

A CDC SOLUTION

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2024

Peter Hamer

Supervised by Dr. G. Henshall

Department of Computer Science

Contents

Abstract	4
Declaration	5
Copyright	6
Acknowledgements	7
1 Introduction	8
1.1 Motivation	9
1.2 Aims and Objectives	10
1.3 Report Overview	10
2 Background	12
2.1 Change Events	12
2.1.1 Flavours of CDC Systems	12
2.1.2 Common Patterns	13
2.1.3 Replication	14
2.2 Existing Solutions	15
2.2.1 Debezium	15
2.2.2 Google Datastream	20
3 Design	22
3.1 Architecture	22
3.1.1 The Producer	23
3.1.2 The Messaging System	24
3.1.3 The Consumer	26
3.2 Choice of Technologies	27
3.2.1 Source Database	27

3.2.2	Destination Database	27
3.2.3	Programming Language and Frameworks	28
4	Implementation	29
4.1	Applications and Tools	29
4.1.1	Git	29
4.1.2	Docker	31
4.1.3	Postman	32
4.1.4	DBeaver	32
4.1.5	IntelliJ	33
4.1.6	Maven	33
4.2	Spring Boot and API Setup	34
4.3	Pipeline Management	35
4.3.1	Pipeline Configuration	36
4.3.2	Pipeline Validation	38
4.3.3	API Management Endpoints	40
4.4	Pipeline Functionality	41
4.4.1	Producer	41
4.4.2	Consumer	47
4.4.3	Metrics	55
4.4.4	Pipeline Resiliency	57
4.4.5	Other Limitations	59
5	Testing	61
5.1	Unit and End-to-End Tests	61
5.2	Performance Tests	61
6	Conclusion	64
6.1	Achievements	64
6.2	Potential Improvements	65
6.3	What Went Wrong?	66
6.4	Further Work	66
	Bibliography	67

Word Count: 14143

Abstract

In today's data-driven world, Change Data Capture (CDC) plays a pivotal role in ensuring that organisations can effectively capture and utilise real-time data changes to enable prompt decision-making and drive actionable insights. This report explores the research and development process of building a flexible, easy-to-use application which captures change data from a source and applies it to a destination. The application aims to provide real-time processing of a source's change data, which is achieved by producing up to 122,137 change events per second and consuming them at less impressive rates of 15,175 per second. This end-to-end process can be closely monitored by the user via the provided metrics system, which exposes a range of fine-grained metrics that provide a deeper insight than that of existing solutions. Finally, the report reflects on the development process and suggests some potential future work to expand upon the project.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

- i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.
- ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.
- iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.
- iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see <http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420>), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see <http://www.library.manchester.ac.uk/about/regulations/>) and in The University’s policy on presentation of Theses

Acknowledgements

I would like to thank my project supervisor, Dr. Gareth Henshall, for his invaluable guidance and constant encouragement throughout the duration of this project. Additionally, I am deeply thankful for my time spent with the change data capture team at Matillion, both throughout my year in industry and beyond. I wish to extend this gratitude to my placement year manager, Ryan Begen, whose mentorship, leadership, and unwavering support have been instrumental in fostering my professional growth and development.

Chapter 1

Introduction

In today's age, data plays a crucial role in the daily operation of every organisation worldwide. This data can come in huge quantities from a diverse set of heterogeneous sources and often needs to be integrated into a single central repository for analysis in a speedy manner, in order to gain any strategic advantage out of the data [26]. We refer to this single central repository as a 'data warehouse', which can be said to be designed to enable an organisation to run powerful analytics on huge volumes of historical data in ways that a standard database cannot [18]. However, to reap the benefits of a data warehouse, we need a method to combine the data residing at our different sources. This is the process of 'data integration' and is often completed through the use of an extract-transform-load (ETL) tool. ETL tools are responsible for the extraction of data from several sources, its cleansing, customization, reformatting, integration, and insertion into a data warehouse [14]. Despite the 'transform' step being incredibly powerful, it is not always necessary. In some cases, a user may want to simply replicate their data from one source, e.g. a on-premise legacy relational database, to a more suitable destination, e.g. a cloud-hosted data warehouse.

Typically, the replication process can be accomplished using either a batch-based method or a near real-time change data capture approach. In a batch-based replication approach, the destination data is updated to reflect the new state of the source data in a periodic manner, such as hourly or daily. These periodic updates result in significant delays between when a change occurs in our source and when we observe its effect in the destination. In some cases, this isn't an issue but if we require near real-time replication of our data, we should use a change data capture approach. A change data capture approach aims to propagate our source changes to our destination in real-time. This process is incredibly efficient and consumes fewer compute resources so there is minimal, if any, performance impact on the source system. Whereas, batch processing

can become inefficient, slow and resource intensive, potentially impacting write performance of new data records [23]. Another potential issue with batch processing is the potential loss of data due to a change being overridden by another change during a single batch window. Imagine the source table we wish to replicate represents an online shopping basket, storing a row for each item a user thinks they might buy. If a user was to add an item to their basket and remove that same item within the same batch window, the downstream destination would have no indication of this activity. The state of the user's basket never changed in the eyes of the batch processing job, so no data is propagated. However, a change data capture approach would detect both an insert and delete and use that information to motivate a change in the destination database.

1.1 Motivation

The motivation to build an application of this nature stems from my time working at Matillion on their change data capture solution. Matillion has built their product around an open-source piece of software called Debezium, which handles the actual monitoring and capture of a database's change data. This design decision enables the team to concentrate on improving user experience through enhancements such as email notifications, expert-level support, and seamless integration with other Matillion products. In turn, Debezium is then managed by engineers who are better equipped to make decisions based on their domain expertise. However, there is a cost involved with this strategy. Due to the open-source nature of Debezium, anyone can contribute and drive its product direction. This can result in the implementation of features that may not align with Matillion's current goals or are challenging to integrate into our system. This reliance on a code base which we lack significant control, can often cause relatively simple features to be technically complex to implement.

This has inspired the development of a CDC engine, enabling such a system to be built without the limitations previously discussed. It is understood that to develop an application, with the same level of functionality as existing solutions, is infeasible for an individual to accomplish within the given time frame. However, there are some areas which can be improved upon, this project aims to focus on those elements. Great emphasis will be placed on ensuring the maintainability and extensibility of the project, acknowledging the potential for significant variations in each use-case across different organisations, while simultaneously ensuring that the expected performance of a CDC solution is delivered.

1.2 Aims and Objectives

The aim of this project is to build an easy-to-use change data capture application, in which a user can track and produce the sufficient change data to replicate their source database in near real-time. With this in mind, the following objectives have been set:

- Create an application which allows the user to produce the necessary change data required to replicate a source database in near real-time. This change data should be source agnostic, theoretically allowing replication to multiple different types of destination databases.
- Build an example data sink, allowing the user to consume their change data and replicate their source database in a different database technology.
- Provide the user with sufficient metrics about their pipeline during runtime. The user should be able to understand the state of their pipeline and whether or not it is producing the right amount of change data.
- Build a level of resiliency into the process, allowing a pipeline to resume its progress after an intermittent issue.

These objectives will drive the design decisions made in the following chapters and will be evaluated against to determine whether or not the project was a success.

1.3 Report Overview

This report consists of the following chapters:

- Chapter 1 gives a brief overview of the problem space, outlining the motivation to undertake a project of this nature, along with what the project aims to achieve.
- Chapter 2 describes the common approaches to implement a change data capture system, then explores some existing solutions to the problem.
- Chapter 3 provides an overview of the proposed architecture of the application, then discusses some of the chosen technologies to be used during the implementation.
- Chapter 4 discusses the implementation process of the produced application, before exploring some of its limitations.

- Chapter 5 explores the various methods for testing the functionality of the application, along with discussing the process of evaluating its performance.
- Chapter 6 concludes the report by discussing the key achievements and potential improvements to be made to the application, before discussing how the project could be expanded upon through further work.

Chapter 2

Background

This section aims to discuss some of the general ideas required to build a CDC application, before discussing some existing solutions.

2.1 Change Events

In its simplest form, a change data capture solution enables users to capture and consume individual data-changing operations as they happen within a source database. Each operation can be categorised as either a create, update or delete event, collectively they define the existence of a record. Each record must be associated with a single create event, followed by any number of updates, and optionally, a delete event.

2.1.1 Flavours of CDC Systems

Broadly speaking, a change data capture system can obtain a source's change events using either a push or pull approach [8].

With a push approach, the source database is responsible for capturing change events and pushing them to downstream services. These services then listen for incoming changes and apply them to the destination. This method typically results in reduced latency between the source and destination, as action can be taken as soon as an operation occurs. However, this comes at the cost of the source handling a lot of the computation, meaning other non-CDC workflows could be negatively impacted. Additionally, in the event that the downstream service(s) are unreachable, changes may be lost if an intermediary, queue-based messaging system is not being used.

Alternatively, a pull approach drastically reduces the workload in the source system, only requiring that each change event is logged to have happened. The downstream service(s) have the responsibility to continuously poll the source for any change events that have occurred, taking the necessary action to apply them to the destination as they arrive. Similar to a push-based approach using a queue-based messaging system, a pull-based approach allows the tracking of what changes have been processed. In the event of a failure, the pulling service can restart processing from its last known position, ensuring all change events are applied to the destination. Due to the polling nature of this approach, changes are often batched together resulting in a higher latency between source and destination. As the time between polls increases, the system becomes less of a near real-time CDC system and more like that of a batch-based migration process which consumes individual changes rather than looking for differences.

2.1.2 Common Patterns

Change events can be generated in a variety of ways, these typically involve accessing a change log which can be user-built or natively handled by the database.

Trigger-based

A trigger-based approach makes use of user defined trigger functions, which are supported by the majority of databases, to automatically build a change log in shadow tables as transactions are committed on a database, see Figure 2.1. Typically, each source table has its own associated shadow table, where every record contains the specifics of an individual operation performed on the corresponding table. The level of detail in a shadow table record can vary, ranging from a comprehensive capture of a change in its entirety to minimal information like the identifying key and operation type. A more detailed record allows for access to the full history of changes but effectively doubles the amount of data written for each transaction [15]. Similarly, the less detailed approach shares the same concern of computational overhead when trying to retrieve the actual values, due to the required join operation. Another drawback of this method is that users not only have to create these trigger functions but also have to maintain and update them to accommodate changes in the source database. Given an organisation is likely to have a range of source types, each with their own trigger functions, this approach can quickly become impractical. In a situation like this, the organisation would likely benefit from transferring the responsibility of creating a change log to the source databases themselves, rather than taking a manual approach.

