Data distribution and access in a microservices architecture

Luís Manuel Duarte Silva dos Santos Leite
Master in Network and Information Systems Engineering
Computer Science Department
Faculty of Sciences of the University of Porto
2024

Data distribution and access in a microservices architecture

Luís Manuel Duarte Silva dos Santos Leite

Dissertation carried out as part of the Master in Network and Information Systems Engineering

Computer Science Department

2024

Supervisor
Pedro Miguel Alves Brandão, Assistant Professor, Faculty of Sciences of the University of Porto

External Host Supervisor
Gil Silva, Advisor, UPdigital

Acknowledgements

I would like to express my sincere gratitude to Professor Pedro Brandão for supervising this dissertation and offering an intriguing research topic on data propagation, which aligned perfectly with my academic interests. My thanks also go to Gil Silva, who, as co-supervisor, provided invaluable guidance and support throughout this work.

Additionally, I wish to extend my heartfelt thanks to my family, friends, and teachers for their unwavering support, kindness, and the knowledge they have shared with me.

Finally, I would like to dedicate a special thanks to my late grandfather, António, who was a constant source of inspiration. His encouragement and motivation always pushed me to strive for success.

Abstract

This research investigates the feasibility of using the Change Data Capture (CDC) paradigm to improve data propagation and synchronization in microservice architectures that integrate monolithic databases with distributed services. The current system, dependent on a central Oracle database, experienced significant latency during data exchanges with microservices, highlighting the need for a more efficient solution.

By implementing a CDC system with tools such as Apache Kafka, Kafka Connect, and Debezium, the study aimed to determine if the new paradigm could reduce latency and enhance scalability. Results from a service where the solution was implemented demonstrated that the CDC system dramatically improved performance, reducing task completion time from four hours to less than six seconds. However, integrating this system with services that depend on existing REST APIs proved challenging and often not worth the effort, unless those services had significant latency constraints that rendered the current REST API communication unfeasible.

Despite these challenges, the CDC approach showed clear benefits in services where latency reduction and database decoupling are crucial. It offers an efficient solution for synchronizing shared data between newly developed services or existing services that do not rely on the current REST API. However, it is less suitable for complex relational databases, as Debezium captures changes at the table level which does not include, table relationships, limiting its effectiveness in environments with intricate table relationships.

In conclusion, while the CDC-based solution showed potential in improving data distribution and database decoupling, its applicability is more straightforward in less complex environments. Even in complex systems, the approach can yield significant latency reductions, making it a promising method for enhancing data distribution in other or future services.

Keywords: Microservices; Change Data Capture; Debezium; Data Distribution

Resumo

Este trabalho investiga a viabilidade da utilização do paradigma de Change Data Capture (CDC) para melhorar a propagação e sincronização de dados em arquiteturas de microserviços que integram bases de dados monolíticas com serviços distribuídos. O sistema atual, dependente de uma base de dados central Oracle, apresentava uma latência significativa durante as trocas de dados com microserviços, o que evidenciava a necessidade de uma solução mais eficiente.

Ao implementar um sistema CDC com ferramentas como Apache Kafka, Kafka Connect e Debezium, o estudo procurou determinar se o novo paradigma poderia reduzir a latência e aumentar a escalabilidade. Os resultados, provenientes de um serviço onde a solução foi implementada, demonstraram uma melhoria dramática no desempenho, reduzindo o tempo de conclusão de tarefas de quatro horas para menos de seis segundos. No entanto, a integração deste sistema com serviços que dependem de APIs REST existentes revelou-se desafiadora e muitas vezes não compensava o esforço, a menos que esses serviços tivessem constrangimentos de latência significativos que tornassem a comunicação via API REST inviável.

Apesar destes desafios, a abordagem CDC mostrou claros benefícios em serviços onde a redução de latência e o desacoplamento da base de dados são cruciais. Apresenta-se como uma solução eficiente para sincronizar dados partilhados entre novos serviços ou serviços existentes que não dependem da API REST atual. No entanto, é menos adequada para bases de dados relacionais complexas, já que o Debezium capta as alterações ao nível das tabelas, sem incluir as relações entre elas, limitando a sua eficácia em ambientes com relações complexas entre tabelas.

Em conclusão, embora a solução baseada em CDC tenha mostrado potencial na melhoria da distribuição de dados e no desacoplamento da base de dados, a sua aplicabilidade é mais direta em ambientes menos complexos. Mesmo em sistemas complexos, a abordagem pode resultar em reduções significativas de latência, tornando-se uma metodologia

promissora para melhorar a distribuição de dados em outros serviços ou serviços futuros.

Palavras-chave: Microserviços; Captura de Dados Alterados; Debezium; Distribuição de Dados

Contents

List of Figures

Glossary

API	Application Programming Interface
CDC	Change Data Capture
DBMS	Database Management System
DML	Data Manipulation Language
ESB	Enterprise Service Bus
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
ORM	Object-Relational Mapping
REST	Representational State Transfer
SQL	Structured Query Language

Chapter 1

Introduction

In modern software engineering, the microservices architecture has emerged as a preferred approach for building scalable and flexible systems. By decomposing applications into smaller, independently deployable services that communicate via messages to ensure loose coupling, microservices offer significant benefits, including enhanced reusability, scalability, resilience, and maintainability.

However, managing data consistency and distribution across these decentralized services presents considerable challenges. In traditional monolithic architectures, all system components are tightly integrated into a single unified codebase. Modules typically communicate within the same machine, without relying on a network, and data consistency is maintained within a single centralized database.

In contrast, microservices often utilize distributed data storage and depend on network communication to transmit data. This distributed nature complicates data synchronization and consistency, as each microservice may maintain its own database, leading to challenges in ensuring that data remains consistent across the entire system.

When microservices interact with a monolithic database, they can encounter bottlenecks. Microservices are built to be independent, loosely coupled, and to manage their own data, whereas a monolithic database is centralized. This disparity can cause performance degradation and latency issues when microservices rely on the monolithic database for communication.

Despite these challenges, one of the most popular approaches is to begin with a monolithic architecture and gradually transition to a microservices-based architecture [1]. This method allows organizations to initially benefit from the simplicity of a monolithic system while progressively adopting the scalability of microservices. However, this transition is

complex, as many systems have numerous dependencies. As these systems grow, moving to a more modular, distributed data architecture becomes essential . This thesis explores solutions to improve data propagation and consistency in microservices environments, focusing on using CDC technology to address challenges in hybrid architectures.

One of the critical challenges in microservices is efficiently propagating data changes across multiple services without introducing significant latency or inconsistencies. The Change Data Capture (CDC) paradigm has emerged as a promising solution to this problem by enabling the detection and propagation of database changes to other services.

1.1 Motivation & Approach

The motivation for this research arises from the need to address specific challenges encountered in the practical implementation of microservices within information system centre where this project was conducted. As microservice architectures have become increasingly common, the effective distribution and the synchronization of data across multiple services have proven to be complex and critical issues. The current system faced challenges in propagating data with sufficiently low latency to maintain normal functionality of the services it supported, which prompted the exploration of more efficient methods for handling data distribution. By investigating the application of CDC paradigm using well-known tools like Apache Kafka and Debezium, this project aims to improve the data propagation process, thereby enhancing the overall reliability and performance of the microservices architecture in a real-world setting.

The primary objective of this thesis is to explore and evaluate the use of the CDC paradigm to propagate data across microservices. The goals of this research are:

- To design and develop a pilot project that utilizes the selected CDC technology for data propagation in a microservices context.
- To assess the feasibility and effectiveness of integrating CDC-based data propagation into an existing microservices systems, where a monolithic database contains most of the data and the microservices rely on smaller databases.
- To identify and analyze the challenges and limitations of using CDC technology in a production environment.

- To determine the overall viability of this approach for broader adoption within the organization where the project was conducted.

The goals outlined above form the foundation of this research and drive the investigation. By achieving these objectives, the project aims to provide valuable insights into the effectiveness of CDC technology for microservice data propagation. This research will contribute key findings to the field, particularly in practical implementation, performance evaluation, and addressing real-world challenges. The following contributions summarize the outcomes of this work and its potential for broader application.

- Developed and implemented a CDC-based solution using tools like Apache Kafka, Kafka Connect, and Debezium for data synchronization and propagation in a hybrid architecture combining monolithic databases and microservices.
- Evaluated the impact of the CDC system on performance, particularly in terms of reducing latency and improving scalability in a real-world setting.
- Analyzed the challenges encountered during the integration of the CDC system with existing services and identified conditions under which the system provided clear benefits.
- Provided recommendations for adopting CDC technology and outlined its limitations in environments with complex database relationships.

These contributions enhance the understanding of how CDC can improve data propagation in microservices architectures, offering a promising approach for reducing latency and boosting system scalability.

1.2 Document Structure

The present document is organized as follows: Chapter 2 provides an overview of the CDC paradigm and related concepts necessary to understand the problem. Chapter 3 discusses the design and implementation of our CDC system, including the chosen technologies. Chapter 4 evaluates our CDC system by integrating it with existing services and reports our findings. Finally, Chapter 5 presents our final remarks on the developed system, its limitations, how well it meets our goals, and outlines future work.

Chapter 2

Background

This chapter introduces key concepts that support the main topic of CDC in microservice architectures. Section 2.1 introduces the concept of software architecture and discusses both monolithic and microservice architectures, including their advantages and drawbacks. Section 2.2 addresses how services communicate within a microservice architecture. Section 2.3 explores data consistency challenges in microservices and outlines common patterns to manage these issues. Section 2.4 delves into CDC paradigm, explaining its role in efficiently propagating data changes across microservices. Finally, in Section 2.5, we connect our problem to the concepts introduced earlier and set the tone for the following chapter.

2.1 Software Architecture

Software architecture refers to the high-level structure of a software system, outlining the organization of its components and the relationships between them. It typically plays a key role as a bridge between requirements and code, serving as a blueprint that guides the system's construction and evolution. By establishing a clear architectural framework, software architects can ensure that the system meets both functional and non-functional requirements, such as performance, scalability, and security. The architecture defines the elements from which the system is built, interactions among those elements, patterns that guide their composition, and constraints on those patterns [2].

2.1.1 Monolithic Architecture

A monolith is a software application in which modules cannot be executed independently. In a monolithic architecture, all system components are tightly coupled into a single,

unified codebase. This approach can simplify development and deployment for smaller applications and offers several advantages. However, it can lead to significant challenges as the system grows in complexity. The key advantages of monolithic architectures include [3]:

- No data consistency problems: All communication occurs internally without relying on inter-process mechanisms. This results in faster communication and avoids compromises between availability, latency, and consistency.
- Easier to start developing: Monolithic architectures are the easiest and fastest to deploy initially, as they lack the additional overhead typically associated with other architectural styles.

The key drawbacks of monolithic architectures include [4]:

- Complex Maintenance and Evolution: large-size monoliths are difficult to maintain and evolve due to their complexity. Tracking down bugs requires long perusals through their extensive codebase.
- Dependency Hell: adding or updating libraries results in inconsistent systems that do not compile or run, or worse, misbehave.
- High Downtime for Changes: any change in one module of a monolith requires rebooting the whole application. For large-sized projects, restarting usually entails considerable downtimes, hindering development, testing, and maintenance.
- Sub-optimal Deployment: the deployment of monolithic applications is usually sub-optimal due to conflicting resource requirements among the constituent modules. Developers often have to compromise with a one-size-fits-all configuration, which is either expensive or sub-optimal with respect to the individual modules.
- Limited Scalability: Monoliths limit scalability. The typical strategy for handling increased inbound requests is to create new instances of the entire application and split the load among them. However, if the increased traffic stresses only a subset of the modules, allocating new resources for the other components becomes inefficient.
- Technology Lock-in: Monoliths also represent a technology lock-in for developers, who are bound to use the same language and frameworks of the original application.

2.1.2 Microservice Architecture

The microservice architecture is an architectural style in which all modules function as microservices. A microservice is a cohesive and independent process, where ”cohesive” means that it implements only functionalities strongly related to the specific concern it is meant to model. These modules adhere to the single responsibility principle (SRP), meaning each service focuses solely on one functionality. The modules interact via messages to ensure loose coupling, so changes in one module do not necessitate updates to others. This principle provides a clear guideline for designing and implementing distributed applications, allowing developers to focus on a few, cohesive functionalities. Even higher-level microservices, which coordinate the functionalities of other microservices, follow this principle.

When deciding how databases will be used in microservices, there are generally three approaches:

- One database per microservice.
- One schema per microservice.
- One set of tables per microservice.

As noted in a survey by [5], the first option, one database per microservice, appears to be the most common method. This approach aligns well with the principles of microservices, promoting data encapsulation and independent scalability, although it may introduce complexity in data management across services.

