A Real-time Data Synchronization Approach for High-availability Micro Applications

Fernando R. de Moraes
São Paulo State University – UNESP
Rio Claro, Brazil; and
Tractian
São Paulo, Brazil
fr.moraes@unesp.br

Daniel de Almeida
São Paulo State University – UNESP
Rio Claro, Brazil
daniel.almeida19@unesp.br

Frank J. Affonso
São Paulo State University – UNESP
Rio Claro, Brazil
f.affonso@unesp.br

ABSTRACT

The Microservice Architecture (MSA) and the Micro Frontend Architecture (MFA) are scalable, flexible, and innovative architectures for developing micro applications (MApps). The integration of these architectures optimizes development by leveraging the strengths of both architectural styles, enabling decoupled deployment, specialized domain teams, isolated failures, scalability, and software maintenance. To the best of our knowledge, the engineering of applications based on MFA poses significant challenges, as practitioners encounter difficulties because of the dependency generated between MApps. This dependency arises from the data sharing between microservice-based applications. In a real-world execution scenario, such applications require substantial data volumes and distributed databases across microservices. While the microservices are designed to communicate via a network using lightweight communication protocols, the data interchange is relatively slow compared to the database communication. Based on the presented context, this paper proposes a real-time data synchronization approach to address the limitations of MApps development concerning communication and data replication. In order to evaluate the applicability, strengths, and weaknesses of the proposed approach, a proof of concept was conducted for a condominium management system. As a result, the evidence gathered in this paper enables us to create a favorable perspective to contribute efficiently to the MApps domain by providing an approach to this research area and an empirical evaluation for researchers and practitioners.

KEYWORDS

Micro frontend, microservice, data replication, communication

1 Introduction

Microservices Architecture (MSA) is an architectural style that facilitates the development of applications as a collection of small services. Each service must be designed to run in an isolated process and communicate with each other through lightweight mechanisms. By adopting this architectural style, the resulting applications may present a set of features, including but not limited to: fault isolation, scalability, continuous and independent deployment, data isolation, small code base, technology-agnostic, resilience, and faster time-to-market [8, 14, 31].

In parallel, Micro Frontend Architecture (MFA) is an MSA-inspired architecture for front-end applications that inherits the benefits and drawbacks of its predecessor. The purpose of MFA is to support the development of contemporary applications (i.e., web, desktop, and mobile), thereby facilitating the independent and efficient creation, deployment, and maintenance of decoupled front-end applications [3, 11]. Moreover, it is noteworthy that MFA and its integration approaches can provide a dynamic frontend capable of rendering applications at runtime [9, 18]. The applications based on MFA/MSA will be referred to from this point onward as MApps (Micro Applications).

Despite the established correlation between the aforementioned architectural styles (i.e., MSA and MFA), recent studies suggest that the MFA is in an initial stage of development that requires further comprehensive investigation [18]. In this sense, Moraes & Affonso [18] have proposed an integration approach called Runtime Service-Based Frontend (RSBF) to support the development of compiled MApps (i.e., Mobile, Desktop). To do so, the RSBF approach was based on concepts of Remote Component Rendering (RCR) associated with the Backend For Frontend (BFF) pattern [29] to solve continuous delivery issues, provide runtime integration, and help teams scale their development process of MApps. Beyond the contribution of the RSBF approach, the study revealed benefits when the backend of the MFA is loosely coupled and has fewer dependencies in relation to the other MApps. However, the aforementioned study also revealed that the decoupling of the MApps necessitates that each one utilize its database and avoid communicating with any other MApp databases to fulfill its responsibilities. In essence, these applications must facilitate the exchange of data (or synchronization) between MApps [19].

A preliminary analysis of data synchronization and communication between MApps reveals that, despite the literature on the microservice architectural style pointing to the use of lightweight mechanisms for communication, the exchange of information is carried out through a network that can cause an inevitable dependency between them [12, 32, 35]. Furthermore, the exchange of information via the network is slower than querying a local database (i.e., replicated data) in a MApp. Based on this scenario, it can be stated that the MApps are decoupled in terms of front-end and back-end, but there is a dependency caused by the granular data distributed in each MApp [13, 30, 37]. According to Moraes et al. [19] and Moraes & Affonso [18], the intercommunication between MApps (or microservices) introduces a series of drawbacks, including but not limited to, increased complexity (e.g., management overhead, communication complexity, deployment challenges, dependency between MApps, among others), data management and consistency (e.g., data consistency, high data availability, among others), network overhead and performance (e.g., network latency,

network traffic, among others), security, testing, and other challenges [28, 39].

Based on the presented context, this paper presents a new approach to support the real-time data synchronization (RTDSync) for high-availability MApps to minimize the impacts related to the aforementioned limitations. The approach outlined in this paper is driven by the concepts of Capture Data Change (CDC) [23] and a pipeline for data replication [28]. From an innovation perspective, the RTDSync approach utilizes such concepts to ensure high availability of MApps using real-time data replication. Furthermore, this approach was designed with the following features: (i) the RTDSync approach is technology-agnostic, meaning its operation is independent of programming languages and technology stacks; (ii) the RTDSync approach is non-intrusive, as the software engineers develop MApps without the need to inject the approach's source code into these applications; and (iii) the RTDSync approach enables filter-based data migration between origin and target MApps, thereby implementing data replication to meet the global application's requirements. An evaluation of the RTDSync approach's applicability was conducted through a proof of concept focusing on a condominium management system. As a result, the approach proposed in this paper demonstrates considerable promise to contribute to the domains of MSA and MFA, as it presents concrete evidence for addressing the synchronization of real-time data in systems that necessitate high availability.

Given the above, the main contributions of this paper can be summarized in three key elements as follows. The RTDSync approach is designed to serve as a means to facilitate real-time data synchronization for high-availability MApps. In terms of application domains, the proposed approach can be applied to microservice-based and MApp-based applications. Regarding the design strategy, it is designed to be technology agnostic and non-intrusive to MApp development. Furthermore, from an operational perspective, the RTDSync approach provides persistent data replication with high availability and an automated data replication pipeline. In addition to the contributions concerning the approach, this paper presents a collection of requirements in Section 3.1 that can provide a solid foundation for directing the development or enhancement of new solutions for real-time data synchronization in MApps or other software domains. Finally, it is important to highlight that the proof of concept detailed in this paper provides a significant theoretical and practical contribution to the advancement of real-time data synchronization for high-availability MApps, both through the use of the approach and the technology stack used to instantiate the RTDSync approach.

The paper is organized as follows: Section 2 introduces the essential concepts related to MSA, MFA, and an overview of related work; Section 3 presents the RTDSync, an approach real-time data synchronization for MApps; Section 4 describes a proof of concept to show the applicability of RTDSync; Section 5 shows a brief discussion of results; and Section 6 summarizes the conclusions and perspectives for future work.

2 Background and Related Work

This section presents the background and related work that contributed to the development of this paper. First, the concepts of microservices, micro frontends are introduced. Next, related work on the research topic of this paper is addressed.