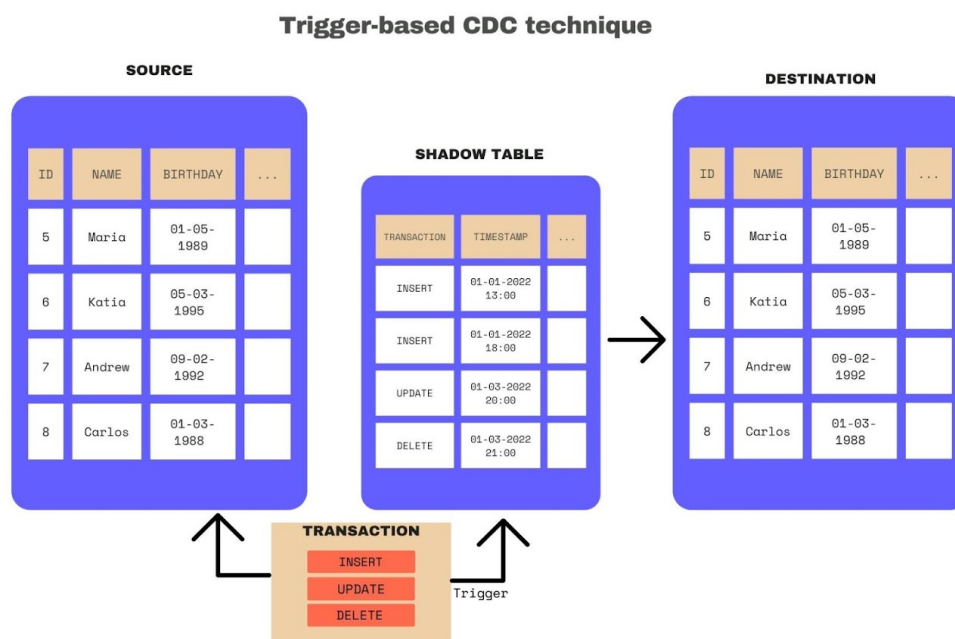


Figure 2.1: Trigger-Based Change Data Capture. [5]

Log-based

Most databases use a transaction log to record changes, in which all committed and recoverable changes can be found in these logs. In the case of a system or database crash, the transaction log enables loss-less database recovery [15]. However, we can scan and analyze the contents of the transaction log to extract the necessary change events for replication [24], see Figure 2.2. Some database technologies provide an API for log-based CDC. Others don't, and in-depth expertise is required to get changes out [15]. As of today, the log-based method appears to be the most widely adopted approach in the industry. This is primarily because it minimizes resource usage, causing minimal disruption to the daily operations of a data source, and eliminates the necessity to modify the database by introducing a shadow table [23].

2.1.3 Replication

If the change log has a record of every operation in a database's history, we can obtain a replicated state by simply consuming every change operation which has occurred in a source since its inception. However, expecting a database to retain a comprehensive record of every operation throughout its lifetime is unreasonable. Typically, the consuming CDC service only has access to a restricted history of change operations, which alone is not enough to build a replicated

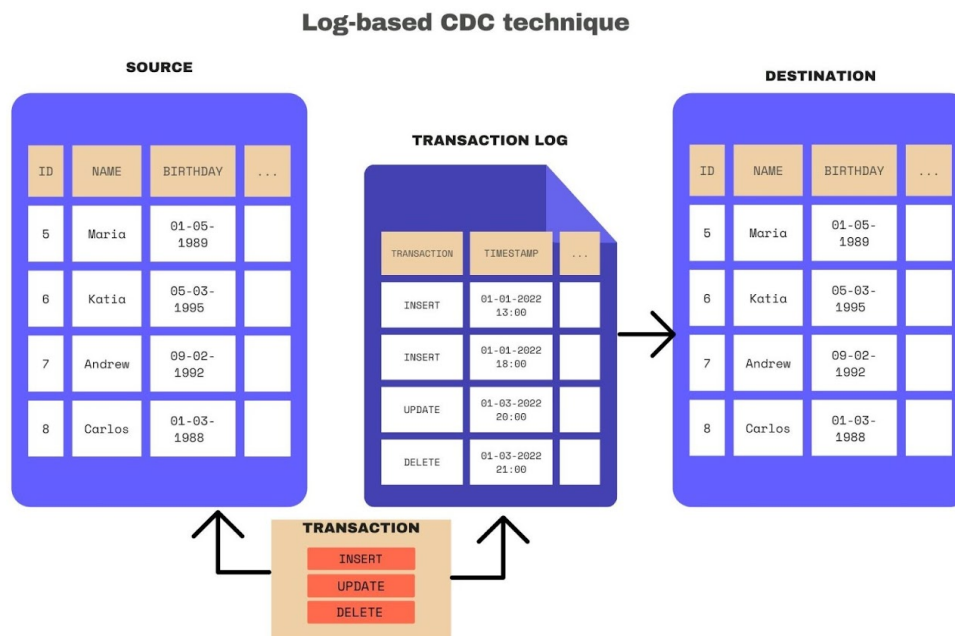


Figure 2.2: Log-Based Change Data Capture. [5]

state. In order to use this limited history for replication, a snapshot must be taken of the relevant tables to gain an initial state. Then we can apply every change event which has occurred since the time of the snapshot, giving us a replicated state consistent with the source database. Once the downstream service has successfully applied the change event to the destination, that event can be safely removed from the change log provided it has a single consumer.

2.2 Existing Solutions

2.2.1 Debezium

Debezium is an open-source CDC solution, which has the primary function of enabling a user to capture row-level change events in real time and output them as an Apache Kafka stream [19]. These changes can then be consumed by services such as the Debezium JDBC sink connector or a custom, user written consumer. Most commonly, you deploy Debezium by means of Apache Kafka Connect, a framework and runtime for implementing and operating source and sink connectors [11]. A typical end-to-end Debezium pipeline can be seen in Figure 2.3.

Debezium supports a wide range of technologies, from NoSQL databases such as MongoDB

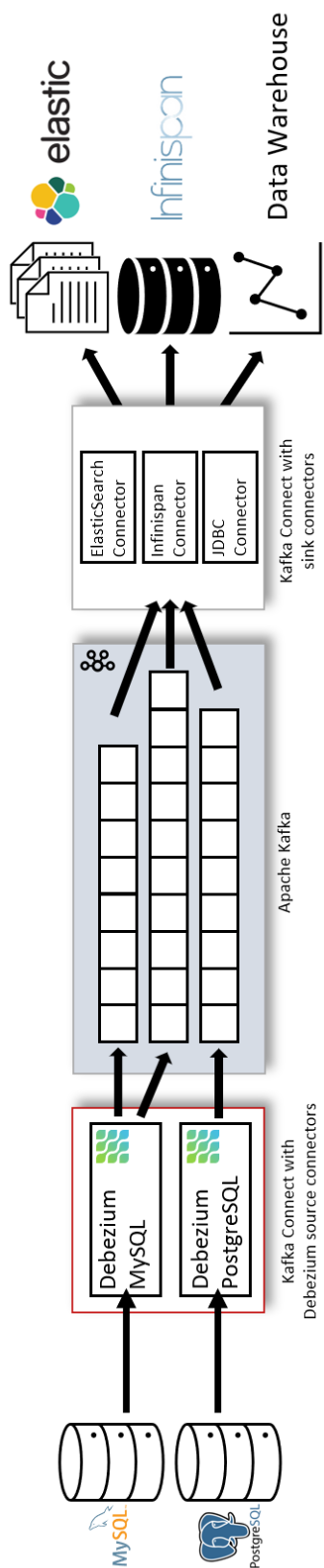


Figure 2.3: Typical Debezium Pipeline [11]

and Cassandra to relational sources like PostgreSQL, DB2 and SQL Server [13]. Despite all source connectors taking a log-based approach to CDC, reading each source's equivalent of a transaction log, the implementation of them can differ drastically between sources. A standard Debezium setup often requires a variety of additional configuration options in order to successfully integrate with an organisation's environment.

Another way to run connectors is through the embedded engine approach, which integrates Debezium as a library embedded into a custom Java application. This allows change events to be consumed within the application itself, without the need for Kafka and Kafka Connect clusters [11]. An example MySQL to MySQL workflow can be seen in Figure 2.4.

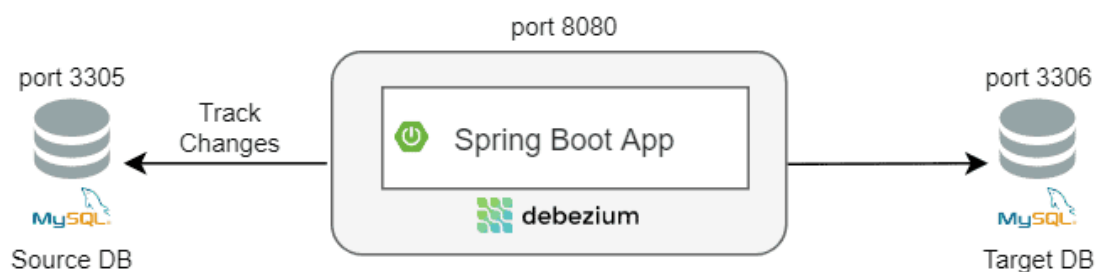


Figure 2.4: Debezium via Embedded Engine [4]

Debezium serves as the core engine of many commercial CDC products. As discussed in Section 1.1, Matillion leverages various Debezium connectors [22] to obtain the change events of a source system. With these events, Matillion allows for two types of processing, either batch-based processing of change events stored in cloud storage or a direct to data warehouse approach. The first option transforms the incoming change events into manageable units to store in a user's chosen cloud storage destination e.g. AWS S3, then consumes these units using a Matillion ETL shared job to load the data into target tables on the chosen data platform e.g. Snowflake, see Figure 2.5. If the user doesn't want to store these historical change events in cloud storage, they can opt for the direct to data warehouse approach. Currently, this option is only available for Snowflake as a destination but the ability to eliminate both the need for cloud storage and a Matillion ETL instance is invaluable. Both approaches accommodate for the following transformation types: 'Copy Table' (standard replication), 'Copy Table with Soft Deletes' (replication with soft deletes) and 'Change Log' (produces tables similar to that of a trigger-based shadow table).

