To make statistically significant claims, follow-up studies are required to validate those hypotheses.

Third, the analysed materials might not fully cover the topic, leaving holes in our theory. We acknowledge that further research is necessary to complement our findings, particularly to add nuanced insights to the results. We tracked the changes in our code system throughout the analysis (see Figure 6). The diminishing number of new and changed codes indicates the proximity to theoretical saturation, indicating a comprehensive understanding of the phenomenon under study. The replication package [16] contains the tracked code system changes.

FIGURE 6 | Changes to the code system over time.

TABLE 7 | Peer debriefings.

Date	Focus
2021-08-13	Determination of the research method
2021-08-24	Planning of the first two iterations
2021-10-28	Preparations for the developer interviews
2021-11-12	Planning of training of developers (to prepare the interviews)

Fourth, we acknowledge potential bias in the action research study due to a role conflict. One of the project's core developers is the lead author of this article, who supervised the executing researcher of this part of the study in their master's thesis. We adopted a stringent approach in our data analysis for mitigating this potential bias. We considered insights and findings valid only when at least two interviewees independently confirmed them. This stringent validation process aimed to enhance the confirmability of the study's results by ensuring that the perspective of a single individual did not solely influence the findings but was instead supported by multiple independent sources.

In addition to these individual mitigation measures, we regularly engaged in continuous professional exchange through peer debriefing sessions among the co-authors to '[...] confirming that the findings and the interpretations are worthy, honest, and believable' [30]. The peer debriefing sessions served as a means to mitigate potential biases and enhance the confirmability of the study. Table 7 provides an overview of the peer debriefing sessions, including their respective focus areas. Moreover, investigator triangulation was employed by having two researchers independently analyse specific parts of the selected literature and discuss differences, further enhancing the confirmability of the study by reducing potential biases.

Despite the limitations, the study offers valuable contributions by addressing an understudied phenomenon in microservice-based projects. Our study serves as a starting point for future research to strengthen, complement, and broaden the presented results.

7 | Conclusion

Consumer-driven contract testing has emerged as a promising testing technique to address the challenges of syntactic interoperability in microservice-based architectures. Through a systematic literature review with 11 selected articles and an action research study with defect seeding, we have made significant strides in understanding when and how to apply CDCT effectively.

Based on our research, we recommend utilizing CDCT when certain conditions are met: (a) the pool of API consumers is known and limited; (b) the needs of the API consumers should drive the APIs of other microservices; (c) there is a preference for isolated tests over integration tests; (d) and sufficient resources are available for learning, setup, and consistent application of the testing technique. If these conditions apply, organizations can leverage CDCT to ensure syntactic compatibility between microservices. CDCT promotes faster feedback iterations than classic integration testing and supports improving the quality of APIs and consumer code.

The four adoption guidelines we have identified offer valuable insights for teams seeking to start out with CDCT by adopting CDCT incrementally (GA1), embedding CDCT into a testing architecture (GA2), and improving the consumers' code quality and the providers' API quality in one go (GA3, GA4). Moreover, the three implementation guidelines provide practical steps to

set up CDCT effectively, considering factors like the contract exchange with a contract broker (GI1), the automated test execution with continuous integration pipelines (GI2), and the discovery of missing contract tests (GI3). The four communication guidelines emphasize the importance of establishing clear communication channels between microservice teams to facilitate effective contract definition and maintenance. Contracts should convey consumers' needs (GC1) and be visualized in service graphs to detect anomalies (GC3). Providers should communicate breaking changes (GC2), while changes in the value range of attributes or parameters should be voiced in general (GC4).

While our research significantly contributes to the understanding of CDCT, we also identified a notable gap in empirical data on CDCT in the existing academic literature. To address this gap, we recommend future research endeavours to collect data through robust research methodologies, such as qualitative surveys, case studies, and action research studies. Gathering empirical evidence will strengthen the validity and applicability of CDCT in real-world scenarios and provide valuable insights into its impact on API development and microservice integration.

In addition to filling the gap in empirical data, we see an opportunity for future work in comparing CDCT with client code generation as a potential competing solution to the interoperability problem. Exploring the strengths and limitations of each approach will shed light on their suitability for different use cases and project contexts. Such a comparative analysis will further enrich the existing knowledge base and enable organizations to make well-informed decisions on how to ensure syntactic interoperability in their microservice-based architectures.