To better understand these concepts, refer to figure 2.1, which illustrates the differences between a traditional monolithic architecture and a microservice architecture. In a monolithic architecture, all components are tightly coupled within a single application, leading to challenges in scalability, maintenance, and flexibility. In contrast, the microservice architecture breaks down these components into independent services, each responsible for a specific functionality. These services can be scaled, deployed, and maintained independently, offering greater flexibility and efficiency.

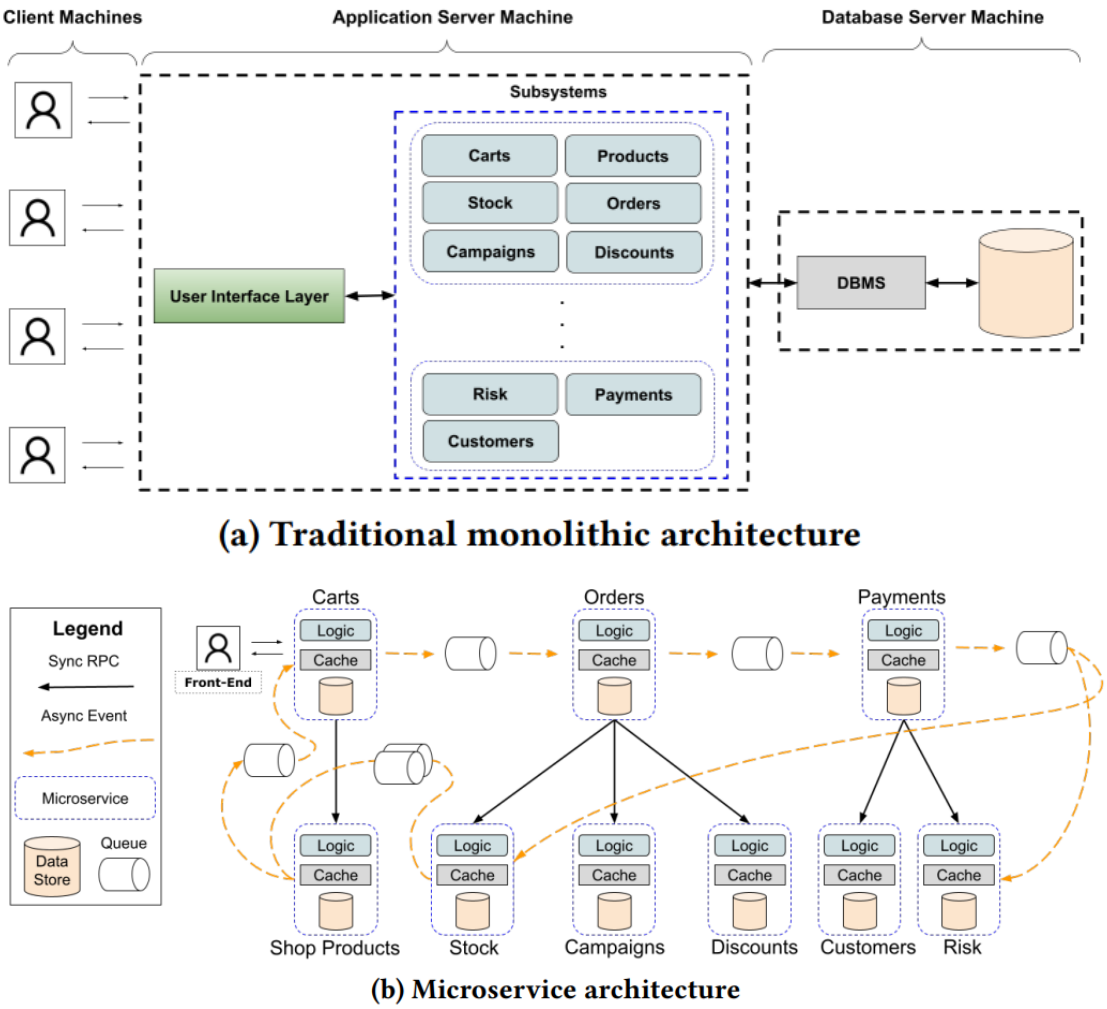

FIGURE 2.1: Monolithic vs. microservice architectures. [5]

With these characteristics, the microservice architecture addresses many issues inherent in monolithic applications [4]:

- Reduced Downtime: Modifying a module in a microservice architecture does not require rebooting the entire system. Only the microservices within that module need to be restarted, resulting in very short re-deployment downtimes and minimizing disruptions during development, testing, and maintenance.
- Optimal Resource Utilization: Microservices lend themselves naturally to containerization, giving developers the freedom to configure deployment environments that best meet their needs in terms of cost and quality of service.

- Scalability: Scaling a microservice architecture does not require duplicating all components. Developers can deploy or remove instances of specific services based on their load, making resource allocation more efficient.

- Avoidance of Technology Lock-In: The only constraint on a network of inter-operating microservices is the technology used for communication. Beyond that, microservices impose no additional lock-in, allowing developers to choose the optimal languages, frameworks, and resources for implementing each service.

While microservices mitigate many issues found in monolithic applications, they introduce their own set of challenges:

- Increased Complexity: Managing a distributed system of microservices is inherently more complex than managing a monolithic system. The system must handle network latency, message serialization, and other complexities related to inter-service communication.

- Overhead in Deployment and Operations: Deploying and managing a large number of microservices can lead to operational overhead. It requires sophisticated deployment pipelines, monitoring, and logging systems to ensure each microservice runs correctly and efficiently.

- Difficulty in Data Management: In a microservice architecture, data is often distributed across different services, complicating transactions and consistency. Ensuring data consistency across microservices can be challenging and may require implementing complex patterns like eventual consistency.

- Inter-Service Communication Latency: Communication between microservices typically occurs over a network, introducing latency compared to intra-process communication in a monolithic architecture. This can impact system performance, especially in latency-sensitive applications.

2.1.3 Hybrid System

As discussed in sections 2.1.1 and 2.1.2, neither monolithic nor microservice architecture suit every context. Monolithic architecture offers low complexity, making it ideal for small systems with few concurrent users. In contrast, while microservice architecture is more

horizontally scalable and suited for large systems, its complexity can be excessive for many applications [3, 6]. A reasonable compromise is to start with a monolithic architecture and transition to a microservice architecture as the system grows and faces higher demands. This strategy often results in a hybrid system where both architectures coexist, enabling organizations to manage complexity while gradually adopting microservices. This phased migration has proven successful in industrial contexts, offering a practical balance between stability and scalability during the transition [7]. However, for success, proper module identification and careful planning of deployment and communication strategies are essential [8, 9].

2.2 Communication Patterns in microservices

Services often need to communicate with each other to fulfill business operations. This inter-service communication is facilitated through Inter-Process Communication (IPC) mechanisms. The choice of IPC mechanism is critical and depends on the nature of the interaction between the services. There are two key dimensions to consider when selecting an interaction style for communication between microservices. The first dimension considers whether each client request is processed by a single service (one-to-one) or by multiple services (one-to-many). The second dimension addresses whether the interaction is synchronous or asynchronous [10].

2.2.1 Synchronous Communication

In synchronous communication, the client sends a request to a service and waits for a response before continuing. This type of communication requires both the client and the service to be operational simultaneously for the interaction to succeed. If the service is unavailable or does not respond within a specified timeout period, the communication fails. Synchronous communication is typically used when an immediate exchange of information is needed, and both parties must coordinate their operations closely. HTTP-based REST and RPC-based technologies, such as Apache Thrift or gRPC, are popular choices for this approach [9]. These tools are commonly used for both one-to-one and one-to-many synchronous communication, with the distinction based on the number of connections the client establishes. However, it is important to note that synchronous one-to-many

communication is not recommended when the number of receivers is large, as it can create a bottleneck.

2.2.2 Asynchronous Communication

In asynchronous communication, the client sends a request to a broker, which is responsible for delivering the requests. The client does not block waiting for the response; instead, the response from the service is delivered to the client later. This allows communication to occur without all parties needing to be online and available simultaneously. Asynchronous communication is often used in microservices that implement Event-Driven Architectures. While there are various options, Message-Oriented Middleware is the most popular choice. Popular one-to-one technologies include ActiveMQ and RabbitMQ (which, although primarily one-to-one, can also function as publish-subscribe systems for one-to-many communication). Popular one-to-many technologies include publish-subscribe systems such as Apache Kafka and Google Pub/Sub [9]. It is important to note that implementing this type of communication generally requires more resources, as it involves deploying and maintaining a dedicated broker.

2.2.3 Hybrid Communication

IPC can also employ a hybrid approach that does not fit neatly into the synchronous or asynchronous categories. An example of this is using a combination of HTTP and a message queue. In this model, a service initially attempts synchronous communication via HTTP. If the target service is unavailable, the communication can fall back to an asynchronous pattern, placing the request in a message queue for later processing [9].

2.3 Data Consistency in microservices

In microservices architectures, maintaining consistent and reliable data across distributed systems is one of the most significant challenges. Each service typically has its own database, which makes ensuring data consistency across the entire system complex and potentially prone to issues. Unlike monolithic architectures, where a single centralized database can enforce strong consistency, microservices rely on independent services that communicate asynchronously, leading to potential delays or conflicts in data propagation.

This chapter explores the trade-offs between strong and eventual consistency in distributed systems. In Section 2.3.1, we discuss the CAP theorem, the PACELC theorem, and how they illustrate the balance between consistency, availability, and latency. We also examine the differences between strong consistency and eventual consistency, with a focus on their application in microservices architectures.

In Section 2.3.2, we introduce common patterns used in microservices to manage data consistency, such as sagas, event-driven architectures, and two-phase commit, which help developers maintain data integrity while addressing the distributed nature of microservices.

2.3.1 Strong and eventual consistency

Brewer's CAP theorem [11] states that a networked shared-data system can have at most two of the following three properties: consistency (C), which ensures a single up-to-date copy of the data; availability (A) of the data; and tolerance to network partitions (P).

When building microservices, the general goal is to avoid shared data (thus forfeiting partition tolerance) by carefully defining the boundaries of each microservice. However, in cases where shared data does exist, a trade-off between consistency and availability can occur frequently within the same system, varying across subsystems or even depending on the specific operation. It is also important to note that the CAP theorem addresses high availability, strong consistency, and partition tolerance, but there are many levels of consistency, and even partitions can have nuances, such as disagreements within the system about whether a partition exists. By explicitly managing partitions, designers can optimize both consistency and availability, achieving a balanced trade-off among all three properties.

As mentioned above, systems that are partitioned and offer availability (AP systems, which are common in a microservice architecture) cannot offer consistency. To tackle this issue a lot of these systems function under a guarantee of eventual consistency. Eventual consistency refers to the guarantee that the system is heading towards a consistent state, meaning that if no more update requests reach the system, then the system will eventually reach a consistent state [12]. The main idea behind it is that data is replicated across participants, each participant performs updates locally, then other mechanism are needed to propagate local updates to other participants and to resolve inconsistencies between them so that they agree on the outcome and eventually converge [13].

This trade-off is further explained by the PACELC theorem [14], which builds on the CAP theorem. PACELC states that in the presence of a partition (P), systems must choose between availability (A) and strong consistency (C). Even when no partition exists (E), there is still a trade-off between latency (L) and strong consistency (C). Systems that adopt eventual consistency are prioritizing lower latency and availability over strong consistency, allowing operations to proceed quickly while accepting that consistency will only be achieved over time. In contrast, systems that prioritize strong consistency ensure that all nodes have up-to-date data immediately, but at the cost of increased latency and potentially reduced availability during network partitions.

2.3.2 Common patterns

To address the data consistency issues discussed, several strategies are employed to ensure either strong or eventual consistency, depending on the system's requirements. The most common approaches include:

- Two-Phase Commit (2PC) for Distributed Transactions: a widely-used method for ensuring strong consistency in distributed transactions. In 2PC, a coordinator service ensures that all participating services agree on the final outcome of a transaction (either commit or rollback). This protocol involves two phases: the prepare phase, where each participant votes on the transaction's outcome, and the commit phase, where the transaction is either completed or aborted based on the votes. While 2PC can guarantee consistency, it is also associated with high latency and potential bottlenecks, particularly in systems requiring high availability and performance [15].

- Event-Driven Architecture (EDA): another approach to managing data consistency in microservices, particularly for systems that favor eventual consistency. In EDA, services communicate through events, which represent changes in state or data. When a service updates its data, it emits an event that is consumed by other services interested in that event. This approach allows services to react to changes asynchronously, thereby decoupling the services and improving system resilience. However, achieving eventual consistency requires careful handling of event ordering and potential event loss [16].