Supporting two types of snapshot, blocking and incremental, Debezium satisfies the conditions

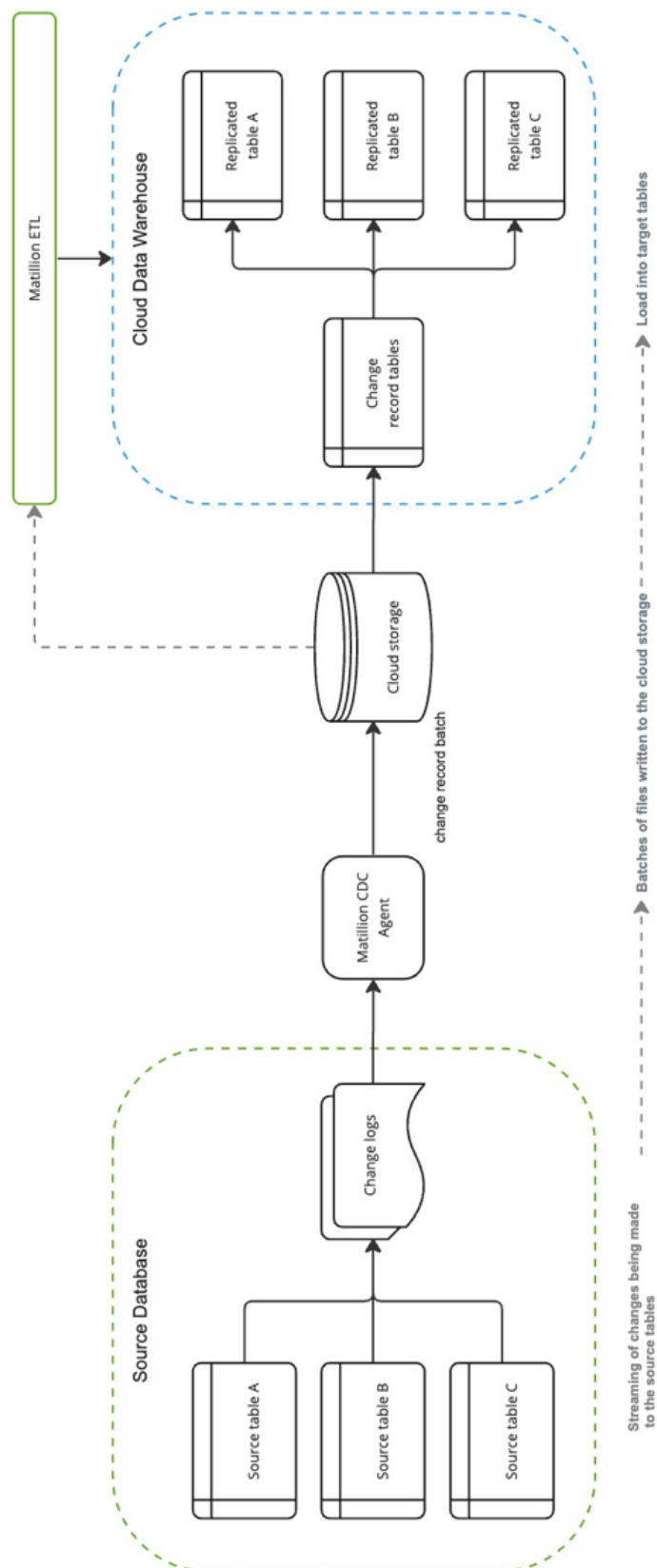


Figure 2.5: Matillion CDC via Shared Job [21]

for replication discussed in Section 2.1.3. A blocking snapshot does not allow the simultaneous streaming of change events during the snapshot and simply captures the initial state of the captured tables at a known point in time. The more complicated of the two, the incremental snapshot, takes a watermark based approach which interleaves streamed transaction log events with rows that are directly selected from tables to capture the full state of a database [2]. Essentially, instead of capturing the full state at once, Debezium captures each table in phases, in a series of configurable chunks [12]. In the event that the available change events can no longer provide accurate replication, Debezium offers the ability to trigger a new snapshot ad-hoc. This allows the user to simply apply all change events which have occurred since the new snapshot to regain their replicated state.

To supplement the core features of Debezium, a wide range of metrics are made available to the user through JMX MBeans. These metrics are divided into two categories, namely snapshot and streaming.

First, some of the more notable snapshot metrics include:

- **RowsScanned:** Provides the number of rows snapshot so far for each captured table.
- **SnapshotDurationInSeconds:** Provides the duration of the current/last snapshot.
- **SnapshotCompleted:** Indicates whether or not a snapshot has completed.
- **SnapshotRunning:** Indicates whether or not a snapshot is currently taking place.
- **CapturedTables:** Provides a list of tables captured by a connector.

These values allow the user to get a general idea of the state/progress of their snapshot. However, there is a major area for potential improvement. With the currently provided metrics, there is no information on which tables have completed their snapshot, as only a general completed flag is made available. This issue can be overcome with the utilisation of the notifications that Debezium provides, once configured a user can consume 'table scan completed' notifications from their chosen channel. However, this separation between the two gives the impression that the metrics system was designed without the user in mind and could be improved relatively easily. Additionally, although rarely important, a common issue across all of Debezium but specifically metrics, is the ambiguity of table identifiers. Throughout the system, tables are often identified with a string of the form 'schema.table'. If a schema or table name contains the splitting character, the actual name of the table is ambiguous e.g. 'exa.mp.le'.

Next, the streaming metrics offer the following useful values:

- `TotalNumberOf...(Create, Update or Delete)...EventsSeen`: Provides the total number of each type of change event produced by the connector.
- `MillisecondsBehindSource`: The number of milliseconds between the most recent change event occurring in the source and the time in which the connector processed it.
- `NumberOfCommittedTransactions`: Provides the number of transactions committed during a connector's lifetime.
- `SourceEventPosition`: Provides the offset (position in transaction log) of the last received event.

A key issue with this set of metrics is the lack of insight provided for each table. The total number of each change type is a nice metric to have but can be greatly improved upon by providing this at a table level view. '`MillisecondsBehindSource`' is incredibly important for a CDC system, as a user is likely using a replication tool of this nature for its low latency, a metric to confirm that the connector is processing in near real time is crucial.

2.2.2 Google Datastream

Google Datastream is a change data capture solution which aims to provide low latency replication through a simple, integrated experience, guiding the user through every step of the process. The solution is serverless, seamlessly scaling up or down when required, meaning users don't need to worry about managing/monitoring instances which other services often require. Additionally, change event data can be formatted in either Avro or JSON format, providing the user with further flexibility.

Datastream focuses on providing a good user experience, with the focus spanning across all aspects of the product, not just the user interface. One area where Datastream has chosen to remove potential friction is when connecting to your source database. A source can be hosted on-premises, hosted in the cloud on a self-managed virtual machine, or using a fully-managed service like Cloud SQL or Amazon RDS [16]. This flexibility allows the product to be considered in a variety of use cases, limiting a CDC solution to a small set of source environments conflicts with the core idea that we use CDC to integrate our various sources into a single location. Each of these connection configurations can be saved and re-used across multiple streams.

The user has the ability to validate various aspects of their stream configuration in the stream creation flow, see Figure 2.6. Validation of this nature allows the user to fail-fast and fix any issues early on, facilitating a smooth on-boarding process. Finally, with the addition of the stream details page, the user can monitor and get a historical view of various stream metrics such as throughput and latency. This level of monitoring can be seen in Datastream's stream status and object status system. A user can identify the general state of their stream with statuses such as 'Running', 'Failed' and 'Paused'. While maintaining the ability to view which tables/objects have been snapshot/back-filled, for complete replication.

The screenshot shows the 'Define PostgreSQL connection profile' page in Google Datastream. On the left is a sidebar with a progress indicator showing six steps: 1. Get Started (demo, PostgreSQL / BigQuery), 2. Define & test source (Not configured), 3. Configure source (Not configured), 4. Define destination (Not configured), 5. Configure destination (Not configured), and 6. Review & create (Not configured). A 'CANCEL' button is at the bottom of the sidebar. The main content area is titled 'Define PostgreSQL connection profile' and includes a description of connection profiles. A dropdown menu for 'Source connection profile *' is set to 'cp-postgresql'. Below this is a table with two rows: 'Connection profile name' (cp-postgresql) and 'Connection details' (pm-demo.cucsweskojip.us-west-2.rds.amazonaws.com : postgres). A link 'Go to this connection profile's overview' is present. The 'Test connection profile' section has a 'Run Test' button and a 'Run Test' label. At the bottom are 'CONTINUE' and 'BACK' buttons.

Define PostgreSQL connection profile

Connection profiles represent the information required to connect to a data location. If you've already defined a connection profile for your data source in the us-central1 region, choose it below. Otherwise, create one.

Source connection profile * cp-postgresql

Connection profile name	cp-postgresql
Connection details	pm-demo.cucsweskojip.us-west-2.rds.amazonaws.com : postgres

[Go to this connection profile's overview](#)

Test connection profile

Run Test to test connectivity to the PostgreSQL source from the us-central1 (Iowa) region.

[RUN TEST](#)

[CONTINUE](#) [BACK](#)

Figure 2.6: Google Datastream's connection configuration page [17]

Chapter 3

Design

This section aims to discuss the general architecture of the system at a high level, along with explaining some of the key design decisions made prior to the implementation.

3.1 Architecture

The proposed system will be divided into three core components: a producer, an intermediary messaging system and an optional consumer. The producer will handle the retrieval, transformation and emission of a source's change events. While the messaging system will act as a staging ground, allowing future consumers to process the changes in a continuous manner. The optional consumer refers to the built-in consumers that will process the change events and apply them to the downstream destination. This component is optional, allowing users to build their own external consumer, if those provided do not meet their needs.

Collectively, the three will be managed/monitored by a user-defined pipeline, see Figure 3.1. A pipeline contains the end-to-end workflow specified by the user, creating and running the necessary components to achieve the defined task. Initially, the application will only allow a single pipeline to run at once. This pipeline will be observed by a metrics service, which will track the status and various other metrics. The user will have the option to enable email notifications, which will notify them if their pipeline stops unexpectedly. Furthermore, a pipeline will be able to resume from the point of failure, foregoing the need to restart the replication process.

A user will be able to create/interact with their pipeline via an API during runtime. This method of interaction can be extended relatively easily in the future to provide a user interface. However,

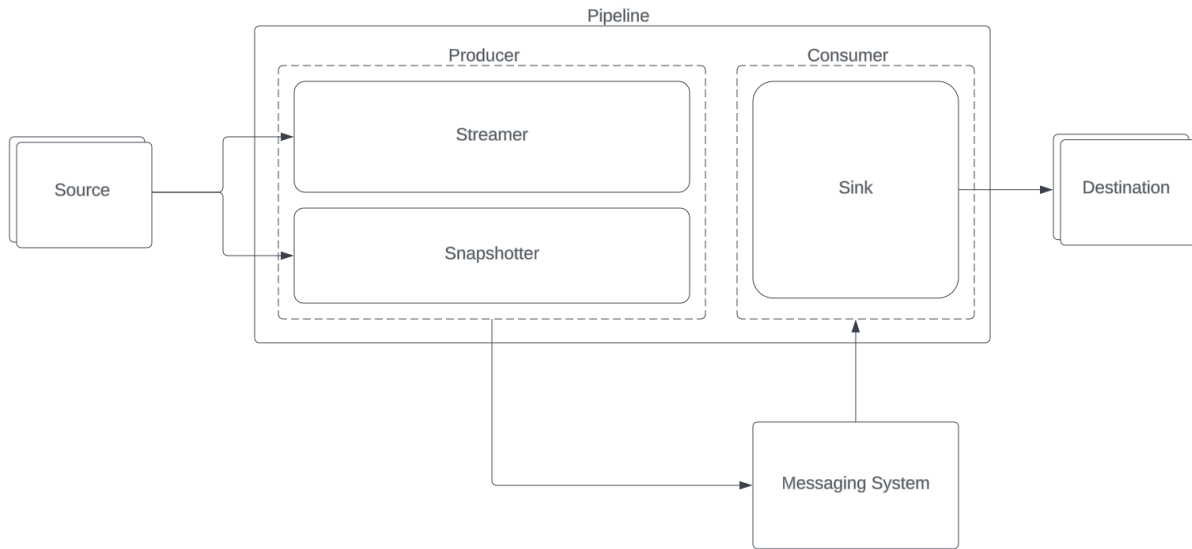


Figure 3.1: Pipeline Architecture.

there is no intention to provide this in this stage of the project. The proposed initial interactions include:

- Run Pipeline: Start up a pipeline with the provided configuration.
- Halt Pipeline: Stop running the current pipeline.
- Get Status: Retrieve a status indicating the state of the current pipeline.
- Get Snapshot Metrics: Retrieve a set of metrics describing the pipeline's snapshot.
- Get General Metrics: Retrieve a set of general metrics about the pipeline.