In conclusion, CDCT represents a promising approach to address the challenges of syntactic interoperability in service-based architectures. Our research has provided valuable insights into its adoption and implementation. As the field continues to evolve, further research with empirical data and comparative analysis has the potential pave the way for broader adoption of CDCT and its success in facilitating seamless integration and high-quality API development within microservice architectures.

Acknowledgements

The present work was performed in partial fulfillment of the requirements for a cumulative dissertation. We thank our colleagues for their continuous feedback and proofreading of this article. During the preparation of this work, the authors used ChatGPT in order to improve and rephrase written paragraphs. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication. Open Access funding enabled and organized by Projekt DEAL.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The datasets generated during the current study are available in a replication package [16] on Zenodo: https://zenodo.org/records/15364770

References

1. P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The Journey so Far and Challenges Ahead," *IEEE Software* 35, no. 3 (2018): 24–35.

2. S. Newman, *Building Microservices* (O'Reilly Media, Inc., 2021).

3. S. Baskarada, V. Nguyen, and A. Koronios, "Architecting Microservices: Practical Opportunities and Challenges," *Journal of Computer Information Systems* 60, no. 5 (2020): 428–436.

4. N. Dragoni, S. Giallorenzo, A. L. Lafuente, et al., "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering* (ChamSpringer, 2017), 195–216.

5. M. Schneider, S. Zieschinski, H. Klechorov, et al., "A Test Concept for the Development of Microservice-Based Applications," in *2021 International Conference on Software Engineering Advances (ICSEA)* (IEEE, 2021), 88–97.

6. J. Lehvä, N. Mäkitalo, and T. Mikkonen, "Consumer-Driven Contract Tests for Microservices: A Case Study," in *International Conference on Product-Focused Software Process Improvement* (Springer, 2019), 497–512.

7. C. F. Wu, S. P. Ma, A. C. Shau, and H. W. Yeh, "Testing for Event-Driven Microservices Based on Consumer-Driven Contracts and State Models," in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)* (IEEE, 2022), 467–471.

8. I. Robinson, "*Consumer-Driven Contracts: A Service Evolution Pattern*," (2006), accessed on December 12, 2023, https://martinfowler.com/articles/consumerDrivenContracts.html.

9. Microsoft, "*Consumer-Driven Contract Testing (CDC) - Code With Engineering Playbook*," (2023), accessed on December 12, 2023, https://microsoft.github.io/code-with-engineering-playbook/automated-testing/cdc-testing/.

10. I. Ghani, W. M. Wan-Kadir, A. Mustafa, and M. I. Babir, "Microservice Testing Approaches: A Systematic Literature Review," *International Journal of Integrated Engineering* 11, no. 8 (2019): 65–80.

11. M. Waseem, P. Liang, G. Márquez, and A. Di Salle, "Testing Microservices Architecture-Based Applications: A Systematic Mapping Study," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)* (IEEE, 2020), 119–128.

12. J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Industry Practices and Challenges for the Evolvability Assurance of Microservices: An Interview Study and Systematic Grey Literature Review," *Empirical Software Engineering* 26 (2021): 1–39.

13. H. M. Ayas, H. Fischer, P. Leitner, and F. G. D. O. Neto, "An Empirical Analysis of Microservices Systems Using Consumer-Driven Contract Testing," in *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (IEEE, 2022), 92–99.

14. A. Koschel, I. Astrova, M. Bartels, M. Helmers, and M. Lyko, "On Testing Microservice Systems," in *Proceedings of the Future Technologies Conference* (Springer, 2021), 597–609.

15. B. Kitchenham, "Procedures for Performing Systematic Reviews," *Keele, UK, Keele University* 33, no. 2004 (2004): 1–26.

16. G. D. Schwarz, *Supplementary Materials for the Study "Ensuring Syntactic Interoperability Using Consumer-Driven Contract Testing"*, (2025), https://doi.org/10.5281/zenodo.15364770.

17. C. Wohlin, "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering* (ACM, 2014), 1–10.

18. V. Clarke, V. Braun, and N. Hayfield, "Thematic Analysis," *Qualitative psychology: A practical guide to research methods* 222 (2015): 248.

19. V. Braun and V. Clarke, "Using Thematic Analysis in Psychology," *Qualitative research in psychology* 3, no. 2 (2006): 77–101.