- Sagas: The Saga pattern is a method of managing distributed transactions without the need for a global transaction coordinator, which can be a single point of failure in

systems using 2PC. In a Saga, a distributed transaction is broken down into a series of smaller, local transactions, each managed by a different service. If a transaction fails at any point, compensating transactions are executed to undo the changes made by previous transactions. This pattern is particularly useful in long-running transactions where partial failures are expected. Sagas are well-suited for systems requiring eventual consistency and high availability [15, 17].

- CQRS (Command Query Responsibility Segregation): CQRS is a pattern that separates the read and write operations of a service into distinct models. This separation allows for optimized handling of commands (which modify data) and queries (which retrieve data). By using different models for reading and writing data, CQRS can improve scalability and performance, especially in systems with high read and write loads. While CQRS itself does not directly address data consistency, it complements other patterns like Event Sourcing, where events are used to rebuild the state of a service, thus helping to manage consistency in distributed systems.

Each of these approaches obeys the CAP theorem, offering different trade-offs between consistency, availability, and performance. The choice of which strategy to implement depends on the specific requirements of the system, such as the need for strong consistency versus eventual consistency, the tolerance for latency, and the complexity of the transactions involved.

2.4 Change Data Capture

Change Data Capture (CDC) is defined by [18] as the process of identifying when data in a data source system has changed and prompting the system to act based on these changes in downstream processes or reminders. While CDC is widely used in ETL (Extract, Transform, Load) processes [19, 20], it also plays a crucial role in scenarios requiring the synchronization of data across related systems, such as in Event-Driven Architectures (EDA). In these architectures, data changes can be propagated as events across various microservices. This approach helps maintain data integrity across different target systems by converting database changes into actionable events.

When compared to batch processing, CDC tools can be seen as good data integration alternatives that can provide the following benefits:

- Capturing data changes for a wide variety of Database Management System (DBMS)

- Reducing load on the DBMS
- Ensuring data consistency across decoupled systems
- Reducing latency in synchronization

Several mechanisms exist that a CDC system can implement, but they all follow one of two architectural styles: push or pull. In Pull CDC, data updates are extracted by using a database resource. In Push CDC, data updates are generated by intercepting Data Manipulation Language (DML) statements on their way to the data source. [21]

2.4.1 Pull-Based CDC

The Pull CDC extracts changes by using database resource, meaning it must query the data source to extract the most recent changes. A Pull CDC uses the following mechanisms to find new changes: Change Identification, Change Capture, and optionally, Change Delivery [21].

2.4.1.1 Change Identification

This mechanism identifies DML statements and flags the changes. The most commonly used techniques to detect these changes are log scanning, triggers, and timestamp records. These techniques flag the changes made, with the flagging process being done either by directly analyzing the DML statements or by using intermediary (shadow) tables in the case of triggers [21].

2.4.1.2 Change Capture

This mechanism occurs after Change Identification and is highly reliant on it, making the method used depend entirely on the method from the previous step. One important consideration in this step is ensuring that only the latest set of changes is captured; otherwise, the volume of data to capture would grow tremendously. This step involves polling and extracting the latest set of changes. Depending on the type of mechanism used, the polling and extraction process can vary [21].

- For log identification, the capture process will periodically poll and capture the newest set of changes. To ensure this, the capture process must keep track of its position in the log file so that it only captures the changes made after that position.

- For trigger identification, the capture process will be polled after certain DML statements. To ensure that only the newest set of changes is captured, it uses the shadow table and selects records where the primary key is greater than the highest primary key value from the set of records retrieved in the previous extraction.
- For timestamp identification, the capture process periodically polls and captures the newest set of changes. To ensure that only the newest set of changes is captured, it selects rows where the timestamp value is greater than the highest timestamp value of the record from the previous extraction.

2.4.1.3 Delivery Process

This final step is optional, as many of these pull-based CDCs run at the target system. Its objective is to deliver the sets of changes to the target systems. These may involve simple protocols with the source system address embedded in the code or more complex middleware systems that offer their own set of advantages and drawbacks, as we will see in section 2.4.3.

2.4.2 Pull-Based CDC Algorithms

Pull-based CDC systems rely on periodic polling to detect and extract changes from the source database. The general process includes Change Identification, Change Capture, and optionally Change Delivery. However, the specific approach can vary significantly depending on the algorithm used. The following sections describe key Pull-based CDC algorithms—Log-Based, Trigger-Based, and Timestamp-Based—each with its own strengths, challenges, and use cases.

2.4.2.1 Log-Based CDC

Log-Based CDC identifies changes in a database by parsing the logs generated by a DBMS during its regular operation. These logs document various database activities, such as transactions, updates, and schema changes. The specific logs parsed depend on the DBMS and the CDC tool in use. Common logs include Write-Ahead Logs (WAL), redo logs, binary logs (bin logs), and transaction logs. This list is not exhaustive, as many CDC solutions exist, each tailored to different DBMSs with unique log structures. While many logs serve similar purposes, they are often referred to by different names across DBMSs.

Log-based CDC systems fall into two main categories: those with DBMS-integrated log scanners and those using external log scanners. In DBMS-integrated systems, the CDC process is embedded within the DBMS itself, functioning as a built-in service that directly interacts with the database's internal processes. This service extracts the changes from the Data Manipulation Language (DML) operations recorded in the logs and writes them to a staging area. External log scanners, on the other hand, operate independently, using tools that access and parse the database logs from the DBMS.

Log parsing is the predominant approach for detecting database changes and is used by solutions such as Debezium, Oracle GoldenGate, Estuary, and AWS Database Migration Service (DMS). This method is popular because it has minimal impact on the performance of the data source and is non-intrusive.

However, implementing log-based CDC is challenging. Each DBMS has its own Data Manipulation Language (DML) syntax, log structure, and available log types, requiring CDC systems to be highly adaptable and tailored to each environment. This complexity presents a significant barrier to the widespread adoption of standardized solutions, as noted in [18].

2.4.2.2 Trigger-Based CDC

Trigger-Based CDC operates by creating triggers after specific DML statements, such as inserts, updates, and deletes. These triggers then execute actions that update additional tables, which are used to track and generate the changes. Typically, Trigger-Based CDC can be implemented using one of two ways: the shadow table method or the trigger method. In the shadow table method, each table that requires replication has a corresponding shadow table. The shadow table contains the state of the original table at the time of the last change capture. To retrieve the most recent changes, the current state of the table is compared with the shadow table. "Subtracting" the data in the shadow table from the original table reveals the inserted and updated data, while "subtracting" the original table's data from the shadow table shows the deleted data. Figure 2.2 shows a diagram that illustrates this process.

While the shadow table method is an intuitive way to obtain incremental data (i.e., data changes), it is only suitable for small datasets. For large databases, the capture speed can be significantly reduced, and the method may fail to capture intermediate changes [22].

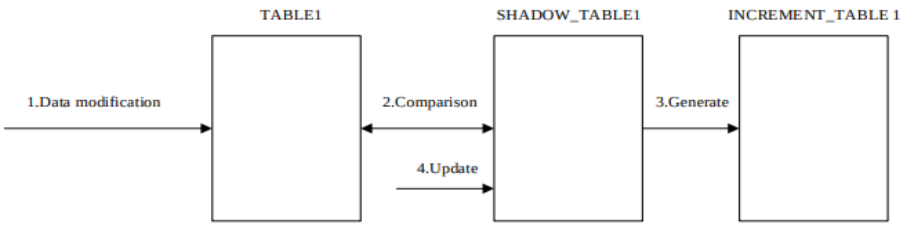

FIGURE 2.2: Diagram of shadow table method. [22]

Among the two methods, the trigger implementation is more commonly used in CDC systems because it scales better and uses fewer resources. Instead of maintaining a shadow table for each table, this method relies on a unique identification table that stores the incremental data generated by each DML operation. This unique identification table records the primary key information, the type of operation (insert, update, delete), the table identifier, and the captured incremental data [22]. Figure 2.3 illustrates how the unique identification table captures and tracks data changes without the overhead of comparing entire tables.

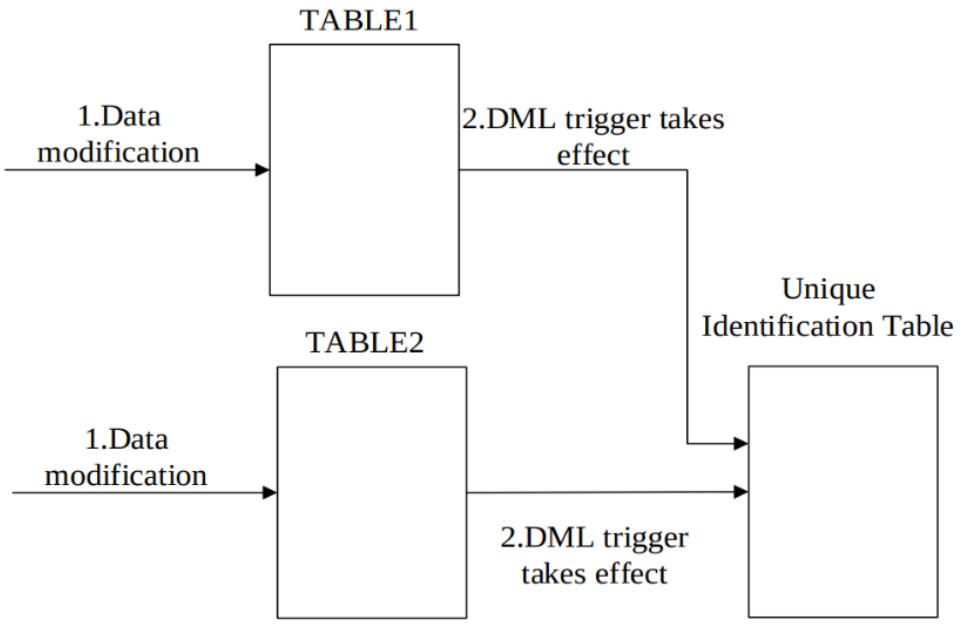

FIGURE 2.3: Diagram of trigger and unique identification table method. [22]

2.4.2.3 Timestamp-Based CDC

This method involves modifying the schema of the source by adding a timestamp field to each table, then periodically polling the data source to compute the changes. It is

the simplest method, making its technical implementation straightforward. However, this approach is highly intrusive, offers slow performance, fails to capture intermediate changes, and lacks support for handling schema changes.

2.4.3 Publish and Subscribe Queues for Pull-Based CDC

Many CDC systems, after pulling changes from the source systems, use middleware like a Publish-Subscribe system or an Enterprise Service Bus (ESB) for their delivery process. This approach offers several key advantages:

- Decoupling the source and target systems: This allows for the integration of various CDC techniques across different DBMSs without affecting the target systems. For example, switching from a Timestamp-Based CDC to a Log-Based CDC can be done without impacting the targets.
- A Pub-Sub or ESB system allows multiple sources and targets to operate simultaneously without the bottlenecks that would occur if each target had to individually poll the source.
- Fault tolerance and reliability: These systems are designed to handle failures gracefully, ensuring the robustness of the data delivery process.

Additionally, many systems that implement CDC already use event streaming platforms like RabbitMQ or Kafka for handling data transmission. These platforms are well-suited for managing high volumes of data, making them a natural fit for integrating CDC processes. When a system is already using an event streaming platform, it can leverage this infrastructure to handle CDC events, reducing the need for additional components. This integration simplifies the architecture, as the event streaming platform can manage both general data streams and CDC events, minimizing the overhead of maintaining separate systems for data propagation [18].

2.4.4 Push-Based CDC

Push-Based CDC extracts information by intercepting DML statements before they reach the DBMS. After these statements are applied to the DBMS, the CDC system then propagates the resulting changes to the target systems. This mechanism consists of two consecutive processes: change identification and change delivery [21].

Change Identification In this process, the CDC system identifies changes made to the DBMS before they are applied. There are two primary methods for this identification: Network-Based CDC, which analyzes network traffic using a network sniffer, and Application-Based CDC, which detects changes within the application interacting with the DBMS. A crucial part of this process is confirming that changes are successfully applied to the source DBMS before proceeding. Failure to do so can result in data consistency issues.

Change Delivery Once changes are identified and confirmed, they are propagated either directly to the target system or through middleware such as a Pub-Sub system or an ESB

2.4.4.1 Challenges and Considerations

Overall, Push-Based CDCs address many of the limitations found in Pull-Based CDCs, such as latency, performance issues related to polling, and the risk of missing intermediate changes. As noted by Eccles [21], Push-Based CDCs are seen as a promising direction for future development. However, both Network-Based and Application-Based CDCs have significant design flaws.

Network-Based CDC systems rely on network sniffers to intercept DML statements before they reach the DBMS. This approach is effective only if the source DBMS communicates in plain text or if the CDC system has access to the encryption keys securing the connection. While useful in some scenarios, its dependency on plain-text communication and encryption keys limits its broader applicability.