3.1.1 The Producer

The producer can be seen as the core engine of the application, emitting the change events to the messaging system as they are generated from the initial snapshot or the ongoing streaming process. The behaviour of these event emitting states is defined by their source-specific snapshotter and streamer, each requiring a different implementation to accommodate for the chosen source technology. The process of adding support for new sources should be relatively straightforward, as a CDC solution is only useful if can support the user's chosen source type. This idea of building extensibility into the project applies to the consumer sinks discussed in Section 3.1.3.

The change events emitted by the producer will be made up of three distinct parts: metadata, before and after. The metadata part will define the metadata associated with each event. This includes general attributes which apply to all source types such as operation type, table identifier, commit time and time produced by the application. In extension to this set, each source will have it's own source-specific metadata, which will contain the change ordering information required for replication. The remaining two subsections define the state of the record before and after the given change event, one of these may be undefined depending on the operation. When present, they will hold a list of column objects, each defining the metadata of the column and its value for that state. An example delete event can be seen in Listing 1.

3.1.2 The Messaging System

The decision to use an intermediary messaging system was reached to enable both the utilisation of built-in consumers and the creation of external consumers. Existing CDC solutions often hide the internal change events from the user, instead directly applying them to the destination and marketing the product as a replication solution. Though, change data capture isn't solely for replication purposes. Often organisations want to access the individual change events and consume them in a way they see fit. This design decision facilitates that use case, while still providing a replication solution via the built-in consumers. However, this comes at the cost of the user needing to manage/monitor the messaging system, as it is not built into the core system. However, this is not a concern as the user will already be managing their own instance of the application.

Initially, the only queue-based messaging system which will be supported will be Apache Kafka. Kafka is extremely popular across the industry, being used by 80% of all Fortune 500 companies [3]. In addition, 90% of respondents to a 2018 survey deemed Apache Kafka as mission-critical for powering many of their applications and use cases [7]. With the continued support from the open-source community, Kafka will likely be utilised within organisations for many years to come. This popularity has given rise to a wide range of documentation and support articles to assist in the construction of the application. Furthermore, Kafka is technically sound for the project, delivering latency's as low as 2ms [3], which will be instrumental in the success of the system. Finally, the solution is scalable, ensuring the the system can handle the throughput required for larger organisations.

The producer will send change events to topic(s) within a user's Kakfa instance. The user will be able to define their 'topic strategy' during pipeline creation, specifying whether they


```
1  {
2    "metadata": {
3      "tableId": {
4        "schema": "public",
5        "table": "newtable1"
6      },
7      "op": "DELETE",
8      "dbCommitTime": 1710852794460,
9      "producedTime": 1710852794462,
10     ...<source-specific-metadata>...
11   },
12   "before": [{
13     "details": {
14       "name": "position",
15       "sqlType": 4,
16       "size": 10,
17       "primaryKey": true,
18       "nullable": false
19     },
20     "value": 1
21   }, {
22     "details": {
23       "name": "name",
24       "sqlType": 12,
25       "size": 16,
26       "primaryKey": false,
27       "nullable": true
28     },
29     "value": "Peter"
30   }],
31   "after": null
32 }
```

Listing 1: An example delete event.

would like to send each event to a topic associated with the event's table name or to a single topic shared among all events.

Each event will be sent as a message in JSON format. Initially, the intention was to serialize each message in Avro format. The Avro format has two main components: the schema and the serialized data. The schema defines the structure of data, specifying the fields it contains, including data types, their names, and the relationships between them [29]. Then, the serialized data holds the actual data content. The Avro format was a candidate due to the potential performance gain when serializing/deserializing the data. However, the serialization/deserialization of an Avro message requires the use of another component to manage the ever-changing schemas, namely a schema registry, see Figure 3.2. This would mean that the user would have to support another instance. An initial test suggested that this approach struggled to handle schema changes in the source database. In result, a pivot was made to the more human-readable JSON format. Which could be said to align more with the core usability goals of the project, due its common use throughout the tech industry. The performance impact of this decision should be monitored carefully throughout its implementation, ensuring the system can still provide near real-time delivery.

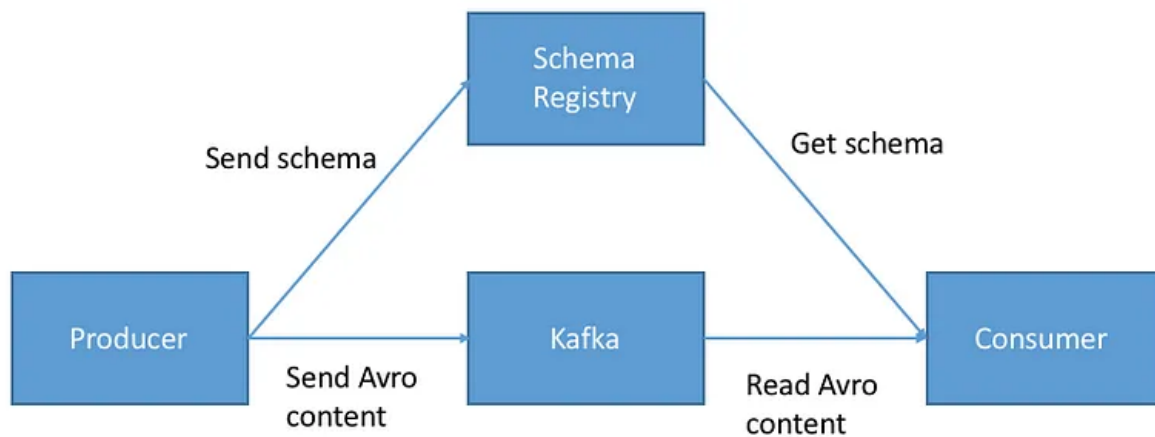


Figure 3.2: Schema Registry Architecture [20]

3.1.3 The Consumer

The consumer component allows a user to apply their change events to a destination, usually to achieve a replicated state. Each individual consumer can be referred to as a sink, with each destination type requiring a different sink implementation. A single destination can often have

multiple sinks, each applying the change events with a different consumption type. An example set of these different consumption types was observed when investigating how Matillion consumes Debezium change events, see Section 2.2.1. This project aims to provide a single example sink for a chosen destination technology, consuming change events for standard replication.

3.2 Choice of Technologies

3.2.1 Source Database

Initially, the only source database technology supported by the application will be PostgreSQL. This is largely due to its popularity, with the 2023 Developer Survey suggesting that 49% of professional developers have used the environment over the past year or would like to over the next [31]. Again, the popularity of the supported database technologies is important to the success of the product, as CDC is a very source-specific process.

PostgreSQL supports log-based change data capture through its version of a transaction log, which is known as the write-ahead log (WAL). Before the database performs the data-changing operation on the actual data, the operation is written to the WAL. Primarily the WAL ensures data consistency/durability, as in the event of a crash or failure, a consistent state of the database can be recovered by replaying the changes recorded in the logs. However, we can leverage these logs by extracting the recorded change data for our CDC application. For more information, please read Section 4.4.1.

3.2.2 Destination Database

Ideally, the initial destination supported by the application would be a data warehouse. The majority of CDC users want to integrate their various sources into a single environment, with the aim of centralized storage without sacrificing performance or scalability. However, the large scale data warehouses that organisations are interested in aren't free. Unfortunately, the project doesn't have the necessary funding required to use this technology.

However, this project aims to prove that replication is possible from the change data emitted by the producer. So at the cost of a potential loss in throughput in the consumer, the proposed destination technology will be MySQL. MySQL was chosen as it was the second most popular

database among professional developers in the 2023 Developer Survey [31]. The use of a familiar example destination should prove out the problem and allow seasoned developers to build their own destination-specific sinks for their chosen data warehouse.

3.2.3 Programming Language and Frameworks

The proposed application will require repeated interaction with an external database, the chosen programming language should provide a native database communication API with adequate transactional control. Additionally, there should be a pre-built tool allowing for easy retrieval of change data from the chosen source's transaction log, see Section 3.2.1. Java meets these two conditions, via the JDBC API and corresponding drivers. Standing out as an exceptional choice for the development of a CDC application due to its relatively good performance and great concurrency support, which will be essential when independently running the various components that define a pipeline.

To implement the API discussed in Section 3.1, the project will leverage the open-source framework named Spring Boot. This framework is widely recognised for its simplicity and reduced need for boilerplate code, which will allow the project developer to focus on the core logic of the application. Spring Boot handles the majority of the communication logic between the client and the server, only requiring the developer to define each individual endpoint.

Chapter 4

Implementation

4.1 Applications and Tools

In this section, I will discuss some of the key tools and applications used to develop the project. The application was be tested locally, with development divided across two distinct environments: half on macOS running on an M1 ARM processor and the remainder on a Windows desktop with an x86 architecture. From the offset, it was clear that the project needed a simple way to transfer the work across the two machines. Additionally, it was important to ensure that the different architectures didn't cause any alterations in behaviour.


4.1.1 Git


The project was managed within a GitHub repository, primarily to leverage git's version control capabilities. This allowed me to easily transfer code changes between my two development environments with ease, by pushing changes done in one environment and pulling them in the other.

Initially, there was an overwhelming number of features to implement within the system and it became difficult to retain all the information required to implement each of them. To overcome this, features/bugs were encapsulated within GitHub issues. Each issue would include all the necessary information to complete a specific task, either detailing the planned functionality or listing potential fixes for a described bug. An example snippet taken from a performance related issue can be seen in Figure 4.1.

Once an issue was properly planned out and understood, a new branch would be made to hold


Consumer is very slow, what can I do? #24


 Closed wukachn opened this issue 3 weeks ago · 4 comments


**wukachn** commented 3 weeks ago ...

Can any of the following properties help? `ConsumerConfig.MAX_POLL_RECORDS_CONFIG`, `ConsumerConfig.FETCH_MAX_BYTES_CONFIG`, `ConsumerConfig.MAX_PARTITION_FETCH_BYTES_CONFIG`.

I think the main performance issue stems from the jdbc connection to mysql. Avoid the massive string?




**wukachn** self-assigned this 3 weeks ago


**wukachn** commented 3 weeks ago · edited Author ...

<https://debezium.io/blog/2023/12/20/JDBC-sink-connector-batch-support/>
Major performance increase when using batched prepared statements over a big block of SQL.