Microservices. According to Lewis and Fowler [14], the microservice architectural style is an approach to developing a single application based on a suite of small services. Each service can be designed to run in its process and communicate with lightweight mechanisms, often an HTTP (Hypertext Transfer Protocol) resource API (Application Programming Interface). These services must be designed around business capabilities, and deployed independently through a fully automated deployment process. The decentralized management of these services enables them to be written in different programming languages and use different data storage technologies. This organizational model promotes the establishment of application-centric teams, thus offering a feasible alternative for improving continuous software delivery. From another perspective, Richardson [28] describes that "*a key characteristic of the microservice architecture is that the services are loosely coupled and communicate only via APIs. One way to achieve loose coupling is by each service having its own data store*". This definition suggests that database isolation for each microservice constitutes a pivotal feature in the MSA, facilitating the decoupling of MApps.

Micro Frontends. Jackson's definition [11] of micro frontends as "*an architectural style where independently deliverable frontend applications are composed into a greater whole*" is an insightful contribution to this paper. According to the aforementioned author, the primary benefits of micro frontends can be summarized in three features: (i) more cohesive and maintainable code bases; (ii) scalable organizations with decoupled and autonomous teams; and (iii) incremental development with continuous delivery of software. As can be observed in this section, these features represent some of the advantages provided by the development based on microservices [22]. A study by Moraes et al. [19] demonstrated that the Micro Frontend Architecture (MFA) enables scale development teams to engage in engineering activities for frontend applications. Additionally, the MFA also promotes the complete decoupling of a front-end monolith into micro frontend applications, where such applications can be developed vertically by any full-stack team (i.e., front-end, backend, and database).

Figure 1 presents a comparative analysis of three distinct architectural styles, namely "Monolith", "Microservice Architecture", and "Micro Frontend Architecture", from the perspective of modular organization. In short, a monolith encompasses all the application's source code, encompassing the database, the backend, and the user interface. In contrast, a microservice-based application comprises a collection of smaller, more manageable services (backend side), associated with a monolithic frontend. A micro frontend application exhibits a similar organization to MSA, subdividing the user interface into smaller components or modules that can be developed, deployed, and maintained independently [11, 19]. Consequently, each module (i.e., micro frontend application) is responsible for a distinct business capability, thereby redefining the development and scaling of user interfaces.

As shown in Figure 1, the development of MApps encompasses the concepts of MSA, assuming its benefits and drawbacks. By breaking a monolithic frontend into smaller parts, a development team can manage each one, promoting modularity, scalability, and

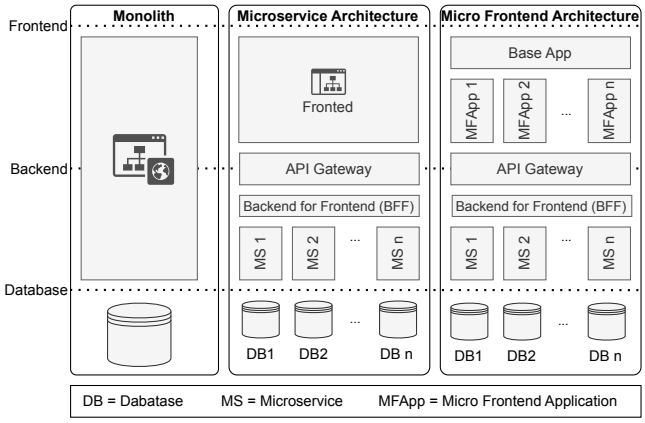

Figure 1: Architectural Comparison

flexibility in technology choices. In contrast, this organization requires special attention to data synchronization, data consistency across a distributed environment, and communication between services to execute system operations [18, 19].

As **related work**, Antonin et al. [33] conducted a practical study (**S1**) on data synchronization and improvement of the database performance based on two case studies. In summary, the study encompassed two evaluation scenarios: data synchronization and performance enhancement. The first focused on the synchronization of data across different databases, while the second investigated the performance of databases under high-load conditions. Regarding the database, this study was based on the Redis [27] for data replication. Although this proposal ensures high data availability, the application's resilience is compromised because of the occurrence of data loss in the Redis storage. Therefore, it can be said that the resynchronization of all application data is required.

Sidath and Indika [38] proposed a solution (**S2**) to reduce inter-service communication latency and improve the overall performance of microservice-based applications, both concerning throughput and response time. In summary, the proposed solution is based on an asynchronous communication pattern using Redis Stream [26] to enable publisher-subscriber communication between services. The proposed solution is independence from a Change Data Capture (CDC) tool, wherein microservices assume responsibility for the distribution of events, ensuring their caching in Redis and facilitating subsequent querying by other microservices.

Bantia et al. [34] developed a methodology (**S3**) and an implementation of a pipeline to reduce the bottleneck and improve the time spent in the publish-subscribe model for microservices. In summary, the main purpose of this methodology is to reduce the CPU and memory consumption of the containers in which the microservices are running. Furthermore, the microservices consumed messages from Kafka in an event-driven approach without focusing on data replication.

A comparative analysis of these studies reveals that none of them presented a complete and robust solution that systematizes the knowledge about real-time data synchronization for high-availability applications. It is worth noting that the studies presented as related work present an empirical implementation for specific problems (i.e., data synchronization, data replication, microservice communication). All studies (S1, S2, and S3) demonstrate the **high-availability** feature; however, none incorporate **resilience** into their solutions. Within the scope of this paper, high availability and resilience are features defined as a system's capacity to maintain operational accessibility despite data synchronization and/or MApps communication failures. It is noteworthy that only S2 demonstrated a degree of **flexibility** by implementing a mechanism to regulate data replication. Studies S1 and S3 implemented a non-intrusive CDC tool, facilitating the **maintenance** of microservices and/or micro-applications. With regard to the evaluation, all studies conducted **case studies** to gather some evidence of their solutions, with studies S2 and S3 conducting some type of benchmark. Specifically, S2 conducted a benchmark comparing service communication with the replicated database using HTTP requests and evaluated the gains in good performance resulting from direct communication with the replicated database. S3 conducted a benchmark comparing the difference between microservices generating events with Kafka compared to a CDC tool (e.g., Kafka Connect), which revealed favorable performance outcomes with the CDC tool.

Despite the important initiatives mentioned, no study has yet emerged that adequately supports the real-time data synchronization for high-availability MApps as a comprehensive and robust solution. The studies presented in this section as related work address the data synchronization in particular approaches, focusing on the microservice communication or data replication as a specific solution to minimize the adversities in a target application. In parallel, it is noteworthy that these studies (related work) did not encompass the minimum essential requirements for this type of solution, specifically the necessity of real-time data synchronization for high-availability MApps.

3 The Proposed Approach

This section presents the RTDSync, an approach for real-time data synchronization to support the development of high-availability MApps. The proposed approach implements a mechanism to transport data in real-time from an origin database to a target database by means of a middleware application between microservices or MApps. Therefore, it can be posited that our approach exhibits flexibility with regard to data replication, as this operation is executed through a non-intrusive strategy. Thus, developers are not forced to make any code changes in the target applications. To do so, our approach uses a CDC mechanism to listen to data changes from an origin database to a messaging queue. In order for this queue to be consumed by the replication engine, the messages must undergo deserialization and transformation of the data (i.e., origin and target). Additionally, to enhance data integrity, the replication engine can execute queries (i.e., lookup operation) against the origin database. Finally, the replication engine persists data in the target database, which will be used by the target application instead of communicating between the existing applications (i.e., origin and target). Regarding the design, the RTDSync approach was developed based on a set of requirements gathered through a literature investigation presented in the Section 2. Section 3.1 provides an overview of such requirements, and Section 3.2 presents the architectural view for the approach proposed in this paper in a modular organization.