In contrast, Application-Based CDC systems, though they offer lower latency, are impractical for real-world use due to their intrusiveness. Each application accessing the DBMS would need CDC routines implemented, leading to excessive coupling between the applications and the CDC code. In large-scale environments with hundreds of applications interacting with the DBMS, this creates significant maintenance challenges.

When the database schema evolves or new data targets require change data, each application's CDC routines must be updated. This introduces a high maintenance burden, as even minor database changes necessitate updates across all applications. The tight coupling of applications to the CDC logic makes the system difficult to scale.

Therefore, while Application-Based CDC systems may provide performance benefits, their practical limitations—particularly the high maintenance requirements and tight

coupling—make them unsuitable for environments with numerous applications accessing the same DBMS.

2.5 Conclusion

As mentioned in Section 1.1, the current system faced latency issues related to data exchange between services. This hybrid architecture consists of a monolithic service that is gradually being migrated to microservices. The monolithic service utilizes an Oracle database as the primary data store, holding the majority of the data being distributed. Communication between this database and other services occurs through a REST API. Several cases of latency problems were identified, particularly those related to the REST API used by the Oracle database.

The goal of this chapter was to provide context to better describe the problem and assist in selecting the appropriate tools. We explored fundamental concepts of microservice architectures, including their communication patterns, data consistency challenges, and the application of the Change Data Capture (CDC) paradigm to address data propagation issues. Additionally, we examined different approaches within CDC, highlighting their respective advantages and drawbacks.

In the next chapter, we will transition from theoretical foundations to the practical implementation of CDC in a real-world hybrid environment. We will discuss the tools and methodologies used to address the data synchronization challenges outlined in this chapter.

Chapter 3

Methodology & Implementation

In this chapter, we outline our approach to finding a solution for handling data propagation between services. In Section 3.1, we detail the requirements our system needs to fulfill and what can be extrapolated from them. Section 3.2 describes the technologies used and how they align with our requirements. Finally, in Section 3.3, we present our contribution: the creation of a CDC system designed to reduce the latency associated with data propagation between services within the company.

3.1 Requirements

As mentioned in Section 2.5, several services experienced delays when receiving the data necessary for normal operations. These delays were specifically observed in services that communicated with the Oracle database via the REST API. Based on this context, we defined the following requirements for our CDC system:

- Support a variety of databases, including Oracle, as it is the primary database holding the majority of the data that needs to be distributed.
- Enable the transmission of data to multiple data sinks simultaneously, since several systems could benefit from the CDC system.
- Ensure a delay low enough to avoid disrupting the normal functionality of applications with which it integrates.
- Avoid data consistency issues in the systems it serves to avoid introducing additional complexity.

From the first and second requirements, we can deduce that the chosen CDC solution should include a delivery mechanism that decouples the source and sink while supporting multiple sources and sinks.

Ideally, our system would employ a push-based CDC to offer the lowest possible delay, ensuring it meets the minimum delay requirements of current and future services. However, as concluded in Section 2.4.4.1, the current push-based CDC alternatives are flawed and not suitable for our purposes. Therefore, our CDC tool must be pull-based and support a wide variety of databases, including Oracle. Figure 3.1 provides a rough sketch of how the system will be structured based on our core requirements and the conclusions drawn.

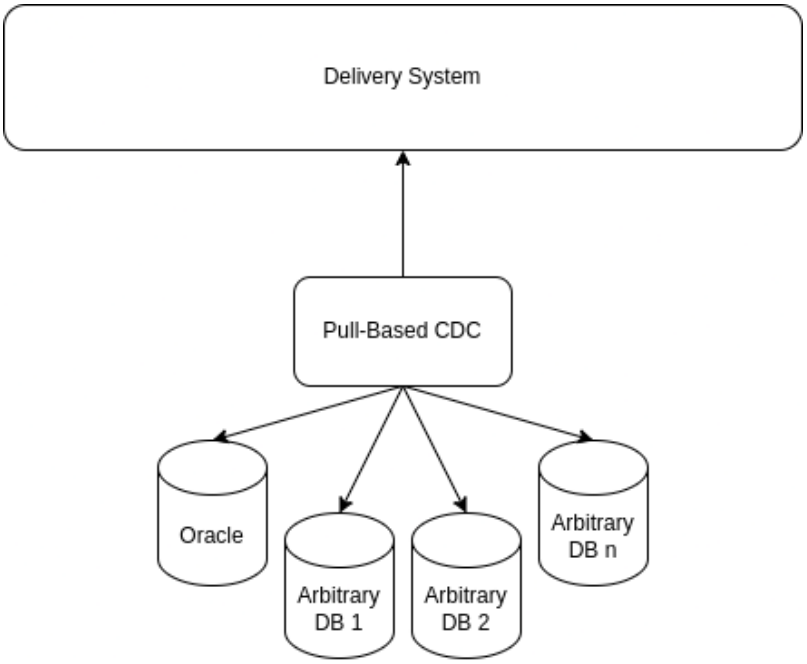

FIGURE 3.1: Early architecture of the CDC system

3.2 Review of Core Technologies

As discussed in the previous section, we needed to select a pull-based CDC tool that supports Oracle and a delivery system that can handle various types of sources and sinks, including our CDC tool.

For this purpose, we chose Apache Kafka, a distributed event streaming platform, as our delivery system due to its popularity and its integration with Kafka Connect. Kafka Connect is a framework for Apache Kafka that supports several CDC tools, making it

compatible with a wide range of data sources while also supporting multiple data sinks. Specifically, we focused on Debezium, a collection of log-based CDC tools that capture all changes made to a data source and stream them to Apache Kafka (via Kafka Connect). Additionally, we employed a schema registry, which is an important tool for Kafka Connect that will be explained in Section 3.2.4.

The rest of this section will provide a detailed explanation of these technologies, starting with the core components: Apache Kafka (Section 3.2.1), Kafka Connect (Section 3.2.2), and Debezium (Section 3.2.3). Next, we will explain the schema registry (Section 3.2.4) and its role within Kafka Connect.

3.2.1 Apache Kafka

The delivery system used is Apache Kafka [23], an open-source distributed event streaming platform. In this section, we will discuss some of the fundamental concepts of Kafka, including topics, partitions, producers, and consumers.

Kafka provides a scalable publish-subscribe messaging system. The primary mechanism behind its scalability is the partitioned nature of topics. A topic is a category or feed name where messages are organized and durably stored. Topics are composed of partitions, with each partition being an ordered, immutable sequence of messages continually appended to a commit log. Figure 3.2 shows the structure of a Kafka topic, divided into multiple partitions. Messages are assigned a unique ID called the offset and are written to the end of each partition, with new data appended sequentially.

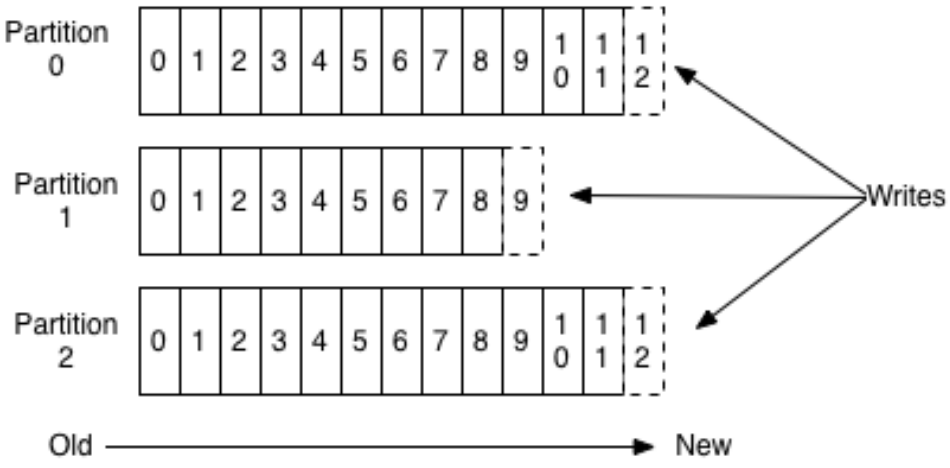

FIGURE 3.2: The anatomy of a Kafka topic. [23]

Each partition has a server that acts as a leader and zero or more servers that act as followers. The leader handles all write and read requests, while the followers replicate the leader's data. Producers publish data to the topics of their choice, and they are responsible for assigning messages to specific leader partitions within a topic. If the leader partition fails to be "alive", a follower partition that is "alive" will be elected as the new leader, ensuring fault tolerance. For a node to be considered "alive", it must maintain its session with the controller and, if it is a follower, replicate the writes happening on the leader without falling "too far" behind.

Consumers in Kafka organize themselves into consumer groups, which subscribe to topics. A consumer group can contain several consumers or only one. This flexibility allows Kafka to function like a queue, distributing messages among consumers, or like a publish-subscribe model (where each consumer is in its own consumer group), enabling all consumers to receive all messages. Figure 3.3 illustrates how Kafka partitions interact with consumer groups.

The final topics to address are Kafka Brokers and Kafka Controllers. Kafka Brokers are servers that store partitions and act as leaders or followers for the partitions they manage. Kafka Controllers are processes that run the Apache Kafka Raft (KRaft) protocol, which Kafka uses for metadata storage and leader election purposes.

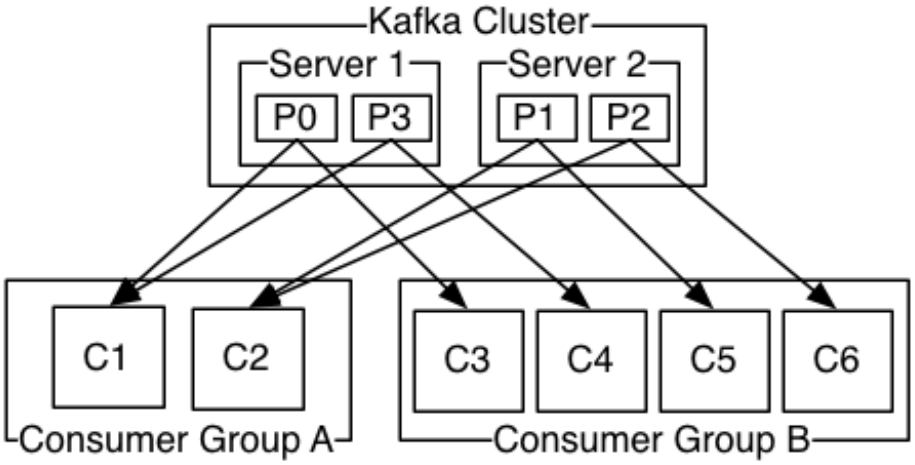

FIGURE 3.3: A 2 server Kafka cluster hosting 4 partitions (P0-P3) with 2 consumer groups. Consumer group A has 2 consumer instances and group B has 4. [23]

Kafka can exhibit different properties from the CAP theorem depending on the configurations chosen for the producer and the topic. Two important configuration settings

related to these properties are `min.insync.replicas`, which specifies the minimum number of replicas that must acknowledge a write for it to be considered successful, and `replication-factor`, which defines how many replicas (including the leader) must exist.

3.2.2 Kafka Connect

Kafka Connect [24] is a framework for Apache Kafka that supports several CDC tools. This framework links data sinks and sources to Kafka. To understand how Kafka Connect operates, it is important to explain the following terms: connectors, connector classes, tasks, transformations, and converters.

- Connectors are high-level abstractions that define where data is copied to and from, using tasks to manage the process. They are categorized into two types: source connectors and sink connectors. Source connectors generate messages by reading changes from a data source and streaming them into Kafka topics, while sink connectors retrieve messages from Kafka topics and deliver them to data sinks.
- Connector Classes define the logic for interacting with the source or sink systems.
- Tasks are the actual implementations of how data is copied to and from Kafka. Each connector instance coordinates a set of tasks that are executed by the Kafka Connect cluster.
- Transformations are simple and lightweight modifications that can be applied to each message before it enters or after it exits Kafka.
- Converters are responsible for serializing and deserializing the messages that are written to and read from Kafka.

When capturing data from a source, a connector generates messages by reading the changes in the source. The messages are then processed by zero or more transformations and subsequently serialized by a converter before being sent to Kafka. When pulling messages from a Kafka topic, the messages are first deserialized by a converter, then processed by zero or more transformations, and finally read by the connector instance, which applies the changes to the sink. Figure 3.4 illustrates the flow of data from a data source to a Kafka cluster and then from a Kafka cluster to a data sink.