Up to 5580 rows/s as of tonight.
<https://blog.devops.dev/mysql-insert-performance-analysis-6305db45335a#:~:text=throughput%20of%20around-,5500,-insertions%20per%20second>

Down to 4300 when `ON DUPLICATE KEY...` bit is appended, maybe append only for create ops.



**wukachn** commented 3 weeks ago Author ...

Can maybe try to use `Load Data`




Figure 4.1: Example GitHub Issue.

any code changes related to the issue. When the change was complete, the commits on the new branch would be pushed to the remote repository and a pull request would be created. Unfortunately, due to the nature of the project, each pull request had to be reviewed by the same developer who made the changes. To try mitigate this issue, there was attempt to let some time pass before reviewing and merging the changes, in the hopes that proper inspection of the code could be done. Once approved, the developer would merge the code changes and move onto another issue.

4.1.2 Docker

As discussed earlier, the project needed a way to ensure the application's behaviour wasn't affected by the underlying system architecture of the current machine. The chosen solution was to run the application in a Docker container, which provides a consistent and reproducible environment. Docker's encapsulation of applications and their dependencies ensures that the application runs consistently, regardless of the underlying infrastructure [1].

For local development, the use of Docker greatly increased productivity. Using a `docker-compose.yml` file, the developer was able to spin up the complete stack required for an end-end replication pipeline. This allowed the developer to easily test any changes made to the system, while providing an easy way for newcomers to try it out.

A user can run the entire application and its required services by executing the following commands:

- `mvn clean install`
- `docker-compose --profile local-postgres --profile local-mysql up --build`

The first command builds the core Java application, see Section 4.1.6. The second creates the following components: An example PostgreSQL and MySQL database to act as a source and destination for a pipeline. The core CDC application, which will receive the user's API requests, to start and manage their pipeline. Finally, a Kafka broker and an instance of Zookeeper to manage it. Each service is defined in the `docker-compose.yml` file, the PostgreSQL definition can be seen in Listing 2.

```
1 postgres:
2   image: postgres:latest
3   environment:
4     POSTGRES_DB: "postgres_db"
5     POSTGRES_USER: "postgres_user"
6     POSTGRES_PASSWORD: "postgres_password"
7   ports:
8     - "5432:5432"
9   volumes:
10    - ./db_setup/setup_postgres.sql:/docker-entrypoint-initdb.d/_
11      ↪ setup_postgres.sql
12   profiles:
13     - local-postgres
14   command:
15     - "postgres"
16     - "-c"
17     - "wal_level=logical"
18     - "-c"
19     - "wal_receiver_timeout=0"
```

Listing 2: Defining a PostgreSQL container.

4.1.3 Postman

Another application which aided local development tremendously was Postman. Postman provides an easy-to-use user interface for sending HTTP requests to APIs, see Figure 4.2. Throughout the development period, any manual testing was done through this interface. Additionally, Postman allows users to create and export collections of API requests. This allowed the developer to create a set of example requests which a new user can easily import to test out the system, the exported collection is named `postman_collection.json`.

4.1.4 DBeaver

DBeaver Community is a free cross-platform database tool which facilitates database interaction using a user interface, via a JDBC connection. It includes a built-in SQL editor with syntax highlighting and supports SQL generation. DBeaver allowed the developer to interact with the source and destination databases during manual testing with ease, as both could be accessed from a single application.

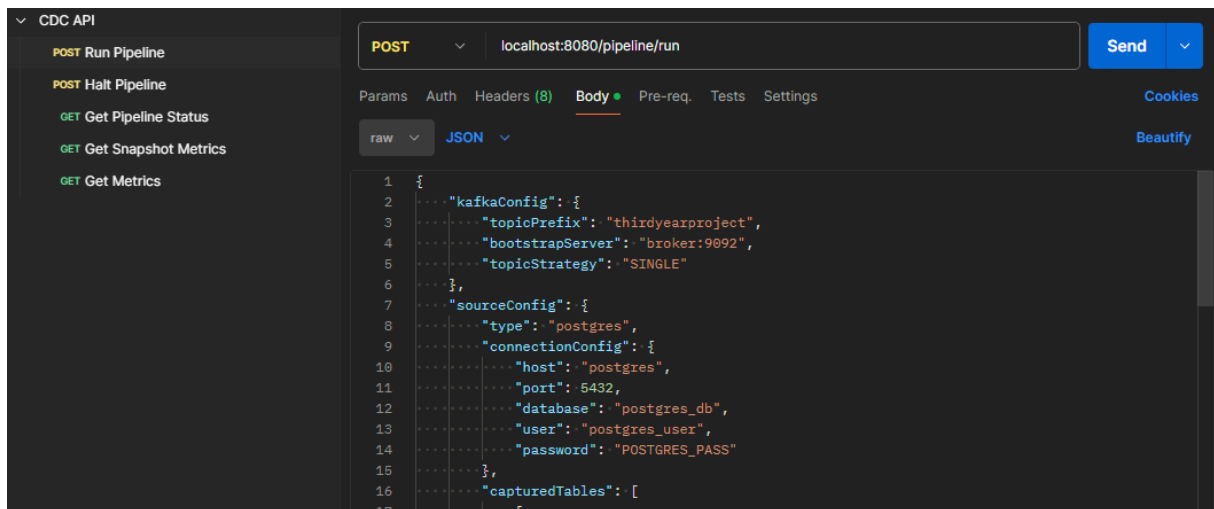


Figure 4.2: Postman User Interface.

4.1.5 IntelliJ

The core application was written within an IDE, namely IntelliJ IDEA Community Edition. IntelliJ provides seamless integration with Git and Docker, both can be used/monitored entirely through an IntelliJ window. Additionally, IntelliJ offers a wide range of features that make writing code a much more efficient experience. Some of the more notable features include: intelligent code completion, powerful refactoring tools, auto styling and the ability to jump to a declaration.

4.1.6 Maven

Maven is an open-source build automation and project management tool widely used for Java applications [6]. Maven automates the source code compilation and allows for pain-free dependency management. A developer manages an application's dependencies through a series of `pom.xml` files. These files contain the dependency definitions of a project, which allow the developer to manage the versions and scope of each dependency. When the developer compiles the project, these dependencies will be installed and integrated into the final build. The application can be compiled via the `mvn clean install` command. The *clean* part removes any previously created artefacts, then the *install* part builds the application, installing any dependencies which aren't already installed.

4.2 Spring Boot and API Setup

In order to manually test the application's functionality during development, the first feature to be implemented was the endpoint which starts a pipeline, see Section 4.3.3.

First, the core application was configured. The developer used the online `Spring Initializr` tool [30] to generate an initial maven project, which used the Spring Web starter and a few other dependencies. Through Spring Boot's auto configuration, the necessary components required for the application were automatically configured. This included configuring an embedded web server, Tomcat by default, without having to manually install and configure an external one. Along with `DispatcherServlet`, which is responsible for handling HTTP requests that come from clients [25].

With the prior setup, the developer could start to create API endpoints. First, a controller class must be created, which will define the various endpoints available to a user. To do this, the class is annotated with `@Controller`, which allows spring to handle its creation and prepare it to handle requests. In this application, a single controller named `PipelineController` defines all the endpoints related to a users pipeline. Each endpoint's path will begin with `/pipeline`, to avoid repeating this when defining each endpoint's method, we can annoate the class with `@RequestMapping("/pipeline")`.

Now that a controller exists to contain our different endpoints, we can define them. The code defining the endpoint to run a pipeline can be seen in Listing 3 and will be described in more detail in Section 4.3.3. The annotation `@PostMapping("/run")` defines the desired HTTP method of the request, being POST. While specifying the rest of the endpoint's path, here the complete path will be `/pipeline/run`. Next, the associated method is defined, with its body specifying the action to be taken. The method parameter and annotation `@RequestBody` specifies the expected structure of the request body, here a `PipelineConfiguration` object is expected, see Section 4.3.1. This object will be received in JSON form and converted into an instance of `PipelineConfiguration`.

If a user tries to make a request to an invalid endpoint or uses an invalid request body, the application will respond accordingly with the corresponding HTTP status code. However, in order to have the application react to custom exceptions thrown during a request, an exception handler class must be created. In this application, the class is called `ApiExceptionHandler` and is annotated with `@ControllerAdvice`. This annotation allows the developer to write methods

```
1 @PostMapping("/run")
2 public void runPipeline(@RequestBody PipelineConfiguration config)
   ↳ throws PipelineConflictException, ValidationException {
3     PipelineInitializer.runPipeline(config, metricsService);
4 }
```

Listing 3: Run Pipeline Endpoint Code.

which handle the the various custom exceptions thrown during a request's lifetime. Listing 4 showcases the handler which will be invoked when a user tries to run multiple pipelines at once. The user will receive an error message back, accompanied by a HTTP conflict status code. The application handles the following situations using these handlers:

- The user tries to run multiple pipelines at once. (Conflict)
- The user tries to halt a pipeline when one isn't running. (Not Found)
- The user's pipeline fails the initial validation. (Bad Request)

```
1 @ResponseStatus(HttpStatus.CONFLICT)
2 @ExceptionHandler(PipelineConflictException.class)
3 public ResponseEntity<ErrorMessage>
   ↳ handleConflict(PipelineConflictException ex) {
4     var errorMessage =
       ↳ ErrorMessage.builder().message(ex.getMessage()).build();
5     return new ResponseEntity<>(errorMessage, HttpStatus.CONFLICT);
6 }
```

Listing 4: Pipeline Already Running Handler.

4.3 Pipeline Management

In this section, the configuration of a pipeline and its validation will be discussed. Then, the two endpoints used to manage the pipeline will be defined, along with the mechanism used to handle a pipeline's runtime.

4.3.1 Pipeline Configuration

Ultimately, the `PipelineConfiguration` class defines the behaviour of a user's pipeline. An instance's definition specifies various details related to the different components which make up the end-to-end workflow. Each pipeline configuration is made up of up to four configuration blocks, which will be discussed in the following sub-sections.

Kafka Configuration

The user must specify the following details related to their Kafka instance:

- Bootstrap server address.
- Topic strategy.
- Topic prefix.

The bootstrap server address will be used to connect to a user's Kafka cluster, the pre-defined local environment uses the value `broker:9092`. To add further flexibility, the topic strategy specifies whether a user wants to send all their events to a single topic or to separate table-based topics. This value is defined as either `PER_TABLE` or `SINGLE`, any other value is invalid. Finally, the topic prefix refers to the string which will be prepended to the beginning of each topic's name. If the user used the `SINGLE` topic strategy and defined their prefix as "cdc-app", then the topic used through the application would be named "cdc-app.all_tables".

Source Configuration

This configuration is source-specific, so its implementation in the codebase must allow for easy extension when more sources get supported. This was done by creating a common interface named `SourceConfiguration`, which leverages JSON sub-types using the annotations: `@JsonTypeInfo` (use = `JsonTypeInfo.Id.NAME`, property = "type") and `@JsonSubTypes.Type` (value = `PostgresSourceConfiguration.class`, name = "postgres"). In short, these annotations allow for a user to specify the source type using a `type` attribute. Then, the application can do the mapping from the request body to the correct class implementation of `SourceConfiguration`.