3.1 Approach requirements

This section presents the requirements that were gathered from the literature and that served as the base of the design of the RTDSync approach [19, 33, 34, 37, 38]. Since the proposed approach will operate in a distributed computing scenario (e.g., MSA or MFA), a number of functional and non-functional requirements will need to be met. Next, a description of each functional requirement (FR) is addressed:

- **FR1.** A real-time data synchronization solution must provide consistent data synchronization. In other words, this entails the implementation of a mechanism that guarantees the consumption of any alterations made to an original database [33, 34, 38].
- **FR2.** In order to facilitate real-time synchronizations (i.e., the interval between the storage of data from its origin and its eventual destination), a solution must implement a CDC mechanism. The mechanism's effectiveness depends on its ability to detect changes in the origin database, thus reducing inter-database transfer time [33, 34].
- **FR3.** A real-time data synchronization solution must implement a mechanism to ensure that data updates are performed in the chronological order of the user's requests. To do so, this mechanism must incorporate a messaging service, thereby facilitating the synchronization of data in accordance with the sequence of user-initiated changes within the origin database [33, 34].
- **FR4.** A real-time data synchronization solution must facilitate data replication through an independent mechanism of the target application (i.e., MApps). This segmentation aims to ensure that developers do not have to inject code related to the proposed approach into such applications. As a result, it is expected that this design strategy will either eliminate or minimize the necessity for developers to augment the learning curve associated with novel technologies when developing this type of application [19, 33, 34, 38].
- **FR5.** To ensure target application data fidelity, a real-time synchronization solution must guarantee complete data transfer between the origin and target databases. To do so, this mechanism must facilitate the execution of lookup operations within the origin database [33].
- **FR6.** A real-time data synchronization solution must enable direct communication with the synchronized data, circumventing any network communication with the origin and target databases that have already replicated their data. As a result, this communication strategy is expected to reduce the volume of requests between application components (e.g., MApps) [33, 34, 38].

In terms of non-functional requirements (NFR), our investigation suggests that at least four must be met, namely [19, 33, 34, 38]:

- **NFR1–Availability.** A real-time data synchronization solution must enable the development of MApps with high data availability. To do so, the solution must be designed to maintain as much availability as possible, since unavailability can cause delays in synchronization between the origin and target databases.
- **NFR2–Performance.** A real-time data synchronization solution must facilitate the transfer of data between the origin and target databases with satisfactory performance. The timely execution of this data transfer is crucial for maintaining data consistency across databases and resolving potential inconsistencies.
- **NFR3–Testability.** A real-time data synchronization solution must provide software testing resources. These resources can be facilitated by the decoupling of MApps from the synchronization mechanism. This segmentation strategy enables these applications to be data-agnostic, eliminating the need for external applications (i.e., microservice and MApp implementations) to execute tests.
- **NFR4–Failure isolation.** A real-time data synchronization solution should facilitate the isolation of faults for each MApp. This solution should be designed to avoid interfering with MApp development by injecting data synchronization code. The absence of dependency between these applications also benefits fault isolation because it enables an MApp to function independently (or isolated) from the rest of the system. Consequently, the unavailability or failure of an MApp would not have a system-wide impact.

3.2 Approach architecture

The architectural overview of the RTDSync approach proposed in this paper is shown in Figure 2. As can be observed, the RTDSync approach is based on a layered design strategy, which acts as a separate layer between the MApps (i.e., **supervised layer**) and the proposed approach (i.e., **supervisor layer**). This strategy enables the approach to function as a non-intrusive solution regarding other application layers, specifically the origin and target MApps.

Regarding the organizational structure, an MApp (i.e., supervised layer) is composed of four constituent elements: a micro frontend, a microservice, a database, and a log store (see Legend area). Following the development phase, MApps are integrated into an architectural model that facilitates their accessibility to a range of client devices, including the Internet of Things (IoT), mobile devices, web browsers, and command-line interfaces (CLIs). This organization enables the utilization of infrastructure resources that facilitate the advantages inherent in microservice or micro frontend architectural styles, such as the separation of code bases, scalability, reusability, maintainability, among others. As illustrated in Figure 2, a software system (i.e., an origin application) can be structured as a set of MApps, leveraging a set of infrastructure to integrate the strengths of microservices and micro frontends.

The RTDSync (i.e., supervisor layer) was designed based on a log-based approach of the database operations (e.g., insert, update, and delete) for each MApp. In short, to facilitate the data synchronization between origin and target databases, the developer is required to specify the database entities to be monitored so that the data is migrated in real time. Moreover, the proposed approach has been meticulously structured into modules, providing a feasible alternative for the evolution or maintenance of resources within each module. Next, a description of each module is addressed, as well as the preliminary setup of these modules and the process for real-time data synchronization.

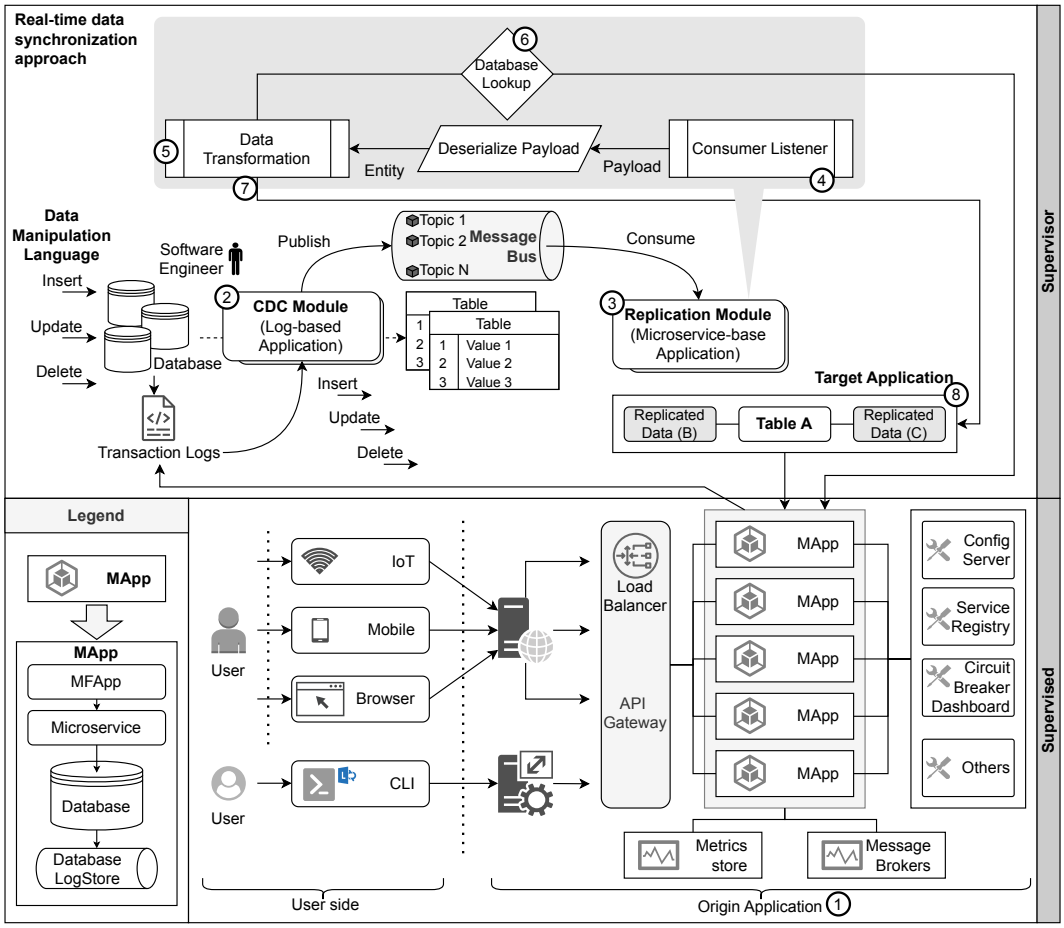

Figure 2: Architecture Overview

Preliminary setup. Considering that the MApps have already been developed, the software engineer must first select the database entities that have real-time data synchronization capabilities (i.e., origin and target). An origin MApp is a full application that encompasses a database designed to fulfill its goals and domain-specific requirements, which may provide a subset of information to a target MApp. This information is essential for the target MApp to fulfill its goals without requesting information from the origin MApp. To do so, the software engineer must define a message topic that will define how data synchronization will occur between the MApps (i.e., origin and destination). The message topic facilitates the real-time synchronization of data among multiple target MApps (see `Message Bus` element).