All operations on a Kafka Connect cluster—such as adding, removing, or viewing connectors, tasks, and connector types—are performed through its REST API. Listing 3.1

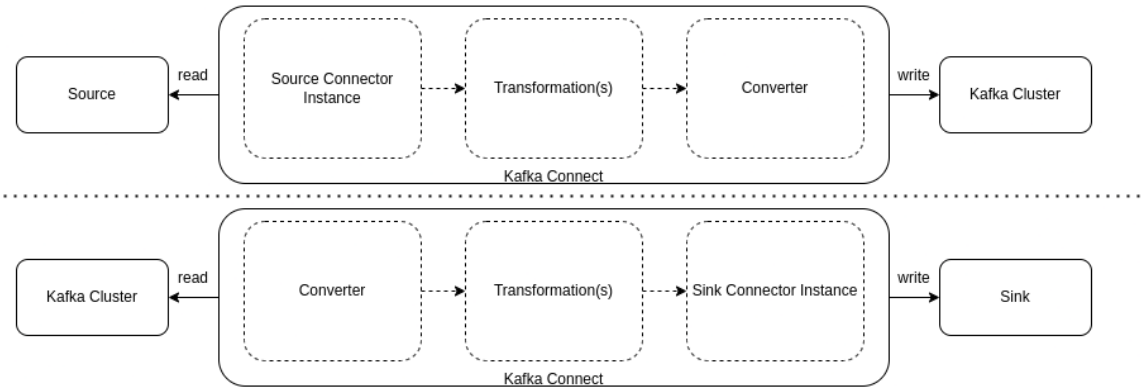

FIGURE 3.4: Diagram demonstrating Kafka Connect inner workings [25].

shows the JSON body of a POST request used to create a Kafka Source Connector called ”inventory-connector.” The ”config” key contains several configurable values. Some, like `connector.class` (which specifies the connector class being used) and `tasks.max` (which defines the maximum number of parallel tasks the connector can run), are common across different connectors, while others depend on the specific connector type.

```
{
    "name": "inventory-connector",
    "config": {
        "connector.class": "io.debezium.connector.mysql.MySqlConnector",
        "tasks.max": "1",
        "database.hostname": "mysql",
        "database.port": "3306",
        "database.user": "debezium",
        "database.password": "dbz",
        "database.server.id": "184054",
        "topic.prefix": "dbserver1",
        "database.include.list": "inventory",
        "schema.history.internal.kafka.bootstrap.servers": "kafka:9092",
        "schema.history.internal.kafka.topic": "schema-changes.inventory"
    }
}
```

LISTING 3.1: Creation of a Connector Instance

3.2.3 Debezium

Debezium[26] is an open-source distributed platform for CDC. It was selected as our tool for implementing pull-based CDC in our system. This pull log-based CDC system leverages Apache Kafka for its delivery process. Debezium, built on top of Kafka Connect,

provides a set of Kafka Connect-compatible connectors. The connectors track data changes in the DBMS by detecting them as they occur and stream a record of each change event to a Kafka topic. Consuming applications can then read the event records from the Kafka topic. Figure 3.5 illustrates the architecture of a CDC pipeline based on Debezium.

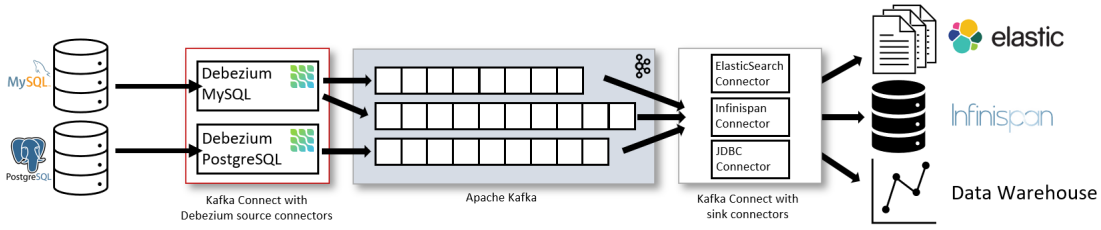

FIGURE 3.5: Debezium Architecture. Source: [26].

3.2.4 Confluent Schema Registry

Kafka Connect, which includes Debezium as a connector, supports various data serialization systems, including JSON, Apache Avro, Protobuf, and String Delimited. By default, the Debezium connector uses JSON serialization, where the schema is included alongside the payload, as shown in Listing 3.2. However, embedding the schema in every message can significantly increase message size, leading to inefficiencies, particularly in high-throughput systems. To mitigate this issue, we used a schema registry.

A schema registry serves as a centralized repository for managing and validating schemas for topic message data. It facilitates the serialization and deserialization of data over the network by storing and serving schemas to producers and consumers. This centralized approach allows for schema reuse across messages, reducing message size and ensuring consistency in data structure.

In practice, the schema registry assigns each schema a unique identifier (ID). When a message is serialized, only the schema ID is included in the message, instead of the entire schema. The consumer then retrieves the full schema from the schema registry using this ID, enabling correct deserialization of the data. This approach not only reduces message size but also simplifies schema evolution, as schema changes can be managed centrally without requiring changes to application logic.

Debezium is compatible with two schema registry systems: Apicurio API [27] and Confluent Schema Registry [28]. For this work, we selected the Confluent Schema Registry due to its simplicity and its status as the industry standard. The Confluent Schema Registry

offers robust features, such as versioning, schema compatibility checks, and integration with Kafka.

```
{
  "schema": {
    "type": "struct",
    "fields": [
      { "type": "int32", "optional": false, "field": "id" },
      { "type": "string", "optional": true, "field": "name" },
      { "type": "string", "optional": true, "field": "email" }
    ],
    "optional": false,
    "name": "dbserver1.inventory.customers.Value"
  },
  "payload": {
    "id": 1001,
    "name": "John Doe",
    "email": "john.doe@example.com"
  }
}
```

LISTING 3.2: Debezium payload using JSON serialization

3.3 Architecture

Now that we have selected our tools, we can delve into the details of how our system architecture operates.

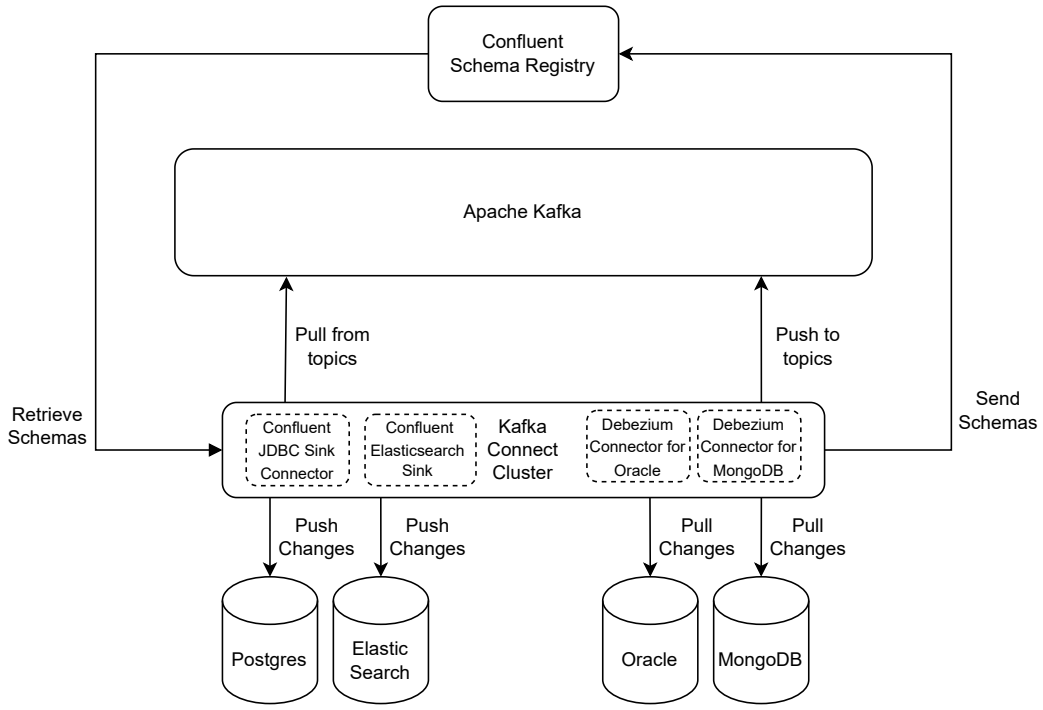

FIGURE 3.6: CDC Architecture

Figure 3.6 illustrates a more complete architecture with our chosen technologies included. For the sake of clarity, PostgreSQL and Elasticsearch are used as examples of data sinks, while Oracle and MongoDB are used as examples of data sources.

In this architecture, our Kafka Connect cluster runs Debezium connectors for Oracle and MongoDB to capture data changes. These changes are then pushed into several Kafka topics, typically one topic per table captured. The Confluent Schema Registry stores the schemas and serialization types associated with these data changes.

The JDBC and Elasticsearch sink connectors continuously poll Kafka to retrieve new data arriving in the Kafka topics. When new data arrives, it is deserialized, and the changes are applied to the PostgreSQL and Elasticsearch databases.

3.3.1 System Management Architecture

Since our system utilizes several virtual machines and technologies, maintaining its normal functioning can be challenging. Error tracking becomes difficult, as pinpointing the exact source of an error—whether it is from a specific technology or a misconfiguration in the virtual machines and resource allocations—can be complex. To address this issue, we implemented logging and monitoring using Grafana Loki and Prometheus.

Prometheus [29] Is an open-source systems monitoring and alerting toolkit. It is widely used for collecting and storing metrics as time series data. A metric is a quantifiable data point, and Prometheus gathers these by scraping exporters via HTTP. Exporters are agents that collect specific metrics, format them, and expose them through HTTP endpoints for Prometheus to scrape. When paired with Node Exporter [30], it can expose system-related metrics. In particular we were interested in collecting metrics such as CPU usage, RAM usage, swap memory usage, and the current state of system processes related to Kafka, Kafka Connect and the Confluent Schema Registry.

Grafana Loki [31] Is a log aggregation system that is horizontally scalable and easy to implement. The Loki logging stack consists of three main components: Promtail, an agent responsible for gathering logs and sending them to a Loki instance; Loki, the main server that stores logs and processes queries; and Grafana, which is used to query and display the logs. Loki was used in our system to aggregate the log files related to Kafka and Kafka Connect.

Grafana [32] Is a open-source platform for monitoring and observability. It provides a wide range of visualization tools for displaying metrics and logs, making it ideal for creating real-time dashboards. In our system, Grafana is used to display the metrics collected by Prometheus and visualize log data processed by Loki.

For our monitoring and logging process every machine instance ran both a Promtail and a Node Exporter process. The Promtail process tailed the log files related to the Kafka and Kafka Connect instances, while the Node Exporter process collected system metrics, such as CPU usage, RAM usage, swap memory usage, and the current state of processes related to Kafka, Kafka Connect, and the Schema Registry. A single machine instance hosted the Loki, Prometheus, and Grafana services. All Promtail processes sent log data to the Loki process, and the Prometheus instance pulled metrics from all the Node Exporter processes. Grafana was then used to display statistics based on the gathered information. In Section 3.4.2, we discuss the dashboards in more detail. Figure 3.7 illustrates how the monitoring and logging architecture operates.

To ensure that the system was easy to replicate and configure, we utilized Ansible, an open-source automation tool that simplifies system management, configuration, application deployment, and task automation. Ansible operates without the need for special

agents on target systems, instead relying on standard protocols like SSH for communication.

One of the reasons for choosing Ansible was the availability of community-built collections and roles, which saved time by eliminating the need to develop these resources from scratch. Collections are packages of reusable content, such as playbooks, modules, roles, and plugins, that enable users to efficiently share, organize, and distribute automation tasks. Roles, on the other hand, are a structured way to group related tasks, variables, and handlers, which simplifies complex automation processes and makes them easier to maintain and reuse across multiple projects or systems. Both collections and roles played a crucial role in deploying and configuring our system.

All the setup and configuration for this system were automated using Ansible, and the details are specified in Section 3.4.