The PostgreSQL source configuration block consists of the following:

- PostgreSQL connection configuration.

- Set of captured tables.
- (Optional) Name of replication slot.
- (Optional) Name of publication.

First, the connection configuration holds all the details required to connect to the user's PostgreSQL database. These details include the port, host, database name, username, password reference and any additional JDBC properties. For security purposes, passwords are expected to be stored as environment variables throughout the application, the password reference is the name of its corresponding environment variable. Then, the user defines the set of tables they wish to capture, these are specified using a custom `TableIdentifier` object to avoid ambiguity, see Listing 5. Finally, the user has the option to specify the publication and replication slot name used by the application. If not specified, the application will use the default values of "cdc_publication" and "cdc_replication_slot".

```
1  @Value(staticConstructor = "of")
2  public class TableIdentifier {
3      String schema;
4      String table;
5
6      @JsonIgnore
7      public String getStringFormat() {
8          return String.format("%s.%s", schema, table);
9      }
10 }
```

Listing 5: Table Identifier Class Definition.

(Optional) Destination Configuration

Optionally, the user can provide a destination configuration. If not provided, the application will not consume any changes. This block uses the same JSON sub-type method as the source block.

The MySQL destination configuration block is made up of the following:

- MySQL connection configuration.
- Sink type.

The connection configuration holds similar values to that of the PostgreSQL version. Whereas, sink type refers to the method of consumption, either having a value of `BATCHING` or `TRANSACTIONAL`, the two options are discussed in Section 4.4.2.

(Optional) Email Notifications Configuration

Another optional configuration option defines the details needed to enable email notifications for when a pipeline fails unexpectedly, see Section 4.4.4. These details include:

- Sender email.
- Sender password reference.
- List of recipient emails.

First, the user provides the email in which the notifications will be sent from, along with the name of the environment variable which stores the value of its password. Then, the list of recipient emails specifies the various email addresses to send the pipeline failure notification to.

4.3.2 Pipeline Validation

Before the system creates and executes the user-defined pipeline, its configuration is put through a set of initial validation checks. This validation phase allows the user to fail-fast, giving the opportunity to make amendments to fix any detected issues before the pipeline is deep into its execution.

These validation checks may be technology-specific, so any interfaces used to generalise parts of `PipelineConfiguration` must provide an implementation of a stubbed `validate()` method. The implemented methods will then be called when the entire pipeline configuration is validated, provided that the configuration block is present, see Listing 6. If any sub-check fails, an exception will be thrown, which will then be wrapped in a `ValidationException` to be handled accordingly.

When validating a PostgreSQL source configuration, several checks are completed. First, the database's `wal_level` must be set to “logical”, ensuring compatibility with the system's streaming process, see Section 4.4.1. Then, the system verifies that neither a publication nor a replication slot currently exists with the same name specified by the user. Finally, the connecting database user must have the necessary permissions for the application.

```
1 public void validate() throws ValidationException {
2     log.info("Validating pipeline.");
3     try {
4         log.info("Validating source.");
5         sourceConfig.validate();
6
7         if (destinationConfig != null) {
8             log.info("Validating destination.");
9             destinationConfig.validate();
10
11             log.info("Validating sink compatibility.");
12             validateSinkCompatibility();
13         }
14
15         if (emailConfig != null) {
16             log.info("Validating email configuration.");
17             emailConfig.validate();
18         }
19     } catch (PipelineConfigurationException | SQLException |
20     ↪ SourceValidationException e) {
21         log.info("Pipeline has failed validation.", e);
22         throw new ValidationException("Pipeline has failed
23         ↪ validation.", e);
24     }
25     log.info("Pipeline has passed initial validation.");
26 }
```

Listing 6: Pipeline Configuration Validation Code.

A PostgreSQL source connection gets indirectly checked to ensure that the application can actually connect to the source database. Whereas, MySQL's sole validation check is to test its connection. Ideally, the system would check the user has the necessary permissions for their chosen sink. However, MySQL lacks an easy way to check user permissions without significant investment. For this reason, permission checks for MySQL have been defined as a potential piece of future work.

Unfortunately, all sink types are not compatible with all topic strategies, so the combination of the two must be validated when an example sink has been configured. A `TRANSACTIONAL` sink can only be used with a `SINGLE` topic strategy and a `BATCHING` sink is only compatible with a `PER_TABLE` topic strategy, see Section 4.4.2.

Finally, the only validation in place for the email notification configuration is to check that an environment variable with the given reference is available. This check is completed indirectly for any other configuration blocks which require a password.

4.3.3 API Management Endpoints

A pipeline can be managed by a user through two management endpoints, enabling them to run or halt a pipeline. The `PipelineInitializer` class manages the application's pipeline using two static methods, `runPipeline()` and `haltPipeline()`. This class tracks the thread on which a user's pipeline is executing, using a static attribute named `pipelineThread`.

Run Pipeline

The user starts their pipeline by sending a POST request to the `/pipeline/run` endpoint, the body of the request should contain their pipeline configuration. Upon receiving the request, the system will call the `runPipeline()` method. First, the method checks to see if a pipeline is already running. If this is the case, a `PipelineConflictException` is thrown, causing the system to respond with a HTTP conflict code, see Section 4.2. If the user doesn't have a currently running pipeline, the system will validate the provided pipeline configuration, see Section 4.3.2. A `ValidationException` will be thrown if any of the checks fail, triggering a HTTP bad request response. Finally, upon passing initial validation, a new pipeline will be created and ran on the thread referenced by `pipelineThread`.

Halt Pipeline

A user can halt their pipeline by sending an empty POST request to the `/pipeline/halt` endpoint, which calls the `haltPipeline()` method. If a pipeline is running, the pipeline thread is terminated, completing any necessary cleanup operations within the pipeline. However, if no pipeline is active, a `PipelineNotRunningException` is thrown in order to respond with a HTTP not found code.

4.4 Pipeline Functionality

Each pipeline instance is defined by the `Pipeline` class, which implements the `Runnable` interface and defines the steps a pipeline must take in an implementation of the `run()` method. Without including steps related to pipeline metrics, these steps include:

1. Create and run consumers (if applicable).
2. Snapshot the source.
3. Initiate streaming of the source.

4.4.1 Producer

As discussed in Section 3.1.1, the producer handles the creation and emission of a source's change events. These events are captured in the initial snapshot phase using an implementation of a `Snapshotter` or in the streaming phase with an implementation of a `Streamer`. Throughout this subsection, any references to either of these interfaces will refer to their PostgreSQL implementation.

Kafka Producer

In order for the `Snapshotter` and `Streamer` to be able to send messages to Kafka, both need access to a class called `ChangeEventProducer`. This class provides the simple method **public void sendEvent**(`ChangeEvent changeEvent`), allowing the two producing components of a pipeline to simply pass each generated change event without worrying about the details of sending each event to Kafka.

`ChangeEventProducer` utilises the class `KafkaProducerService` to actually send each event, see Listing 7. Here the `sendEvent` method determines the corresponding topic for each change

event, based on the chosen topic strategy. Then sends the message using a `KafkaTemplate<String, ChangeEvent>` instance. This instance is configured to serialize the topic name using the provided `StringSerializer`, while serializing each change event using a custom `ChangeEventSerializer`, see Listing 8.

```
1 public void sendEvent(ChangeEvent changeEvent, TopicStrategy
  ↳ topicStrategy) {
2     String topic;
3     if (topicStrategy == TopicStrategy.PER_TABLE) {
4         topic = String.format("%s.%s", topicPrefix,
  ↳ changeEvent.getMetadata().getTableId().getStringFormat());
5     } else {
6         topic = String.format("%s.all_tables", topicPrefix);
7     }
8     CompletableFuture<SendResult<String, ChangeEvent>> future =
9     kafkaTemplate.send(topic, changeEvent);
10    ...
11 }
```

Listing 7: Code for creating topics

```
1 public class ChangeEventSerializer implements
  ↳ Serializer<ChangeEvent> {
2
3     private final Gson gson = new Gson();
4
5     @Override
6     public byte[] serialize(String topic, ChangeEvent changeEvent) {
7         return gson.toJson(changeEvent).getBytes();
8     }
9
10    @Override
11    public void close() {}
12 }
```

Listing 8: ChangeEventSerializer Code.

Snapshotter

The snapshot behaviour of a source is defined by an implementation of the abstract `Snapshotter` class, which defines abstract methods to be implemented in order to execute the complete snapshot using the `snapshot()` method, see Listing 9. This approach to snapshotting was inspired by Debezium's `RelationalSnapshotChangeEventSource` class [10], taking a similar approach to provide clear and consistent snapshot implementations across different source types. The remainder of this subsection will describe the process of taking a PostgreSQL snapshot.

```

1  public void snapshot(Set<TableIdentifier> tables) throws
    ↳ PipelineException, IOException {
2      log.info(String.format("Starting snapshot on the following tables:
    ↳ %s", String.join(", ", tables.stream().map(TableIdentifier::getS_
    ↳ tringFormat).toList())));
3
4      metricsService.startingSnapshot();
5      try {
6          log.info("Step 1: Setting up start point of snapshot.");
7          createSnapshotEnvironment(tables);
8
9          log.info("Step 2: Snapshotting structure of tables");
10         captureStructure(tables);
11
12         log.info("Step 3: Snapshotting content of tables");
13         snapshotTables(tables);
14
15         log.info("Snapshot Complete.");
16     } catch (Exception e) {
17         throw new PipelineException("Pipeline failed during
    ↳ snapshotting phase.", e);
18     } finally {
19         snapshotComplete();
20     }
21     metricsService.completingSnapshot();
22 }

```

Listing 9: General Snapshotter Steps.

When configuring the snapshot environment, the application aims to start a transaction at a

known Log Sequence Number (LSN), which provides a consistent view of the database throughout the entire snapshot. An LSN refers to a position in the WAL. To do this, a replication slot is created using the following SQL statement: `CREATE_REPLICATION_SLOT <slot_name> LOGICAL pgoutput;`. This statement creates a logical replication slot and provides an internal snapshot of the source, returning the LSN of the snapshot and an associated name to reference it. A replication slot can be seen as a reservation on the WAL entries from a given LSN onwards, ensuring no logs are flushed from that point until it has been progressed. This replication slot will be used during the streaming phase in conjunction with a publication, to retrieve any associated WAL entries from the given LSN onwards. The application pre-emptively creates the publication for the captured tables with the SQL statement `CREATE PUBLICATION <publication_name> FOR TABLE <table_id_csv>;`. A publication is essentially a group of tables whose data changes are intended to be replicated through logical replication [27], essentially defining the group of tables we will be monitoring through the WAL in the streaming process.