`CDC Module`. The objective of this module is to capture changes (i.e., identifying and tracking) from the origin databases of the MApps so that an external system can process such changes for data synchronization (**FR1** and **FR2**). To achieve this objective, it is essential that the databases of each MApp are configured to expose their transaction operations (see `Database LogStore` element) [2, 4, 10]. Among the most common approaches found in the literature (i.e., timestamps on rows, timestamps on files, replication, and trigger) for implementing a CDC mechanism, it is suggested that this module uses the log-based approach for the following reasons [4]: (i) ease of handling logs, since the logs are stored in a transaction repository that can be easily integrated with other external systems; (ii) ease of implementation, since the log-based approach is more widespread among software developers and requires a shorter learning curve; and (iii) restoration capability, since the logs can be reused to replicate the entire change process.

`Message Bus`. This element represents a communication mechanism through a message bus, acting as an intermediate element between data capture and data replication (see `Replication Module`). Neha et al. [20] characterized a publish/subscribe messaging system as "*a pattern that is characterized by the sender (publisher) of a piece of data (message) not specifically directing it to a receiver. Instead, the publisher classifies the message somehow, and that receiver (subscriber) subscribes to receive certain classes of messages*". The RTDSync approach delineates producers (publishers) as topics representing captured database changes, while consumers (subscribers) are defined as processes that consume these changes for data synchronization (**FR3**). It is noteworthy that a generated topic may be utilized by multiple consumers. In other words, a change detected in an origin database can be reflected in more than one target database. Evidence presented by Neha et al. [20] suggests

that implementing the publish/subscribe messaging system is not a trivial task, as it necessitates configuration management, message consistency, and reliable performance. To address these challenges, it is recommended to adopt a data streaming platform to manage configuration and build high-performance data pipelines.

`Replication Module.` The objective of this module is to consume data from the `Message Bus` so that data replication can be performed (**FR4**). To achieve this objective, this module must implement a mechanism (`Consumer Listener` element) to read topics present in the `Message Bus` element, which represents the changes detected in the origin database. Next, the payload undergoes deserialization (`Deserialize Payload` element), enabling the subsequent transformation of data (`Data Transformation` element). During the transformation process, an on-demand data query (see `Database lookup`) may be initiated to the origin databases. In short, this data query is to be utilized in instances where the transformation process encounters incomplete data (**FR5**). For instance, a scenario that may arise is when the message transmitted from the `Message Bus` contains data from a relational table, and the target application requires the relationship data. In such a scenario, the on-demand data query should be utilized for the transformation process, thereby retrieving the necessary data from the origin database. Once message bus topic payloads have been processed and data transformations completed, instantiated objects can be written to the target database. Finally, the data must be persisted to the target databases (see `Target Application`) of each MApp (**FR6**).

Synchronization process. Regarding the RTDSync operating mode, real-time data synchronization between origin and target MApps can be summarized in a preliminary configuration step and eight execution steps. As reported in this section, the software engineer must define which MApps should present real-time data synchronization so that the origin and target applications can be identified and configured. Next, a description of each execution step is addressed:

(1) When data is transacted during the MApp execution (e.g., an insert, update, or delete operation), a transaction log is generated (`Database LogStore`) containing the data of this transaction (or change) in the database.
(2) The `CDC Module` is responsible for identifying these transaction logs, capturing the pertinent information, and publishing it to a message topic managed by the `Message Bus`.
(3) The `Replication Module` is responsible for monitoring the messaging topics for new messages. As such, each message published in a monitored topic is consumed by the application present in the aforementioned module.
(4) In the `Replication Module`, the message containing a transaction log payload undergoes deserialization (`Deserialize Payload`) in a designated data model, yielding an `Entity`.
(5) For the entity generated in the previous step to be persisted, the deserialized data model can be transformed into a new model (`Data Transformation`), which may contain only part of the origin data or a change in its formatting. Note that transformation operations are executed according to the target application's requirements.
(6) Given the potential need for data completeness in the target data model – including its relationships and aggregated data, among others – the replication module can query the data (`Database Lookup`) in the replication origin database. This process enables the accurate definition of the target data model (i.e., replicated data).
(7) Once the data has been transformed into the target model, the persistence operation is performed in the target database.
(8) In this step, the focus is on data replication, as illustrated by the `Target Application` with one origin table (Table A) and two replicated data targets, B and C. As a result, when an origin MApp that contains such a database performs its operations using the replicated data, communication with another target MApp is unnecessary.

4 Proof of concept

This section presents a proof of concept to evaluate the applicability, strengths, and weaknesses of the RTDSync approach proposed in this paper. The subject application for this empirical analysis is a condominium management system, referred to as CondSys. Next, a description of the system under consideration and the empirical strategies used to conduct this proof of concept are presented.

Subject Application. CondSys is a system that enables condominium management by a single condominium administrator. A condominium is defined as a legal entity composed of units, which may include houses or apartments, as well as specific areas, which may include garages, and common areas, which may include rentable areas, courts, playgrounds, and swimming pools, among others. From an administrative perspective, each condominium must have a manager, who is responsible for the interests of the residents and ensuring the proper functioning and maintenance of the condominium. This manager may be a resident elected in an assembly or a professional hired for that purpose. Regarding the residences, each unit must have only one responsible resident, who may be the unit owner or a tenant. These residents must inform other residents (e.g., children and spouses) of the unit so that access to areas of the condominium can be facilitated. Concerning the utilization of shared spaces, the CondSys system must enable the rental of common areas by residents for designated periods. To do so, the condominium manager must establish a reservation period for the common areas (e.g., 30, 60, or 90 days). A reservation must contain specific information about the common area being requested, the resident who is making the request, and the area itself. Due to limitations in scope and space, the remaining system functionalities are not addressed in this paper. However, it is worth highlighting that this system has three distinct views: condominium administrator, resident, and building manager. These views provide various features for efficient condominium administration.

Preliminary setup. Based on the requirements delineated in this section, the CondSys system was organized into several MApps, as described in Section 3.2. Since the approach proposed in this paper operates on the backend of the MApps, the configuration and coding of the frontend will not be addressed in this section. Thus, in order to illustrate the behavior of the RTDSync approach, three MApps will be described in this section, namely: Resident, Condominium, and Reservation. Regarding development, each MApp was implemented in Java [21] programming language (version 21 LTS), using the Spring Boot [36] framework (version 3.4.4), with the

following databases: MongoDB [16], PostgreSQL [24], and MariaDB [15]. As the proposed approach does not necessitate intrusion or additional configuration in the MApps, the implementation details of each application will be omitted.