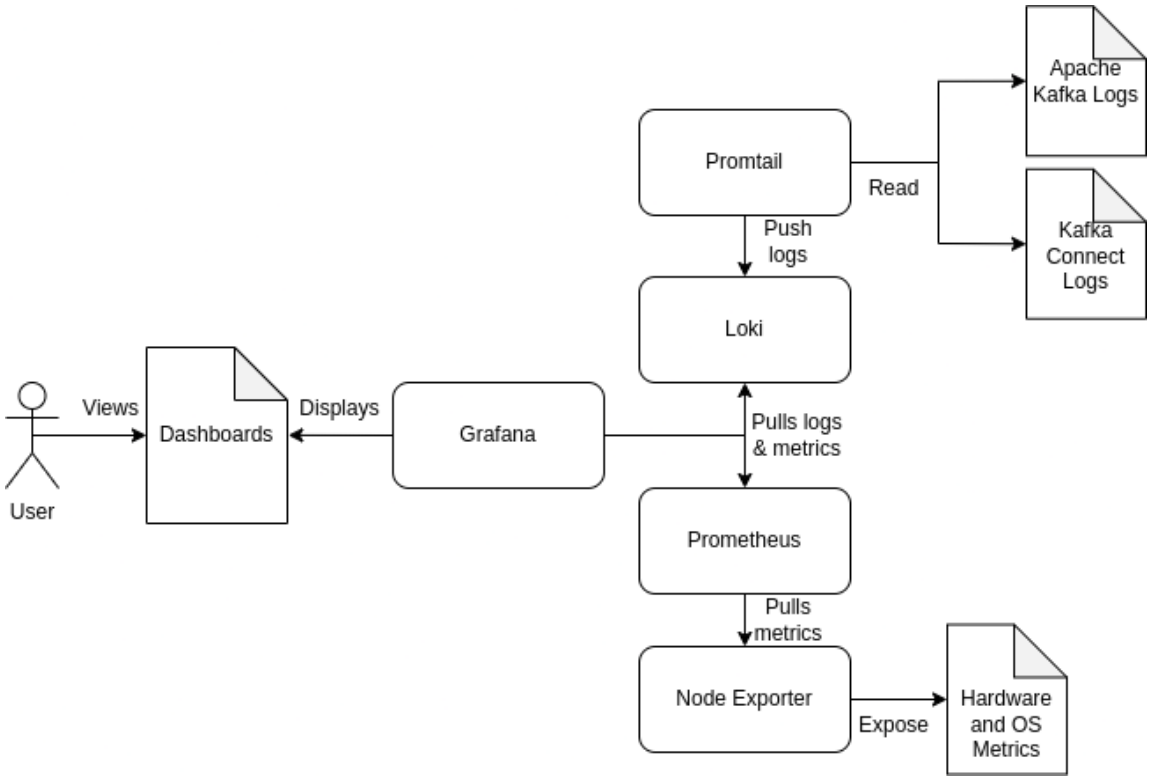

FIGURE 3.7: Monitoring and Logging Architecture

3.4 Setup

In this section, we discuss key configurations made to the technologies used. In Section 3.4.1, we cover our infrastructure and how we configured it using Ansible. Section 3.4.2 explains how we set up logging and monitoring with Grafana Loki, Prometheus, and

Grafana, along with the types of information displayed by this setup. In Section 3.4.3, we discuss the types of serialization offered by the Confluent Schema Registry, select a serializer, and explain our choice. Section 3.4.4 details the connectors we loaded into Kafka Connect, why we selected them, and how they operate.

3.4.1 Machines and Processes

To make the setup of this system repeatable and easily configurable, we utilized the `confluent.platform` [33], an Ansible collection maintained by Confluent [28]. This collection was chosen because it provides playbooks to deploy, manage, and configure all of our core technologies: Apache Kafka, Kafka Connect, and the Confluent Schema Registry. To automate the logging and monitoring configuration, we also used the following collections and roles: `prometheus.prometheus` [34], `grafana.grafana` [35], `voidquark.loki` [36], and `voidquark.promtail` [37].

The setup prioritized a minimum viable product while maintaining system reliability and scalability. We deployed six virtual machines, each equipped with 2GB of RAM, 50GB of storage, and 2 vCPU cores. Individual domains were assigned to these machines to support role separation:

- **Kafka Cluster**: Three machines hosted Apache Kafka. This configuration met the Raft protocol requirement of at least $2N + 1$ machines to ensure fault tolerance, allowing the cluster to withstand N machine failures. These machines were:
  - `debez-kafka-1.dev.uporto.pt`
  - `debez-kafka-2.dev.uporto.pt`
  - `debez-kafka-3.dev.uporto.pt`

  Each Kafka machine hosted the following processes: Kafka Broker, Kafka Controller, Promtail, and Node Exporter.

- **Kafka Connect**: Two machines were allocated for Kafka Connect. Testing revealed that a single machine could not handle the load generated by both source and sink connectors, prompting the addition of a second machine. These machines were:
  - `debez-connect-1.dev.uporto.pt`
  - `debez-connect-2.dev.uporto.pt`

Each Kafka Connect machine hosted the following processes: Kafka Connect, Promtail, and Node Exporter.

- **Metrics and Monitoring**: One machine, `debez-metrics.dev.uporto.pt`, hosted the Confluent Schema Registry along with Grafana, Promtail, and Loki. Because the Schema Registry has low resource demands, it was co-located with the monitoring and logging stack without performance issues. While this setup met current needs, increased system loads in the future may require distributing these processes across additional machines.

Figure 3.8 shows a diagram illustrating all the virtual machines and the processes they host.

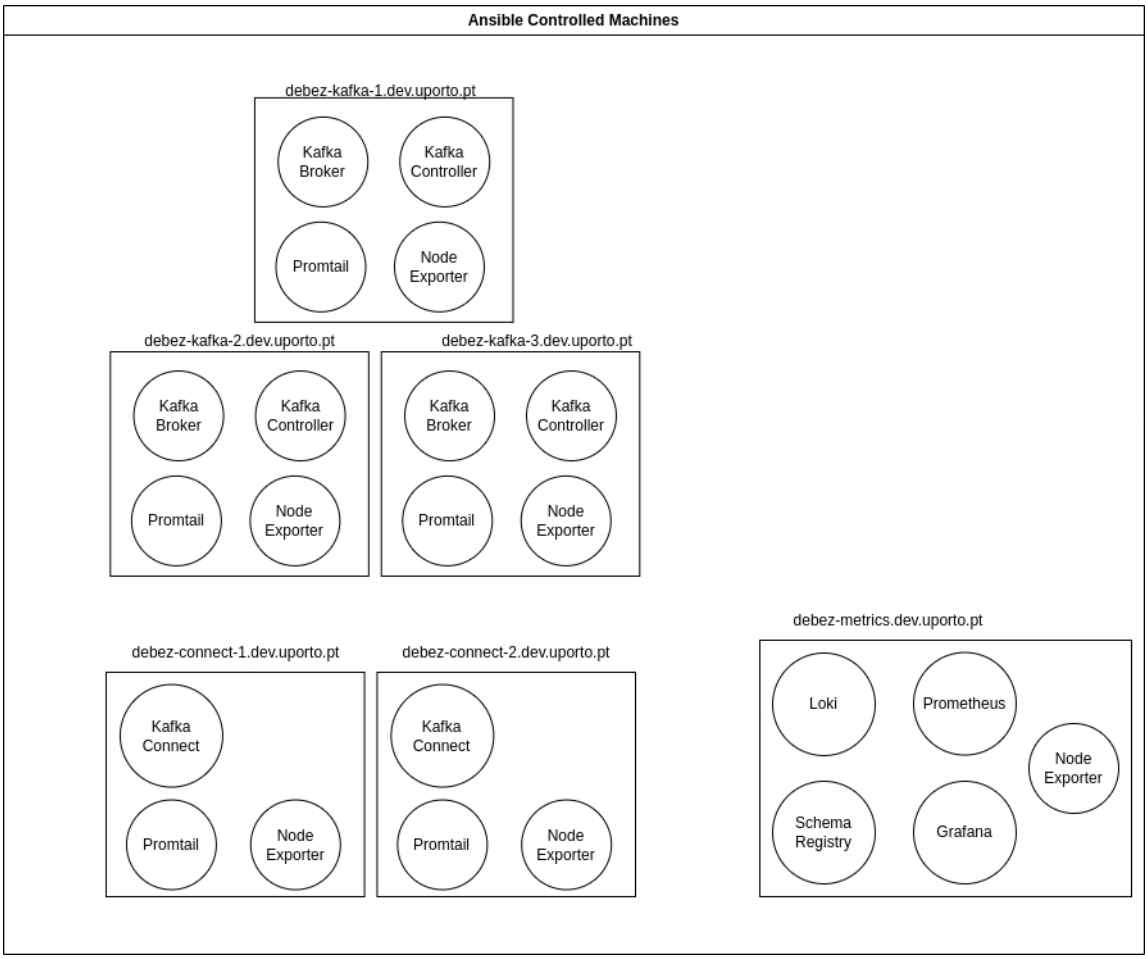

FIGURE 3.8: Servers and processes

3.4.2 Logging and Monitoring

To configure monitoring, we used the `prometheus.prometheus` collection. This allowed us to set up a Node exporter process on all machines and configure a Prometheus process to pull data from all Node exporter processes.

For logging, we used the `voidquark.loki` and `voidquark.promtail` roles to configure Grafana Loki and Promtail, respectively. A Loki instance ran on the monitoring server, listening at a specific URL, while Promtail processes were deployed on all Kafka and Kafka Connect instances. The logs collected included the server.log from both Kafka Broker and Kafka Controller processes, as well as the connect.log from the Kafka Connect process. These processes were configured to use the log4j logging framework, making it easier to filter logs based on their level.

We used Grafana to view the logs aggregated by Loki and the metrics collected by Prometheus. To deploy and configure Grafana, we used the `grafana.grafana` Ansible collection. We utilized two dashboards: one provided by Grafana Labs to monitor the resource usage of our instances, and another custom dashboard to view the specified logs and the status of Kafka, Kafka Connect, and Confluent Schema Registry processes.

Figure 3.9 shows the dashboard used to monitor the logs and process states. The ”Job” variable allows us to select the instance type (Kafka Broker, Kafka Controller, or Kafka Connect), displaying only logs related to those processes. We can further narrow the logs to a specific instance using the ”Instance” variable. Additionally, the ”Level” variable enables filtering based on log level, and the ”Search” variable allows searching within the displayed log files.

Figure 3.10 shows the dashboard used to monitor the resource usage of our instances. To select the instance we are interested in we change the value of the ”Host” variable.

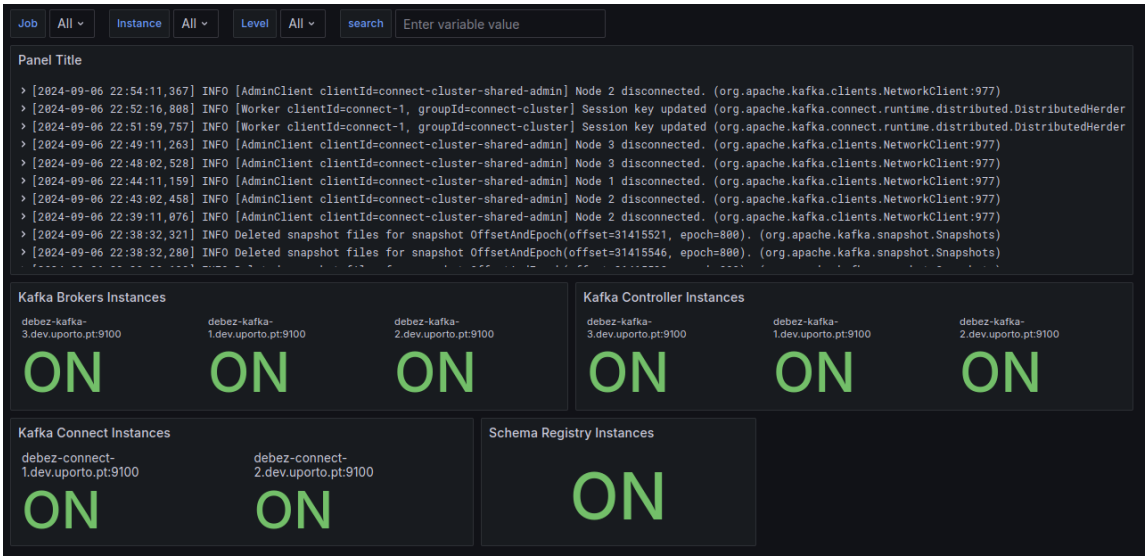

FIGURE 3.9: Grafana Dashboard to view logs and processes state

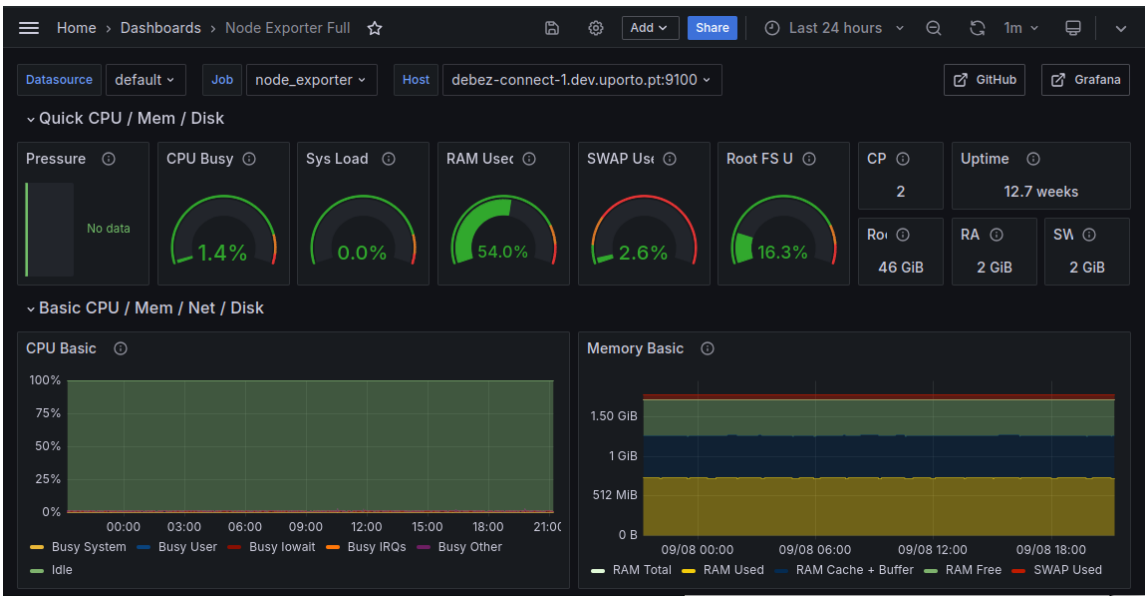

FIGURE 3.10: Grafana Dashboard to view the resource usage of an instance

3.4.3 Serialization

The choice of serialization format in our CDC system directly affects schema readability and the space efficiency of the messages stored in Kafka topics. The Confluent Schema Registry supports multiple serialization formats, including Avro, JSON Schema, and Protobuf. Each format has its strengths: Avro is efficient and well-suited for scenarios requiring compact binary serialization and schema evolution. Unlike JSON Schema, which keeps data in a human-readable format, Avro produces binary-encoded data optimized