Given the previous steps, the system can obtain the required consistent view of the database by executing the following two statements without committing, starting the transaction which will be used to snapshot the data:

- `SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;`
- `SET TRANSACTION SNAPSHOT '<snapshot_name>';`

The first sets the transaction isolation level to `Repeatable Read`, meaning only data committed before the transaction began will be seen within it and no concurrent transaction will effect the view of the snapshot transaction. The second sets the state of the transactions view to that of the snapshot taken at the LSN provided by the replication slot. It's worth noting that the snapshot transaction should be created on a different JDBC connection to that which created the replication slot. If the replication slot connection closes or has another statement executed on it, the snapshot becomes invalid.

Next, the snapshotter captures the structure of each captured table. This process is defined in a custom method which works across different JDBC connections and will be able to be reused across source types. These details are captured in a map and will be referred to when capturing the actual data.

Now that we know the structure of each table, we can begin to snapshot the actual data. This can be done using a simple `SELECT * FROM <schema>.<table>` statement for each table.

Then iterating through the result set, building/sending an event per row. The column details of each column gets embedded into the change event and are used to retrieve the correct data type from the result set for each column. The operation of these events will be defined as `READ`, specifying the event has been produced from the initial snapshot and the LSN of the event is the LSN associated with the replication slot. Each event is sent to Kafka using the `sendEvent()` method discussed previously.

By default, the PostgreSQL driver will attempt to load the entirety of a table's contents into the application's memory at once, causing obvious memory issues for large tables. To avoid this issue, the driver can access the results of the query using a cursor, which will retrieve the content in manageable chunks. To do this, the connection must not be in auto-commit mode and the connection fetch size must be set to the number of rows wanted per chunk [28], the application uses a value of 1024.

Once the snapshot is complete, the user should have the necessary data to create the start point of their replicated state downstream. Finally, the two JDBC connections used during the snapshot process can be closed.

Streamer

The streaming behaviour is defined in a similar way to the snapshot behaviour, following the same abstract principles to lay out the steps of the process, see Listing 10. The remainder of this subsection will describe the process of streaming new change events from a PostgreSQL source.

```
1 public void stream() throws PipelineException {  
2     log.info("Starting to stream changes.");  
3     try {  
4         initEnvironment();  
5         streamChanges();  
6     } catch (Exception e) {  
7         throw new PipelineException("Pipeline failed during streaming  
8             ↳ phase.", e);  
9     }  
}
```

Listing 10: General Streamer Steps.

First, to setup the streaming environment for PostgreSQL, the application must create a replication stream, see Listing 11. A replication stream refers to a stream of changes that have been recorded to the WAL, which can be polled by the application. During the creation of the replication stream, we specify the publication and replication slot. This ensures the stream only contains entries related to the tables in our publication and limits the entries to those which occurred after the initial LSN of the replication slot.

```
1  @Override
2  protected void initEnvironment() throws SQLException {
3      BaseConnection conn = (BaseConnection)
4          ↪ replicationConnection.getConnection();
5      this.replicationStream =
6          conn.getReplicationAPI()
7              .replicationStream()
8              .logical()
9              .withSlotName(replicationSlot)
10             .withSlotOption("proto_version", 1)
11             .withSlotOption("publication_names", publication)
12             .start();
13  }
```

Listing 11: Initiating PostgreSQL Streaming Environment.

Once the replication stream has been created, we can start to read from it, see Listing 12. The application will continue to check for new entries using the non-blocking `readPending()` method. This method retrieves the next WAL record from the source if one is available, otherwise returning null. It was chosen to use the non-blocking version over the blocking alternative to ensure compatibility with the systems halting mechanism, see 4.3.3. If a WAL record is returned, the current LSN is retrieved and the message is processed using an instance of the class `PgOutputMessageDecoder`. It's worth noting that the replication stream provides one WAL record at a time but will always group records by transaction, so all records between a `Begin` and a `Commit` will be part of the same transaction.

The application's `PgOutputMessageDecoder` class is heavily influenced by Debezium's `PgOutputMessageDecoder` [10] and GitHub user `davyam`'s `Decode` class [9]. The purpose of this class is to decode and construct change events from the relevant logical replication messages retrieved from the replication stream, which are in `ByteBuffer` form on arrival. The class

```
1  @Override
2  protected void streamChanges() throws SQLException, IOException {
3      try {
4          while (!replicationStream.isClosed()) {
5              var message = replicationStream.readPending();
6              if (message == null) {
7                  continue;
8              }
9              var lsn = replicationStream.getLastReceiveLSN();
10             pgOutputMessageDecoder.processNotEmptyMessage(message, lsn);
11         }
12     } finally {
13         replicationStream.close();
14         replicationConnection.close();
15     }
16 }
```

Listing 12: PostgreSQL Streaming Code.

tracks the current structure of each captured table, with `Relation` messages indicating that the structure must be recaptured as a change was detected. This structure will be used to decode the `Insert`, `Update` and `Delete` messages as they arrive. Additionally, the application is interested in `Begin` and `Commit` messages, as they contain interesting metadata like the LSN of the commit operation and the transaction ID.

In order to associate a commit LSN with a transaction's operations, the application must queue the decoded changes until a corresponding `Commit` is decoded. Once the commit has been processed, the change events will be amended to include the commit LSN, before being sent to their corresponding Kafka topics. This process is handled by the `PostgresTransactionProcessor` class.

4.4.2 Consumer

The consumer retrieves change events from Kafka topics and applies them to the destination. These consumers are created as a pipeline starts up and will continue to process events until the pipeline stops.

Initially, the project aimed to deliver a single transactional sink. However, upon implementation, the performance of the sink was not at the level expected by a CDC system. The decision was made to develop a second batching sink, with the aim to sacrifice transactional guarantees for better performance.

Creating Consumers

Given that a user has provided a configuration for a destination, consumer(s) must be created on pipeline startup in order for the sinks to apply the changes downstream. This process is handled by the `KafkaConsumerManager` class using the static method `createConsumers()`.

First, if the required topics don't already exist, the application will create them. Topics can be named in one of two ways, depending on the user's chosen topic strategy. When using a `SINGLE` approach, the application uses a single topic named `<user_prefix>.all_tables`. Whereas, a `PER_TABLE` topic strategy uses a topic per captured table, each with the name `<user_prefix>.<schema>.<table>`. These topic(s) are created through the use of an `AdminClient` instance, see Listing 13.

Next, the application will create the consumers, running each of them on a new thread. With a `SINGLE` topic strategy, only one `ChangeDataConsumer` will be initiated. Whereas, a `PER_TABLE` approach will initiate a `ChangeDataConsumer` for each schema present within the captured tables, in an attempt to balance the load. The general Kafka consumer behaviour is defined in the `ChangeDataConsumer` class, handling the polling of change events from the Kafka topic(s). Each `ChangeDataConsumer` instance uses an implementation of the `ChangeEventSink` interface to process and apply the retrieved change events.

Consuming from Kafka

The `ChangeDataConsumer` class utilises a `KafkaConsumer<String, ChangeEvent>` instance to consume change events from Kafka. This instance is configured to retrieve up to ten thousand change events per poll, the value was optimised to maximise the throughput of the two sink types. The retrieved change events are deserialized using the custom `ChangeEventDeserializer` class.

The main `run()` method defines the core loop for the `ChangeDataConsumer`, see Listing 14. First, the `KafkaConsumer` is instantiated. Then, the consumer subscribes to the relevant topics and begins to poll for change events. This polling operation has a timeout of two seconds, if ten


```

1  private static void createTopicsIfNotExists(
2      String bootstrapServer,
3      String topicPrefix,
4      Set<TableIdentifier> tables,
5      TopicStrategy topicStrategy) {
6      Properties adminProps = new Properties();
7      adminProps.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
8          ↪ bootstrapServer);
9
10     try (AdminClient adminClient = AdminClient.create(adminProps)) {
11         if (topicStrategy == TopicStrategy.PER_TABLE) {
12             for (var table : tables) {
13                 var topicName = topicPrefix + "." +
14                     ↪ table.getStringFormat();
15                 createTopic(adminClient, topicName);
16             }
17         } else if (topicStrategy == TopicStrategy.SINGLE) {
18             var topicName = topicPrefix + ".all_tables";
19             createTopic(adminClient, topicName);
20         }
21     } catch (Exception e) {
22         log.error(e.getMessage(), e);
23     }
24 }
25
26 private static void createTopic(AdminClient adminClient, String
27     ↪ topicName) throws ExecutionException, InterruptedException {
28     boolean topicExists =
29         ↪ adminClient.listTopics().names().get().contains(topicName);
30     if (!topicExists) {
31         NewTopic newTopic = new NewTopic(topicName, 1, (short) 1);
32         adminClient.createTopics(Collections.singletonList(newTopic)).all()
33             ↪ get();
34     }
35 }

```

Listing 13: Create Topics Code.

thousand events aren't available, the operation will pull what it can. Finally, if the polling operation retrieved any events, they are processed by an implementation of the `process()` method from the `ChangeEventSink` interface.

Shared Sink Behaviour

Despite the two MySQL sinks having unique methods of delivering changes to the destination, the two share some similarities. Both share the same initial `process()` method, only differing in their implementation of the `deliverChanges()` method, see Listing 15.

First, the two sinks order their change events by their commit LSN, then by their operation LSN. This means that all transactions are in commit order and all operations within a transaction are in the order they took place. In theory, the Kafka topic(s) should preserve this order but to ensure that there are no discrepancies due to the asynchronous nature of sending events to Kafka, the events are reordered at the start of each polled batch.

The value used to order the events can be referred to as the offset. A PostgreSQL offset is a string of 39 characters, 19 for the commit LSN and 19 for the operation LSN, with the two separated by a tilde character. Each LSN is in numerical form. To ensure that two LSNs of different lengths can be easily compared, these number values get padded to 19 characters with prepended zero's. Change events generated during the snapshot phase do not have a commit LSN, so the first part of their offset will contain 19 zeros. An example PostgreSQL offset would be `00000000000123456789~00000000000123456123`.