Regarding the preparation of each MApp to work with the RTDSync approach, as described in Section 3.2, the software engineer must configure the `Database LogStore` elements. These elements can be defined as resources provided by database manufacturers that enable the immediate use of the resource for production environments without requiring additional implementation of its use. Next, a description of each element used to instantiate the aforementioned elements is addressed.

MongoDB Operations Log. As stated by MongoDB [17] the operations log (oplog) "*is a special capped collection that keeps a rolling record of all operations that modify the data stored in your databases. If write operations do not modify any data or fail, they do not create oplog entries*". The MongoDB Oplog is the state-of-the-art collection for log-based replications, which makes it possible to create tools to fetch these log entries for the messaging bus.

Postgres Logical Replication. The PostgreSQL database contains a reliable feature for high-availability databases known as Logical Replication. This feature facilitates data replication through the logs of each database change, which are registered in the Write-Ahead-Logging (WAL). The WAL ensures that modifications to data files are documented in this log before they migrate to permanent storage. In short, this feature facilitates the notification of changes stored in the WAL to the message bus application for replication [25].

Approach instance. With regard to the RTDSync modules, the `CDC Module` was instantiated with Kafka Connect (KC), while Apache Kafka [1] was used for implementing the `Message Bus` element. According to Confluent.io [5], KC can be defined as a component that enables scalable and reliable streaming of data between Apache Kafka and other external systems. In the context of the RTDSync approach, KC will facilitate the development of connectors for each MApp, thereby ensuring the capture of alterations made to the database for each application (i.e., MApp).

As described in Section 3.2, the RTDSync approach instance is based on a service for publishing, subscribing, storing, and processing real-time data streams. To do so, the `Message Bus` element was instantiated in Kafka [1], "*a distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications*". In short, this element is responsible for monitoring updates, inserts, and delete operations on the target database, ensuring that these operations can be published into the data pipeline. This pipeline enables the efficient and reliable replication of data by the consumer as captured on the target databases.

The Debezium [7] platform was instantiated in combination with the `CDC Module`, the `Message Bus` element, and `Replication Module` to facilitate the CDC across different databases. In summary, Debezium consumes these changes and disseminates them to the Kafka stream, ensuring the preservation of message order within the data pipeline. Moreover, in the context of the RTDSync approach (see Section 3.2), the development of a consumer application is necessary for the processing of this data. This application must transform the data into a new data schema, which is then persisted in the target database.

The implementation of the `Replication Module` is similar to a microservice-based application. From a behavioral perspective, the application must systematically read the topics present in the `Message Bus` element, which represents the change queue detected in the origin databases in the format of topics for each entity whose data is to be synchronized in real time. Since the operation of this module is associated with the data to be replicated/synchronized, its implementation occurs in parallel with the development of each MApps (i.e., according to each topic definition of the `CDC Module`).

In order to enable replication/synchronization, the developer must implement two classes for the CDC mechanism (i.e., Debezium), namely: `DebeziumMessage` and `DebeziumPayload`. Listing 1 shows the full contents of the `DebeziumMessage` class. This class is the core of the CDC mechanism, as it enables listening for changes and receiving objects as a Kafka entity to perform transformation logic. Therefore, it can be said that any entity that reflects an object in the database can be contained in the `DebeziumMessage` to be processed in a consumer method. In short, this class represents data change events captured from database tables at the row level (i.e., insert, update, and delete operations). To do so, this class must contain three attributes: (i) data related to the entity schema (Line 4); (ii) payload data (Line 5); and (iii) status of the message on the `Message Bus` (Line 6). As a result, the data transformation process utilizes a JSON-formatted (JavaScript Object Notation) object of this class, which includes the instantiated entity. The `@JsonIgnoreProperties` annotation (Line 1) signifies that unknown properties will be ignored during deserialization, thereby avoiding the occurrence of an exception. The `@Getter` annotation, which is part of the Lombok library, automatically generates getters for the attributes of a class.

Listing 1: Source code of the DebeziumMessage class

```
1  @JsonIgnoreProperties(ignoreUnknown = true)
2  @Getter
3  public final class DebeziumMessage<P> {
4      private final Object schema;
5      private final DebeziumPayload<P> payload;
6      private Acknowledgment acknowledgment;
7
8      public DebeziumMessage(Object schema,
           ↪ DebeziumPayload<P> payload, Acknowledgment
           ↪ acknowledgment) {
9          this.schema = schema;
10         this.payload = payload;
11         this.acknowledgment = acknowledgment;
12     }
13 }
```

Listing 2 shows the implementation of the `DebeziumPayload` class, which refers to the data content of a change event that occurred in a database table. This class receives a generic `P` object that represents the entity to be deserialized from the message topic. The `before` attribute (Line 4) contains the data payload before the database persists the change, and the `after` attribute (Line 5) contains the payload after the change. The `operation` attribute (Line 6) is an enumeration that indicates the type of operation to be performed on the database, namely: create, update, delete, or truncate data.

Listing 2: Source code of the DebeziumPayload class

```
1  @JsonIgnoreProperties(ignoreUnknown = true)
2  @Getter
3  public final class DebeziumPayload<P> {
4      private final P before;
5      private final P after;
6      private final DebeziumOperation operation;
7
8      @JsonCreator
9      public DebeziumPayload(@JsonProperty("before") P
           ↪ before, @JsonProperty("after") P after,
           ↪ @JsonProperty("op") String op) {
10         this.before = before;
11         this.after = after;
12         this.operation = DebeziumOperation.of(op);
13     }
14 }
```

After the modules and elements of the RTDSync approach have been configured and instantiated, the software engineer can then proceed with the coupling step with the MApps that will synchronize data. Next, the implementation details for the coupling step between RTDSync modules and CondSys system are addressed.

Empirical research strategy. Because of space limitations, only the `Reservation` MApp will be used to demonstrate the behavior of the RTDSync approach for real-time data synchronization for the CondSys system. As illustrated in Figure 3, the `Reservation` MApp is composed of a set of domain classes and two additional classes (dotted line) that symbolize the data replication process between the origin MApps (i.e., `Resident` and `Condominium`) and the target MApps (i.e., `Reservation`). Thus, when data is entered into the origin MApps (i.e., `Resident` and `Condominium` MApps), it is synchronized with the target MApp (i.e., `Reservation` MApp). Therefore, it can be stated that the `Reservation` MApp can operate independently, thus eliminating the need for data search in other MApps to fulfill its designated purpose, which is to register a reservation of a rentable area for a condominium's resident.