for performance and storage. Avro schemas are defined using JSON, making them easy to manage, even though the data itself is in a binary format. Protobuf also uses a binary format and excels in cross-platform communication, providing a similar balance of efficiency and schema management. For this work, we chose Avro serialization due to its widespread use, efficiency in handling large volumes of data, and robust schema management [38].

Listing 3.2 shows the JSON payload for a Customer record. In Listings 3.3 and 3.4, we present the binary payload of the same message when serialized with Apache Avro and the corresponding JSON schema (as stored in the schema registry).

```
0x02 0x6A 0x6F 0x68 0x6E 0x20 0x64 0x6F 0x65 0x10 0x6A 0x6F 0x68 0x6E 0x2E 0x64 0x6F
     0x65 0x40
```

LISTING 3.3: Example of Avro serialized binary message (represented in hexadecimal)

```
{
  "type": "record",
  "name": "Customer",
  "namespace": "dbserver1.inventory",
  "fields": [
    { "name": "id", "type": "int" },
    { "name": "name", "type": ["null", "string"], "default": null },
    { "name": "email", "type": ["null", "string"], "default": null }
  ]
}
```

LISTING 3.4: Debezium payload schema in Avro (defined using JSON)

3.4.4 Kafka Connect

To use Kafka Connect, you need to load it with connectors. We selected connectors for data sources and sinks related to company projects experiencing latency issues, though we integrated our system with only one project. The data sources for these projects were Oracle and PostgreSQL. Thus, we loaded Kafka Connect with the following source connectors:

Debezium Source Connector for Oracle This connector captures row-level changes in Oracle Server databases. It offers three options for change identification: using OpenLogReplicator, an external log scanner; Oracle's LogMiner, the default option, a DBMS-integrated log scanner; or the XStream API, another DBMS-integrated log scanner. Changes are then propagated by the Kafka Connector into Kafka Topics.

Debezium Source Connector for PostgreSQL This connector captures row-level changes in PostgreSQL database schemas. It uses PostgreSQL's logical decoding for change identification, generating changes that Kafka Connect later collects using pgoutput, the standard logical decoding output plug-in in PostgreSQL.

The possible targets from the projects were PostgreSQL and Elasticsearch databases. Therefore, we loaded the following sink connectors:

Confluent Elasticsearch Service Sink Connector This connector writes data from multiple Kafka topics to indexes in Elasticsearch. It periodically polls Kafka topics for messages and writes those messages to indexes using the Elasticsearch REST API.

Confluent JDBC Sink Connector This connector consumes messages from multiple topics and writes those events to a relational database. It periodically polls Kafka topics and writes changes to any relational database using the JDBC (Java Database Connectivity) API, provided you supply a compatible JDBC driver, such as pgJDBC a JDBC driver for PostgreSQL.

With this setup complete, our system is now ready for integration with a project. In the next chapter, we will discuss how this system was integrated into one of the projects that experienced latency issues.

Chapter 4

Empirical Study

To assess the practicality and limitations of our CDC system, we integrated it with an existing project that experienced latency issues due to shared data. This chapter outlines that integration, starting with Section 4.1, which details the system and its challenges. In Section 4.2, we discuss our solutions. Finally, in Section 4.3, we evaluate the approach, its limitations, and the constraints of our CDC system.

4.1 Target Application

This project involves a Django application that manages course-related information stored in the company's primary Oracle database. Currently, services access this database through REST API calls. Several APIs expose different parts of the database, but our focus is on the Courses API, which provides course-related data. The Courses API is a Django service responsible for handling API requests, retrieving information from the database, and serializing it into JSON format. Upon receiving an API call, it follows these steps:

1. Fetch relevant data from the primary database by executing one or more PL/SQL procedures.
2. Serialize the retrieved data into JSON format. The serialization is split between code written directly in the API endpoint and a Django Serializer module.
3. Return the serialized JSON response

A key component of this project for our integration is a Python script executed daily via a CronJob. This script currently performs the following tasks:

1. Retrieve course-related data from the primary database using multiple REST API calls.

2. Apply a series of transformations to the fetched data to meet the service's requirements.

3. Store the transformed data in a PostgreSQL database using the Django Object-Relational Mapping (ORM).

Figure 4.1 provides an overview of how the project currently fetches course information from the primary database.

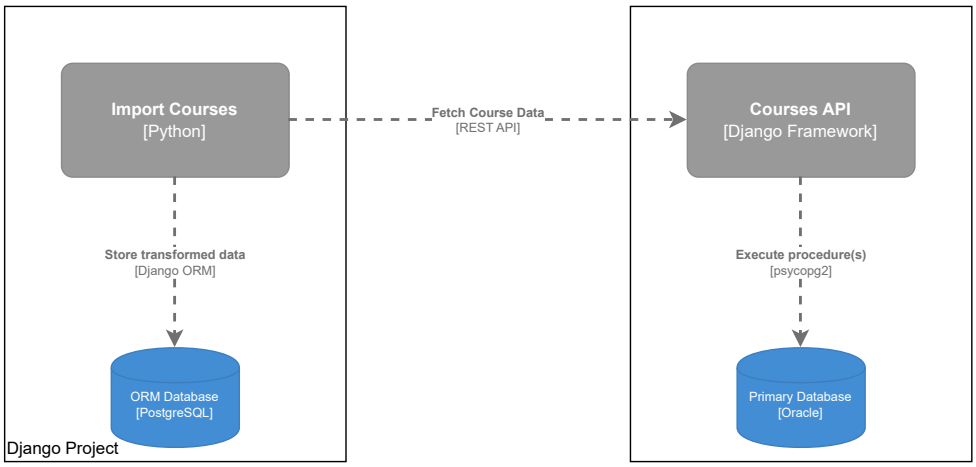

FIGURE 4.1: Current setup for fetching and processing course information.

The problem arises from the time required to complete the first step of the service. Currently, the script retrieves information for 1,739 courses. There are six individual endpoints (API Calls A, B, C, D, E, F in fig 4.2), with one endpoint potentially being called zero or more times per course. The script calls each of these APIs to request a part of the information needed. In the best-case scenario, each course requires five API calls. This results in a minimum of 8,695 API calls per day, which takes over four hours to complete. Figure 4.2 illustrates this process.

These six API endpoints can trigger up to 19 individual PL/SQL procedures that access a total of 27 tables. The complexity of these procedures varies: some are simple with no dependencies, while others are more complex, involving multiple calls to other procedures.

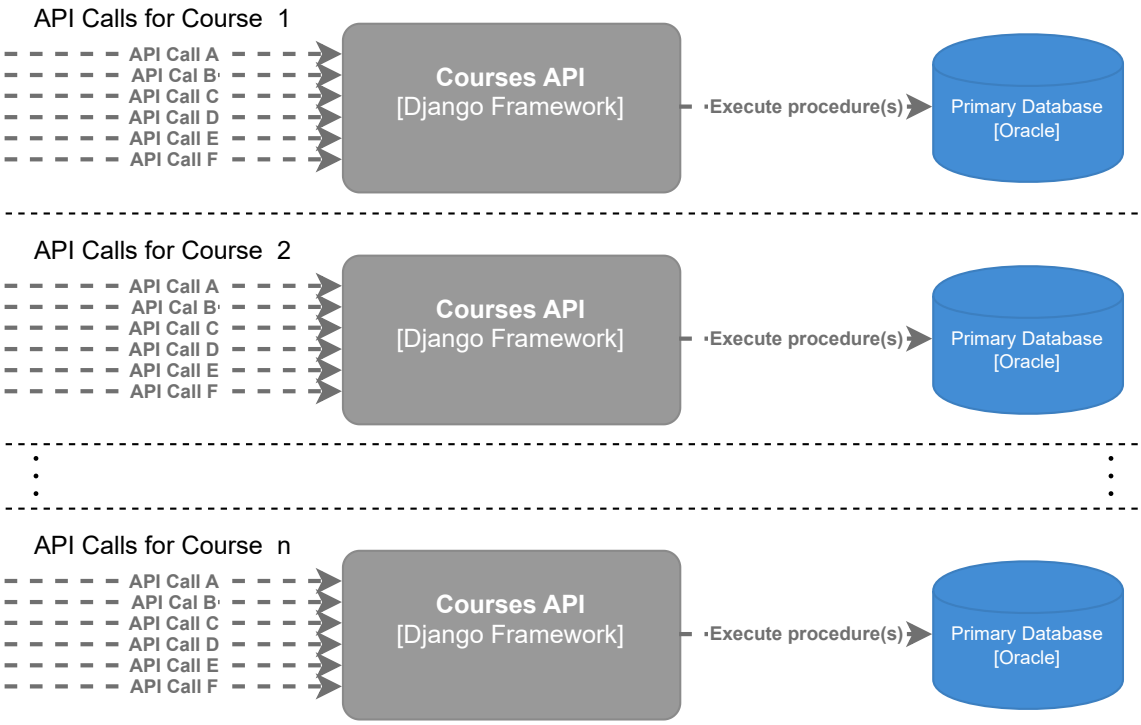

FIGURE 4.2: Retrieving course information, highlighting the multiple API calls required per course

4.2 Migration to CDC

The data stored in PostgreSQL is also present in the Oracle database, but in a completely different structure. This means that merely replicating the relevant Oracle database data using our CDC system is insufficient. We must either refactor all service modules that use the database to accommodate the original format or convert the data to fit the expected format. Refactoring all modules would contradict the principles of microservices, so we chose the latter approach. Our chosen approach was as follows:

1. Capture changes from the primary database on the tables used by PL/SQL procedures related to courses. This is achieved using our CDC system, specifically the Debezium Connector for Oracle.

2. Replicate these tables in the PostgreSQL database using our CDC system, specifically the Confluent JDBC Sink Connector.

3. Create PostgreSQL queries and views that replicate the functionality of the PL/SQL procedures, but adapted for bulk operations. Instead of calling a set of procedures 1,739 times via an individual API endpoints, a single call to the equivalent set of queries and views retrieves the same information.

4. Develop a Python module, named Debez, to fetch and serialize the queries and views into a similar JSON format. This module only needs to execute six functions (designated as Python Call A*, B*, C*, D*, E*, F*) to obtain the same information that was previously gathered through 8,695 API calls. Figure 4.3 illustrates this new method of retrieving the information.