20. R. L. Baskerville, "Investigating Information Systems With Action Research," *Communications of the association for information systems* 2, no. 1 (1999): 19.

21. K. Petersen, C. Gencel, N. Asghari, D. Baca, and S. Betz, "Action Research as a Model for Industry-Academia Collaboration in the Software Engineering Context," in *Proceedings of the 2014 International Workshop on Long-Term Industrial Collaboration on Software Engineering* (ACM, 2014), 55–62.

22. G. I. Susman and R. D. Evered, "An Assessment of the Scientific Merits of Action Research," *Administrative science quarterly* 23, no. 4 (1978): 582–603.

23. S. Kemmis, R. McTaggart, and R. Nixon, *The Action Research Planner: Doing Critical Participatory Action Research* (Springer, 2014).

24. S. P. Ma, I. H. Liu, C. Y. Chen, and Y. T. Wang, "Version-Based and Risk-Enabled Testing, Monitoring, and Visualization of Microservice Systems," *Journal of Software: Evolution and Process* 34, no. 10 (2022): e2429.

25. J. Stählin, S. Lang, F. Kajzar, and C. Zirpins, "Consumer-Driven API Testing With Performance Contracts," in *Advances in Service-Oriented and Cloud Computing: Workshops of ESOCC 2016, 2018* (Springer, 2016), 135–143.

26. M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Márquez, "Design, Monitoring, and Testing of Microservices Systems: The Practitioners Perspective," *Journal of Systems and Software* 182 (2021): 111061.

27. S. P. Ma, C. Y. Fan, Y. Chuang, W. T. Lee, S. J. Lee, and N. L. Hsueh, "Using Service Dependency Graph to Analyze and Test Microservices," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2 (IEEE, 2018), 81–86.

28. H. Li, J. Wang, H. Dai, and B. Lv, "Research on Microservice Application Testing System," in *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)* (IEEE, 2020), 363–368.

29. Postel, J., "*Rfc0793: Transmission Control Protocol*," (1981).

30. S. Spall, "Peer Debriefing in Qualitative Research: Emerging Operational Models," *Qualitative Inquiry* 4, no. 2 (1998): 280–292.

Supporting Information

Additional supporting information can be found online in the Supporting Information section.

Appendix A

Developer Interviews - Interview Guide

1. **Training materials**
   - How did you approach the initial training?
   - Were the materials comprehensible?
   - Were there unexpected or counterintuitive insights?
   - Are there any questions that remain unanswered? If yes, which ones?
   - Was additional material consulted? If yes, why?
2. **CI integration**
   - How did you approach the integration of CDC tests into the CI pipeline?
   - Were there any issues? If yes, which ones?
   - Do you have ideas for improvement?
3. **Implementation of CDC tests**
   - How was the implementation of CDC tests carried out?
   - Identification of service interactions to be tested:
     - Which interactions were identified?
     - How was the identification process conducted?
     - How were different types of notifications handled?
     - Did you consider the value 'Not a Number' for IDs in interactions?
   - Implementation of consumer-side tests:
     - What adjustments were necessary in the UI source code?
     - Were there any issues or uncertainties during the implementation? If yes, how were they addressed?
     - Does the separation between test and fixtures files make sense to you?
   - Implementation of provider-side tests:
     - What adjustments were necessary in the Notification Service source code?
     - How did you implement the mocking?
     - Were there any issues or uncertainties during the implementation? If yes, how were they handled?
   - Did you uncover any software defects?
   - Did you encounter any limitations encountered with Pact?
   - Did you experience a difference in effort between consumer-side and provider-side test development?
4. **CDCT in general**
   - Comparison of CDCT with other testing methods:
     - Which testing methods did you employ in the project previously?
     - Did you experience difference in training efforts between CDCT and these other testing methods?
     - Did you experience difference in development efforts between CDCT and these other testing methods?
     - Did you experience difference in test execution efforts between CDCT and these other testing methods?
     - Did you experience difference in runtime efforts between CDCT and these other testing methods?
   - Advantages, disadvantages, challenges, and guidelines:
     - Which advantages do you associate with CDCT?
     - Which inherent disadvantages do you associate with CDCT?
     - Which challenges do you associate with CDCT?
   - Which guidlines do you associate with CDCT? How reasonable do you think these guidelines are?