In order to enable data synchronization, the developer must implement a consumer class, as shown in Listing 3, in the Replication Module for each entity from the origin MApp (i.e., origin data) that is to be synchronized in the target MApp (i.e., target data). As can be observed, the transformation code (Line 8) was omitted due to space limitations of this paper, but the full implementation is available in a source code repository [6]. It is worth noting, however, that this omitted code has to implement the transformation from the origin to the target data. Since the implementation of this proof of concept was based on Spring Boot, the `RentableAreaConsumer` class (Line 2) must be annotated with `@Component` (Line 1) in order to enable the framework to detect and manage this class automatically. The attributes declared in Lines 3 and 4 are used to identify a topic and the group to which this topic belongs. Furthermore, the `RentableAreaConsumer` class is required to implement a method to consume (Line 7) the data from a topic (`Message Bus` element) for data transformation before its persistence in the target database. The `@KafkaListener` annotation (Line 6) is used to specify that a method is the listener for a specific topic based on the annotation parameters (see attributes declared in Lines 3 and 4). This annotation facilitates the reception of topics as environment variables and

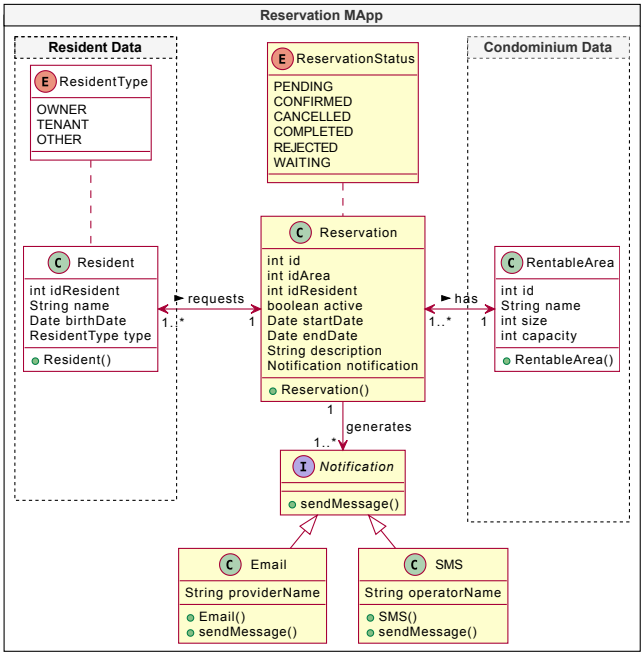

Figure 3: UML Model for Reservation MApp

uses groups to maintain a list of topics, which will undergo modification for each tenant (i.e., MApp). Finally, the `ConsumerRecord` class (Line 7), as a parameter of the consumer method, represents a key/value pair received from a topic. Regarding the functionality of the `RentableAreaConsumer` class, upon the insertion of data into the origin MApp (i.e., `Condominium` MApp – PostgreSQL), the data necessary for replication is synchronized with the target MApp (i.e., `Reservation` MApp – MariaDB). This data is referenced by the `RentableArea` class (dotted line – `Condominium data`) of the `Reservation` MApp.

Listing 3: Example of consumer method

```
1  @Component
2  public class RentableAreaConsumer {
3      private static final String topic =
           ↪ "MSCondominium.public.rentable_area";
4      private static final String group =
           ↪ "case-study-group";
5      //...
6      @KafkaListener(topics = topic, groupId = group)
7      public void consumer(ConsumerRecord<String, String>
           ↪ record) {
8      // Transformation logic
9      }
10 }
```

Figure 4 illustrates the behavior of the RTDSync approach through a sequence of steps that encompasses a data synchronization scenario based on three MApps: `Resident`, `Condominium`, and `Reservation`. Regarding these MApps, it is important to note that the development of these applications was conducted in accordance with the guidelines and architectural organization delineated in

Section 3 (Figure 2 – **Legend** area). However, it should be noted that Steps 1 to 3 illustrate the deployment of these applications within the execution environment. Steps 4 to 5 illustrate that the modules of the RTDSync approach are in listening mode to capture changes in an origin database. As can be seen between Steps 6 and 14, there is a complete cycle for changes in the origin database (`Resident` MApp) until synchronization with the target database (`Reservation` MApp). Step 6 represents a change scenario in the origin application (`Resident` MApp), which is captured by the `CDC Module` (Step 7). Next, Step 8, this change is published on the `Message Bus` as a change topic that will be consumed by the `Replication Module` (Step 9). Steps 9 to 11 illustrate a sequence of steps for consuming this topic. To do so, the `consumer()` method is called in Step 10, thereby facilitating the deserialization of the topic as a message with content in Step 11. After the deserialization process, an entity is generated (Step 12), and data transformation from the origin entity to the target entity can occur (Step 13). Next, the entity that has undergone transformation is synchronized with the target database (Step 14). Steps 15 to 24 will not be described in this section because they are similar to the set of Steps 6 to 14 and are focused on another application (`Condominium` MApp). Finally, Step 24 illustrates the synchronization between the two origin MApps (`Resident` and `Condominium`) and the target MApp (`Reservation`). Therefore, it can be stated that, after the data synchronization process, the `Reservation` MApp can fulfill its purpose, which is to facilitate the reservation of a rentable area of a condominium for a resident, without the necessity of external application communication with other MApps.

5 Discussion of Results

This section summarizes the main findings and discusses the relevance of the RTDSync approach to the software development and software engineering communities. Next, the main findings and results are addressed.

Design strategy. The RTDSync approach was designed as an independent solution (i.e., supervisor layer) in relation to the target application (i.e., MApps). This feature enables the approach to act as an independent layer, facilitating real-time data synchronization without interfering with the development of such applications. Furthermore, the RTDSync approach's modular organization allows it to behave as a scalable, reusable solution that can handle different types of databases simultaneously and serve as a reference for designing new solutions.

Learning curve. The approach proposed in this paper was designed to facilitate the preservation of developers in their native development environment. Therefore, it can be said that the cognitive effort required to learn to use RTDSync is reduced by the presence of a facilitator (i.e., supervisor layer) situated over the target applications (i.e., MApps). As a result, developers can focus on developing the applications without having to worry about injecting source code to handle real-time data synchronization.

Autonomous teams. As reported in this paper, the RTDSync approach has revealed a tendency to preserve the autonomy of development teams. As detailed in Sections 3 and 4, the MApps are independent applications, with their business rules and data entirely decoupled from other MApps. Therefore, features such as loose coupling, scalability, flexibility, fault isolation, continuous deployment, decentralized architecture, technology agnostic, and composability can be maintained in relation to the precursor architectural styles (i.e., MSA and MFA).

Testing isolation. As the RTDSync enables the design of MApps in a manner that tends to mitigate dependencies among such applications, this approach also benefits testing activity by allowing an MApp to be tested both in isolation and as part of a wider system (i.e., target system). In short, the proposed approach enables the independent design and development of MApps, thereby facilitating the execution of individual tests without the target system being operational. The isolation of tests shows that the RTDSync approach enables effective testability, minimizing the risk of propagating defects between MApps and enabling development teams to identify and resolve problems faster (**NFR3**).

Communication and high availability. As reported in Section 2, the evidence suggests that the communication between distributed applications has been an obstacle to the adoption of architectural styles such as MSA and MFA. These obstacles can be summarized as high response time, low performance, and strong coupling inherent to network communication. Based on this scenario, it can be said that the RTDSync approach is a feasible alternative for addressing the challenges associated with communication and coupling between distributed applications. By synchronizing data between the origin and target databases, the proposed approach enables significant minimization of communication between MApps (**NFR4**). Therefore, this feature ensures that communication only occurs when full origin information is needed, which directly contributes to the high availability of an MApp (**NFR1**).

Limitations. Despite the system utilized as a proof-of-concept (see Section 4) not addressing performance requirements and/or resource testing (**NFR2**), the preliminary evidence presented in this paper suggests that the proposed approach offers promising indicators regarding this NFR. This finding is reinforced by the results obtained thus far and by the studies presented as related work (see Section 2), which presented some indicators regarding the performance of CDC and messaging tools. These findings support the hypothesis that a favorable scenario can be established for the performance of the RTDSync approach. Nevertheless, it is also acknowledged that this approach does not constitute a "silver bullet" solution to all issues concerning communication among MApps. The development of software of this nature (i.e., MSA/MFA) is a multifaceted subject that encompasses numerous details when applied in diverse scenarios, environments, and application domains.