FIGURE 4.3: New way of retrieving course information

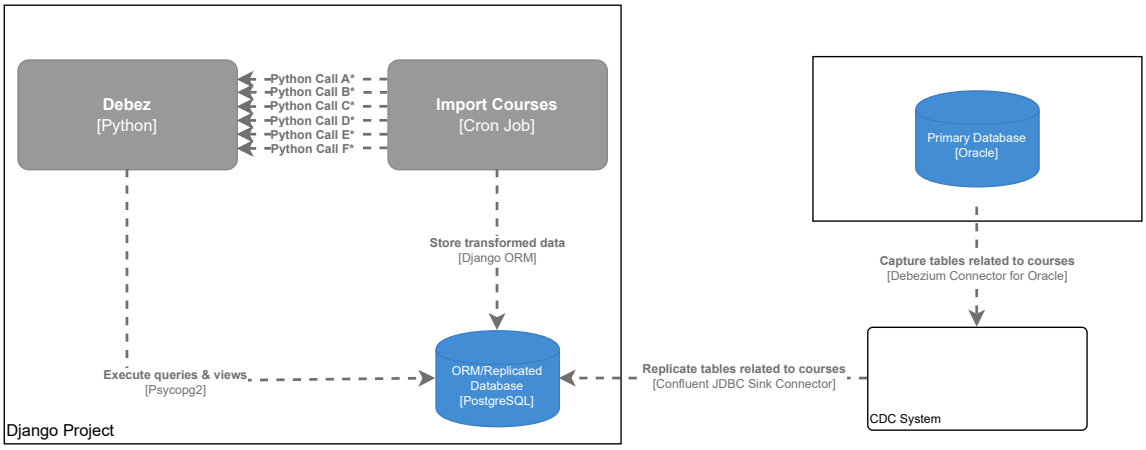

FIGURE 4.4: New architecture

With this approach the data is replicated in a PostgreSQL database along with the ORM data of the app. To access it the Import Courses module does 6 python calls to the Debez Module. After receiving a call the Debez module accesses data in the database via a set of queries and views and serializes it before returning the call to the Import Courses module. The Import Courses module then does the transformations it originally did and stores the information in the same PostgreSQL database using Django ORM. Figure 4.4 illustrates this process.

As we saw, several changes were made to the project. First, we replicated the tables related to the PL/SQL procedures by creating equivalent PostgreSQL queries and views tailored for bulk Python calls. This involved replicating 26 tables, creating 4 views, and designing 22 queries on the target database. Second, we developed the Debez module to handle the six Python calls and execute the necessary queries and views. This module was also responsible for serializing the queries and views into the JSON format expected by the

Python calls. Part of the serialization logic was coded directly into the module, while the rest utilized the Django Serializer module. Finally, we adapted the Import Courses module to perform only 6 Python calls, significantly reducing the number of individual REST API calls.

4.3 Findings

To evaluate the performance of the new method, we conducted multiple tests to measure how quickly data could be fetched and processed. Specifically, we fetched course data 10 times, measuring the time from the initiation of bulk calls to the completion of data serialization. It's important to note that the tables were already synchronized, ensuring that the data was consistent with the primary Oracle database. The data fetched was based on a list of active courses, so sequential calls returned the same data, assuming no changes were made to the source database.

The results indicated that the new method averaged around 5.977 seconds for data retrieval, with the longest time recorded being 6.23 seconds and the shortest 5.44 seconds.

In contrast, the previous method of accessing data, which relied on REST APIs and procedures, took over 4 hours to complete the synchronization process. This method's inefficiency was due to its fragmented approach, where the database components were treated as distinct resources and each HTTP request added significant overhead. The old method also involved thousands of HTTP protocol executions and numerous network round trips, further exacerbating latency.

Although the new method proved to be much more efficient, achieving a substantial reduction in synchronization time, it introduced additional complexity. Maintaining the Debez module, which was integral to the new approach, required careful management since changes in the Oracle database could impact the queries and views used. Despite these complexities, the system demonstrated the capability of transmitting thousands of messages per second.

Chapter 5

Conclusion and Future Work

This thesis explored the use of CDC for data propagation across microservices in a hybrid system that integrates both monolithic and microservices architectures. The primary objectives were to design and develop a CDC pilot project, assess its integration into an existing system, identify challenges and limitations, and evaluate its potential for broader use within the organization.

The implementation of the Change Data Capture (CDC) system using Debezium and Kafka significantly reduced data propagation latency. This system offers an alternative to the existing REST APIs in certain scenarios, with demonstrated improvements in performance. It utilizes a publish/subscribe model to efficiently communicate changes to multiple systems simultaneously. Leveraging Kafka and Kafka Connect, the system benefits from Kafka's partitioned architecture for reliability and the scalable nature of its clustered setup.

It was noted that services that do not depend on the existing REST API integrate more seamlessly with this system. The synchronous nature of APIs and differences in data formats across services can complicate this integration. Also this project exclusively utilized Apache Kafka, Debezium, and Kafka Connect, without exploring alternative tools that could offer additional benefits. Moreover, the proposed architecture is less suited for complex relational databases, as Debezium captures changes only at the table level. This limitation reduces its effectiveness when relationships between multiple tables are critical to the database design.

In such cases, using a message broker with custom messages that include relationships, or employing tools like ksqlDB or Kafka Streams, may be more effective. Both tools are built

on Apache Kafka and allow foreign key joins using Kafka topics, addressing Debezium's limitations in handling complex relational data.

Future work should focus on enhancing integration methods for existing services to manage complexity effectively. It is crucial to develop strategies for integrating both existing services with the CDC approach and for designing best practices for new services that may arise. Additionally, evaluating the types of services that best align with our system—whether it involves communication between existing services, new services, or a combination of both—is important. Further studies are needed to analyze system throughput under load, particularly when Kafka handles multiple services and Kafka Connect writes to replicating tables. This analysis will not only improve understanding of the system's capabilities but also guide decisions on when to add new nodes to Kafka and Kafka Connect to scale efficiently.

In conclusion, the system effectively reduced latency while demonstrating both reliability and scalability. Although integration may not be advantageous for all services—particularly those dependent on existing REST APIs that do not require low latency—the system shows promise for new services and scenarios where low latency is critical. Future work should focus on a more thorough evaluation of these scenarios to identify and optimize the conditions that maximize the system's benefits.

Bibliography

[1] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, 2019, pp. 1–7. [Cited on page 1.]

[2] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs: Prentice Hall, 1996, vol. 1. [Cited on page 5.]

[3] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022. [Cited on pages 6 and 10.]

[4] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *CoRR*, vol. abs/1606.04036, 2016. [Online]. Available: http://arxiv.org/abs/1606.04036 [Cited on pages 6 and 8.]

[5] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data management in microservices: State of the practice, challenges, and research directions," 2021. [Online]. Available: https://arxiv.org/abs/2103.00170 [Cited on pages ix, 7, and 8.]

[6] N. Bjørndal, A. Bucchiarone, M. Mazzara, N. Dragoni, S. Dustdar, F. B. Kessler, and T. Wien, "Migration from monolith to microservices: Benchmarking a case study," *Tech. Rep.*, 2020. [Cited on page 10.]

[7] J.-P. Gouigoux and D. Tamzalit, "From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture," in *2017 IEEE international conference on software architecture workshops (ICSAW)*. IEEE, 2017, pp. 62–65. [Cited on page 10.]

[8] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in *Current Trends in Web Engineering: ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers 17*. Springer, 2018, pp. 32–47. [Cited on page 10.]

[9] I. K. Aksakalli, T. Çelik, A. B. Can, and B. Tekinerdoğan, "Deployment and communication patterns in microservice architectures: A systematic literature review," *Journal of Systems and Software*, vol. 180, p. 111014, 2021. [Cited on pages 10 and 11.]

[10] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018, ch. Interprocess communication in a microservice architecture. [Cited on page 10.]

[11] E. Brewer, "Cap twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012. [Cited on page 12.]

[12] S. Burckhardt, A. Gotsman, and H. Yang, "Understanding eventual consistency," *Microsoft Research, Redmond, WA, USA, Tech. Rep. MSR-TR-2013–39*, 2013. [Cited on page 12.]

[13] S. Burckhardt *et al.*, "Principles of eventual consistency," *Foundations and Trends® in Programming Languages*, vol. 1, no. 1-2, pp. 1–150, 2014. [Cited on page 12.]

[14] W. Golab, "Proving pacelc," *ACM SIGACT News*, vol. 49, no. 1, pp. 73–81, 2018. [Cited on page 13.]

[15] K. Xiang, "Patterns for distributed transactions within a microservices architecture," *Red Hat Developer Blog*, 2018, accessed: 2024-08-02. [Online]. Available: https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture [Cited on pages 13 and 14.]

[16] B. Stopford, *Designing event-driven systems*. O'Reilly Media, Incorporated, 2018. [Cited on page 13.]

[17] H. Garcia-Molina and K. Salem, "Sagas," *SIGMOD Rec.*, vol. 16, no. 3, p. 249–259, dec 1987. [Online]. Available: https://doi.org/10.1145/38714.38742 [Cited on page 14.]

[18] L. Hao, T. Jiang, Y. Lin, and Y. Lu, "Methods for solving the change data capture problem," in *Advances in Natural Computation, Fuzzy Systems and Knowledge Discovery*, N. Xiong, M. Li, K. Li, Z. Xiao, L. Liao, and L. Wang, Eds. Cham: Springer International Publishing, 2023, pp. 781–788. [Cited on pages 14, 17, and 19.]

[19] Denny, I. P. M. Atmaja, A. Saptawijaya, and S. Aminah, "Implementation of change data capture in etl process for data warehouse using hdfs and apache spark," in *2017 International Workshop on Big Data and Information Security (IWBIS)*, 2017, pp. 49–55. [Cited on page 14.]

[20] D. M. Tank, A. Ganatra, Y. Kosta, and C. Bhensdadia, "Speeding etl processing in data warehouses using high-performance joins for changed data capture (cdc)," in *2010 International Conference on Advances in Recent Technologies in Communication and Computing*, 2010, pp. 365–368. [Cited on page 14.]

[21] M. Eccles, "Pragmatic development of service based real-time change data capture," Ph.D. dissertation, Aston University, 2013. [Cited on pages 15, 19, and 20.]

[22] Q. Hu, Z. Gan, and B. Zhang, "Design and implementation of oracle database incremental data capture based on trigger and identification table," in *Journal of Physics: Conference Series*, vol. 1237, no. 2. IOP Publishing, 2019, p. 022161. [Cited on pages ix, 17, and 18.]

[23] Apache Software Foundation, "Apache kafka documentation," https://kafka.apache.org/documentation/, 2024, accessed: 2024-08-31. [Cited on pages ix, 25, and 26.]

[24] Confluent, "Kafka connect," https://docs.confluent.io/platform/current/connect/index.html, 2024, accessed: 2024-08-31. [Cited on page 27.]

[25] I. Confluent, "Connectors, configuration, converters, and transforms," https://developer.confluent.io/courses/kafka-connect/how-connectors-work/, 2024, accessed: 2024-09-03. [Cited on pages ix and 28.]

[26] Debezium, "Debezium: Open source distributed change data capture," accessed: 2024-08-08. [Online]. Available: https://debezium.io/ [Cited on pages ix, 28, and 29.]

[27] Apicurio, "Apicurio registry," https://www.apicur.io/registry/, accessed: 2024-08-08. [Cited on page 29.]

[28] Confluent, "Confluent documentation," https://docs.confluent.io/, accessed: 2024-08-08. [Cited on pages 29 and 34.]

[29] Prometheus, *Prometheus Documentation*, https://prometheus.io/docs/, 2024, accessed: 2024-09-05. [Cited on page 32.]

[30] ——, "Node exporter," 2024, accessed: 2024-09-05. [Online]. Available: https://github.com/prometheus/node_exporter [Cited on page 32.]

[31] Grafana Labs, *Loki Documentation*, 2024, accessed: 2024-09-05. [Online]. Available: https://grafana.com/docs/loki/latest/ [Cited on page 32.]

[32] ——, *Grafana Documentation*, 2024, accessed: 2024-09-05. [Online]. Available: https://grafana.com/docs/grafana/latest/ [Cited on page 32.]

[33] C. Inc., "cp-ansible: Confluent platform ansible playbooks," 2024, accessed: 2024-08-29. [Online]. Available: https://github.com/confluentinc/cp-ansible [Cited on page 34.]

[34] P. Community, "Ansible collection for prometheus," 2024, accessed: 2024-08-29. [Online]. Available: https://github.com/prometheus-community/ansible [Cited on page 34.]

[35] G. Labs, "Ansible collection for grafana," 2024, accessed: 2024-08-29. [Online]. Available: https://github.com/grafana/grafana-ansible-collection [Cited on page 34.]

[36] voidquark, "Ansible role for loki," 2024, accessed: 2024-08-29. [Online]. Available: https://github.com/voidquark/loki [Cited on page 34.]

[37] ——, "Ansible role for promtail," 2024, accessed: 2024-08-29. [Online]. Available: https://github.com/voidquark/promtail [Cited on page 34.]

[38] Apache Software Foundation, "Apache avro documentation," accessed: 2024-08-08. [Online]. Available: https://avro.apache.org/docs/ [Cited on page 38.]