6 Conclusion and Future Work

This paper presents RTDSync, an approach that enables dealing with data synchronization for high-availability MApps. To do so, this approach provides a means to transport real-time data from an origin database to a target database by adopting a non-intrusive development strategy. This strategy enables the software engineers to concentrate their efforts on developing MApps without the necessity of incorporating cross-cutting code into such applications. As delineated in Section 3, the proposed approach was designed based on a set of functional and non-functional requirements that were identified from the literature review presented in Section 2. Based

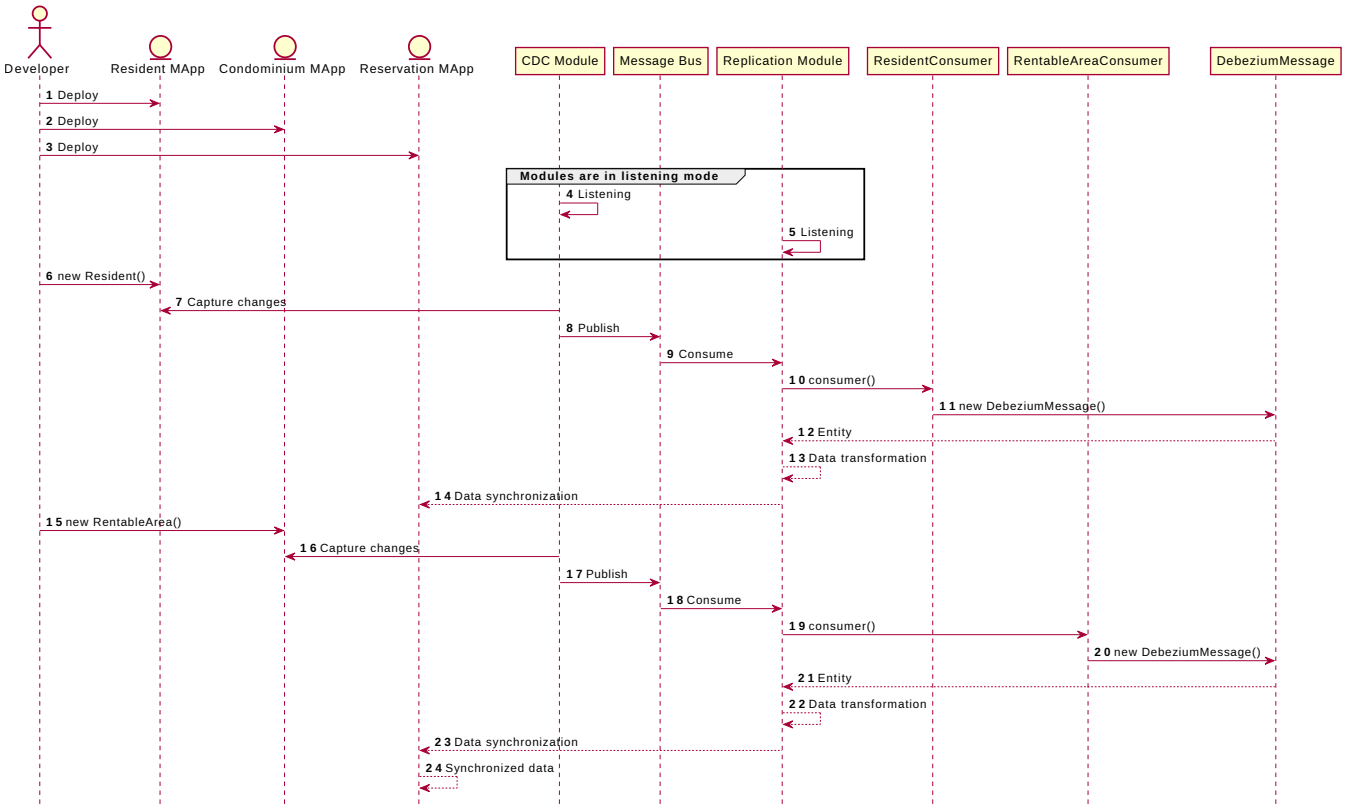

Figure 4: Sequence diagram for RTDSync operation

on the presented context, the main contributions of this paper can be synthesized as follows.

The approach in this paper may prove beneficial to the development and software engineering communities. To the best of our knowledge, it is the first initiative that can effectively support real-time data synchronization for high-availability MApps. In its favor, this approach combines features such as non-intrusive, technology-agnostic, and persistent data replication. From a design perspective, this approach presents an automated pipeline for data replication based on a modular organization that allows the solution to be flexible in both scale to meet demand and evolution to meet new application requirements.

The proof of concept presented in Section 4 provides an overview of the RTDSync approach, focusing on data synchronization between the origin and target databases/MApps. In this sense, the resulting prototype and the source code repository serve as a reference for others interested in data synchronization, as both can be adapted to work in microservice-based or micro frontend-based applications.

Regarding future work on the RTDSync, at least three activities may be achieved. The first is to conduct more case studies to expand the evaluation scenarios for the proposed approach. Thus, further investigation of alternative communication and synchronization scenarios, together with a range of databases, is imperative. The evaluation of the RTDSync's behavior concerning performance or resource testing is the second activity to be conducted. This type of evaluation requires careful definition of different data loads, application domains, system complexity scenarios, and architectural complexity. The third is to instantiate the RTDSync to other programming languages to evaluate its behavior in a heterogeneous computational environment, considering the inherent heterogeneity that is characteristic of MApps. The applicability of the RTDSync in an industrial scenario is the fourth activity to be conducted. This activity will enable an evaluation of its behavior when applied in a real-world development and execution environment.

ARTIFACT AVAILABILITY

The artifacts of this paper are available at Moraes et al. [6].

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES).

REFERENCES

[1] Apache. 2025. Apache Kafka. on-line. Avaliable in https://kafka.apache.org, accessed on September 4, 2025.
[2] Artem Bashtovyi and Andriy Fechan. 2023. Change Data capture for migration to event-driven microservices Case Study. In *The 18th International Conference on Computer Science and Information Technologies (CSIT)*. IEEE, Lviv, Ukraine, 1–4. doi:10.1109/CSIT61576.2023.10324262
[3] Yan Bian, Dechao Ma, Qing Zou, and Weirui Yue. 2022. A Multi-way Access Portal Website Construction Scheme. In *The 5th International Conference on Artificial*

Intelligence and Big Data, ICAIBD 2022. Institute of Electrical and Electronics Engineers Inc., Chengdu, China, 589 – 592. doi:10.1109/ICAIBD55127.2022.9820236

[4] Harry Chandra. 2018. Analysis of Change Data Capture Method in Heterogeneous Data Sources to Support RTDW. In *The 4th International Conference on Computer and Information Sciences (ICCOINS)*. IEEE, Kuala Lumpur, Malaysia, 1–6. doi:10.1109/ICCOINS.2018.8510574

[5] Confluent. 2025. Kafka Connect. on-line. Avaliable in https://docs.confluent.io/platform/current/connect/index.html, accessed on September 4, 2025.

[6] Fernando Rodrigues de Moraes, Daniel de Almeida, and Frank José Affonso. 2025. A Real-time data synchronization approach for high-availability micro applications. doi:10.5281/zenodo.17053719

[7] Debezium. 2025. Debezium Stream changes from your database. on-line. Available in https://debezium.io, accessed on September 4, 2025.

[8] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*, Manuel Mazzara and Bertrand Meyer (Eds.). Springer International Publishing, Cham, 195–216. doi:10.1007/978-3-319-67425-4_12

[9] Michael Geers. 2020. *Micro frontends in action*. Manning Publications, New York, NY.

[10] Fikri Muhaffizh Imani, Yohana Dewi Lulu Widyasari, and Satria Perdana Arifin. 2023. Optimizing Extract, Transform, and Load Process Using Change Data Capture. In *The 6th International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*. IEEE, Batam, Indonesia, 266–269. doi:10.1109/ISRITI60336.2023.10468009

[11] Cam Jackson. 2019. Micro Frontends. on-line. Avaliable in https://martinfowler.com/articles/micro-frontends.html, accessed on September 4, 2025.

[12] Anja Kapikul, Dušan Savić, Miloš Milić, and Ilija Antović. 2024. Application Development From Monolithic to Microservice Architecture. In *The 28th International Conference on Information Technology (IT)*. IEEE, Zabljak, Montenegro, 1–4. doi:10.1109/IT61232.2024.10475769

[13] Justas Kazanavičius and Dalius Mažeika. 2023. The Evaluation of Microservice Communication While Decomposing Monoliths. *Computing and Informatics* 42, 1 (May 2023), 1–36. doi:10.31577/cai_2023_1_1

[14] James Lewis and Martin Fowler. 2014. Microservices. on-line. https://martinfowler.com/articles/microservices.html, acessed on September 4, 2025.

[15] MariaDB Foundation. 2025. MariaDB Server: the innovative open source database. on-line. Avaliable in https://mariadb.org, accessed on September 4, 2025.

[16] MongoDB. 2025. MongoDB. on-line. Avaliable in https://www.mongodb.com, accessed on September 4, 2025.

[17] MongoDB. 2025. Replica Set Oplog. on-line. Avaliable in https://www.mongodb.com/docs/manual/core/replica-set-oplog, accessed on September 4, 2025.

[18] Fernando Rodrigues Moraes and Frank José Affonso. 2024. A New Integration Approach to Support the Development of Build-time Micro Frontend Architecture Applications. In *Proceedings of the XXXVIII Brazilian Symposium on Software Engineering* (Curitiba, Brazil) *(SBES '24)*. Association for Computing Machinery, New York, NY, USA, 1–7.

[19] Fernando Rodrigues Moraes, Gabriel Nagassaki Campos, Nathalia Rodrigues Almeida, and Frank José Affonso. 2024. Micro Frontend-Based Development: Concepts, Motivations, Implementation Principles, and an Experience Report. In *The Proceedings of the 26th International Conference on Enterprise Information Systems*. INSTICC, SciTePress, Angers, France, 175–184. doi:10.5220/0012627300003690

[20] Neha Narkhede, Gwen Shapira, and Todd Palino. 2017. *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale* (1st ed.). O'Reilly Media, Inc., Unavaliable.

[21] Oracle. 2025. Java. on-line. Avaliable in https://www.java.com, accessed on September 4, 2025.

[22] Severi Peltonen, Luca Mezzalira, and Davide Taibi. 2021. Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review. *Information and Software Technology* 136 (2021), 106571. doi:10.1016/j.infsof.2021.106571

[23] Kevin Petrie, Dan Potter, and Itamar Ankorion. 2018. *Streaming change data capture: A foundation for modern data architectures*. O'Reilly Media, Newton, Massachusetts, EUA.

[24] PostegreSQL. 2025. PostgreSQL: The World's Most Advanced Open Source Relational Database. on-line. Avaliable in https://www.postgresql.org, accessed on September 4, 2025.

[25] PostgreSQL. 2025. Write-Ahead Logging (WAL). on-line. Avaliable in https://www.postgresql.org/docs/current/wal-intro.html, accessed on September 4, 2025.

[26] Redis. 2025. Introduction to Redis streams. on-line. Avaliable in https://redis.io/docs/latest/develop/data-types/streams, accessed on September 4, 2025.

[27] Redis. 2025. Redis. on-line. Avaliable in https://redis.io, accessed on September 4, 2025.

[28] Chris Richardson. 2018. *Microservices Patterns*. Manning Publications, New York, NY.

[29] Chris Richardson. 2025. Variation: Backends for frontends. on-line. Avaliable: https://microservices.io/patterns/apigateway.html, acessed on September 4, 2025.

[30] Ritu, Shruti Arora, Aanshi Bhardwaj, Ashima Kukkar, and Sawinder Kaur. 2024. A Comparative Analysis of Communication Efficiency: REST vs. gRPC in Microservice- Based Ecosystems. In *International Conference on Emerging Innovations and Advanced Computing (INNOCOMP)*. IEEE, Sonipat, India, 621–626. doi:10.1109/INNOCOMP63224.2024.00107

[31] Dharmendra Shadija, Mo Rezai, and Richard Hill. 2017. Towards an understanding of microservices. In *2017 23rd International Conference on Automation and Computing (ICAC)*. IEEE, Huddersfield, UK, 1–6. doi:10.23919/IConAC.2017.8082018

[32] Antonin Smid, Ruolin Wang, and Tomas Cerny. 2019. Case study on data communication in microservice architecture. In *Proceedings of the Conference on Research in Adaptive and Convergent Systems* (Chongqing, China) *(RACS '19)*. Association for Computing Machinery, New York, NY, USA, 261–267. doi:10.1145/3338840.3355659

[33] Antonin Smid, Ruolin Wang, and Tomas Cerny. 2019. Case study on data communication in microservice architecture. In *The Proceedings of the Conference on Research in Adaptive and Convergent Systems* (Chongqing, China) *(RACS '19)*. Association for Computing Machinery, New York, NY, USA, 261–267. doi:10.1145/3338840.3355659

[34] Srijith, Karan Bantia R, Govardhan N, and Anala M R. 2022. Inter-Service Communication among Microservices using Kafka Connect. In *The 13th International Conference on Software Engineering and Service Science*. IEEE, Beijing, China, 43–47. doi:10.1109/ICSESS54813.2022.9930270

[35] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. 2017. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing* 4, 5 (2017), 22–32. doi:10.1109/MCC.2017.4250931

[36] VMWare Tanzu. 2025. Spring Boot. on-line. Avaliable in https://spring.io/projects/spring-boot, accessed on September 4, 2025.

[37] Sidath Weerasinghe and Indika Perera. 2022. Evaluating the Inter-Service Communication on Microservice Architecture. In *The 7th International Conference on Information Technology Research (ICITR)*. IEEE, Moratuwa, Sri Lanka, 1–6. doi:10.1109/ICITR57877.2022.9992918

[38] Sidath Weerasinghe and Indika Perera. 2023. Optimized Strategy for Inter-Service Communication in Microservices. *International Journal of Advanced Computer Science and Applications* 14, 2 (FEB 2023), 272–279.

[39] Lei Zhang, Ke Pang, Jiangtao Xu, and Bingxin Niu. 2023. High performance microservice communication technology based on modified remote procedure call. *Scientific Reports* 13 (07 2023). doi:10.1038/s41598-023-39355-4