Once sorted, the sinks will ensure that both the databases and tables exist in the destination. A MySQL database can be seen as the equivalent of a PostgreSQL schema. To ensure the databases exist, the sink will find all schemas present in the set of change events, then execute a series of **CREATE DATABASE IF NOT EXISTS** statements. A similar approach is taken with the tables, finding the first non-delete change event per table, then building a series of **CREATE TABLE IF NOT EXISTS** statements from the "after" blocks. This approach was taken as it provides a simple solution to the problem, without much performance impact. To avoid the unnecessary execution of these statements after the first time, the application could track which tables have been created. However, this information would become inaccurate as soon as a user drops a resource within MySQL, meaning constant monitoring would be required.

```
1  @Override
2  public void run() {
3      try (var consumer = createConsumer(bootstrapServer)) {
4          List<String> topics;
5          if (topicStrategy == TopicStrategy.PER_TABLE) {
6              topics = tables.stream().map(table -> topicPrefix + "." +
7                  ↪ table.getStringFormat()).toList();
8          } else {
9              topics = List.of(topicPrefix + ".all_tables");
10         }
11
12         consumer.subscribe(topics);
13         while (!isSoftInterrupted) {
14             ConsumerRecords<String, ChangeEvent> consumerRecords =
15                 ↪ consumer.poll(Duration.of(2000, ChronoUnit.MILLIS));
16             List<ChangeEvent> changeEvents =
17                 ↪ StreamSupport.stream(consumerRecords.spliterator(),
18                 ↪ true)
19                 .map(ConsumerRecord::value)
20                 .collect(Collectors.toList());
21             if (!changeEvents.isEmpty()) {
22                 eventProcessor.process(changeEvents);
23                 metricsService.consumeEvents(changeEvents);
24             }
25         }
26     } catch (Exception e) {
27         consumerExceptionHandler.handle(new ConsumerException("Consumer
28             ↪ Error.", e));
29     } finally {
30         log.info("Consumer closed.");
31     }
32 }
```

Listing 14: ChangeDataConsumer Core Loop Code.

```
1  @Override
2  public void process(List<ChangeEvent> changeEvents) {
3      Collections.sort(changeEvents);
4
5      createDatabasesIfNotExists(changeEvents);
6      createTablesIfNotExists(changeEvents);
7
8      // Each row gets updated only when the new_offset >
9      //   old_offset.
10     deliverChanges(changeEvents);
11 }
```

Listing 15: Common Sink Process Method Code.

During table creation, both sinks add an extra column with the name `cdc_last_updated` to each table. This column tracks the offset of the change event which last updated a particular row. As change events are processed, an event will only change the data if its offset is newer than the offset which caused the last update. This mechanism ensures that each row in the destination reflects the most up to date change event processed by the sink.

The two sinks begin to diverge as they start to apply the change events to the destination. However, both share the same approach to building schema change statements. The `compareStructureAndBuildSchemaChange()` method compares the “before” and “after” blocks of an event. If a difference in structure is found, the required **ALTER TABLE** statements will be built and returned to the sinks to be executed as needed. For the two example sinks, these statements will be executed before applying the change event which produced the schema change. Currently, this method doesn’t support column name changes but can support the following:

- Adding new columns.
- Deleting existing columns.
- Altering existing columns (type, size, nullability).

Both sink types handle delete events as you would expect, using a row-scoped delete statement with an extra offset condition. To handle create, read and update events, the sinks use an “upsert” statement, an example statement can be seen in Listing 16. This statement inserts the new values if the key doesn’t exist. Otherwise, the values will be updated, provided that the last

offset is older than the new offset.

```
INSERT INTO cdc_public.newtable1 (position,name,...)
VALUES (3,"three",...)
ON DUPLICATE KEY UPDATE
position = IF(VALUES(cdc_last_updated) > cdc_last_updated,
              VALUES(position), position),
name = IF(VALUES(cdc_last_updated) > cdc_last_updated,
          VALUES(name), name),
...
```

Listing 16: Example Upsert SQL.

Transactional Sink

The transactional sink aims to apply the changes to the destination in transactional order. To maintain this order across all tables, all change events must be sent to a single Kafka topic, using the `SINGLE` topic strategy. If the system was to use a `PER_TABLE` strategy, then a global ordering could not be enforced between the different schema-based consumers. For this reason, the application will validate that a user is not using an invalid combination of the two configuration options during pipeline startup.

When delivering the changes, the sink utilises regular JDBC `Statement` batching, enabling the JDBC property `rewriteBatchedStatements` for some added performance. The transactional sink will iterate through the current batch of ordered events. If the current event indicates a schema change, the SQL batch is executed early and the table alteration is made. Then, the SQL batch will be extended/started by the new statement which corresponds to the data change seen in the current event. Each SQL batch will be executed when it reaches the maximum size of 1000 statements or if the iteration cycle completes.

Initially, this sink was implemented by constructing a large string of concatenated SQL statements. This approach did not perform well. The implementation of the batching sink brought the more efficient JDBC `Statement` batching approach to light. However, due to the nature of the transactional sink, the performance gains to be had were not as great as the batching sink. The transactional sink cannot gain the same performance benefits due to the variability of change events when processing in a transactional order.

Batching Sink

The batching sink provides higher performance at the cost of losing a global change order between tables. The sink groups together change events by table, then batches them together using a more performant JDBC `PreparedStatement` batching approach. This causes the destination to be periodically in an inconsistent state, as changes are no longer being applied in transactional order between all tables, instead opting for transactional order within each table. Although the sink doesn't provide a consistent state throughout, it's worth noting that it provides an eventually consistent state. If the system applies all available changes across all topics, the destination will be in a replicated state consistent with the source. Unfortunately, the performance gain of the sink is dampened if the source tends to produce lots of delete events or has regular schema changes, as these cannot be batched together with the prepared "upsert" statements.

First, the batching sink groups the change events by table and begins to iterate through the table groupings. At the start of each iteration, the current table's change events are divided into batches. These batches indicate groupings of change events which can be batched together to apply their operation as described in the previous paragraph. Delete events cannot be batched with other operations, so they are contained as a sole member of a batch. Additionally, all members of a batch must contain the same table structure in their "after" block, as **ALTER TABLE** statements cannot be intertwined with the batching process. Finally, each batch may contain up to 1000 change events.

Once the table-based batches have been constructed, the application begins to iterate over them. If a batch contains a single delete event, it is executed via a regular JDBC `Statement`. However, if a batch is made up of other operations, the application will check for a schema change using the first change event in the batch. If a schema change is required, an **ALTER TABLE** statement is executed. Then, the application begins to build the `PreparedStatement`. Iterating through the entries of the batch, the values of each "after" block are batched together and executed/committed on the destination database.

Finally, when using the batching sink a pipeline will only pass validation provided that the `PER_TABLE` topic strategy is being used. This decision was made as the reason a user would choose to use this type of sink would be for its performance benefits. Using the `SINGLE` topic strategy with multiple schemas would likely result in a less performant consumer throughput, so the developer decided to restrict the user. Upon reflection, the application should allow for both types of topic strategy, as the sink type doesn't strictly require either of them, unlike the

transactional sink.

4.4.3 Metrics

The application tracks various metrics using a `@Service` class named `MetricsService`. As the application runs a user's pipeline, it will call out to the metrics service to inform when a status change is required, or when an event is being produced/consumed. Each of these interactions will have an effect on the pipeline's metrics.

A user can obtain metrics about their pipeline via three API endpoints:

1. `GET /pipeline/status`
2. `GET /pipeline/metrics/snapshot`
3. `GET /pipeline/metrics`

The first endpoint retrieves the current status of the pipeline. The value changes in the following manner: `NOT_RUNNING` → `STARTING` → `SNAPSHOTTING` → `STREAMING` and will return to `NOT_RUNNING` upon unexpected failure or a halt action. The `STARTING` status refers to the period of time where the pipeline is creating the consumers, see Section 4.4.2.

The second endpoint provides metrics about the initial snapshot, these values include:

- Snapshot Complete?
- Duration (seconds)
- List of table scoped metrics: rows snapshot and completed?

This set of metrics allows the user to view the current progress of their snapshot or view the details of a completed one. To avoid updating the table scoped data after each processed row, the `Snapshotter` updates these values every 500ms using the `updateSnapshotRows()` method. An example snapshot metrics response can be seen in Listing 17.

Finally, the user can get a set of general metrics using the third endpoint. This endpoint allows the user to get an overall view of the performance of their pipeline, allowing for bottlenecks to be identified. The following values are provided:

```
{
  "completed": true,
  "durationSeconds": 88,
  "tables": [
    {
      "table": {
        "schema": "public",
        "table": "towns"
      },
      "rows": 10000000,
      "completed": true
    },
    {
      "table": {
        "schema": "public",
        "table": "newtable2"
      },
      "rows": 4,
      "completed": true
    },
    {
      "table": {
        "schema": "public",
        "table": "newtable1"
      },
      "rows": 3,
      "completed": true
    }
  ]
}
```

Listing 17: Get Snapshot Metrics Response.

- Pipeline start time (epoch ms)
- Time lag: source ↔ producer (ms)
- Time lag: producer ↔ consumer (ms)
- Number of transactions
- Total number of events produced
- Total number of events consumed
- Produced number of create, read, update and delete events per table

The two most important metrics are the time lag values, allowing a user to see just how real-time the system really is. The time-lag values provide insight into the end-to-end journey of a change event and how much time has passed between each component. The application tends to have an impressive source to producer time lag but will struggle to provide the same performance between the producer and consumer when the source is producing events at a higher rate than the consumer throughput. An example set of these general metrics can be seen in Listing 18.

4.4.4 Pipeline Resiliency

A CDC pipeline is often mission critical for an organisation, so it was important to implement systems to negate the impact of an unexpected failure. Ideally, this would mean that a pipeline could fail during the streaming phase and restart at the point of failure, without the need for a new snapshot or the processing of already processed change events. However, due to the time restrictions of this project, the developer was not able to implement a mechanism of this nature.

Instead, the developer opted to provide optional email alerts for unexpected pipeline failures. When a pipeline fails unexpectedly, it calls out to the `EmailHandler` class. The `sendEmail()` method builds a `MimeMessage` instance which is then sent by the static `Transport.send()` method, see Listing 19. The content of the email will inform the recipients of the pipeline failure, providing the cause of the failure by concatenating the exception's message and it's causes.

When a pipeline stops, it is important to clean up any resources used by the pipeline, ideally returning to the same state as before the pipeline was initially ran. This includes resetting the metrics service and cleaning up the consumer threads. However, the application doesn't clean

```
{
  "pipelineStartTime": 1711620111347,
  "dbProducerTimeLagMs": 3,
  "producerConsumerTimeLagMs": 29,
  "numOfTransactions": 7,
  "totalProduced": 17,
  "totalConsumed": 17,
  "tables": [
    {
      "table": {
        "schema": "public",
        "table": "newtable2"
      },
      "operationCounts": {
        "create": 3,
        "read": 4,
        "update": 1,
        "delete": 3
      }
    },
    {
      "table": {
        "schema": "public",
        "table": "newtable1"
      },
      "operationCounts": {
        "create": 3,
        "read": 3,
        "update": 0,
        "delete": 0
      }
    }
  ]
}
```

Listing 18: Get Metrics Response.

```
1 public void sendEmail(Exception triggerEx) {
2     log.info("Attempting to send failure notification emails.");
3     try {
4         var message = new MimeMessage(session);
5         message.setFrom(new InternetAddress(senderEmail));
6         for (String recipient : recipientList) {
7             message.addRecipient(Message.RecipientType.TO, new
                ↪ InternetAddress(recipient));
8         }
9         message.setSubject("Change Data Capture Pipeline Failure");
10        message.setText(getEmailContent(triggerEx));
11        Transport.send(message);
12        log.info("Pipeline failure notification email(s) sent successfully.");
13    } catch (Exception e) {
14        log.error("Failed to send failure notification emails.", e);
15    }
16 }
```

Listing 19: Send Email Code.

up the publication, replication slot or topics. This decision was made as when the previously discussed mechanism gets implemented, the application will likely reuse these resources. Additionally, the developer wanted to avoid deleting any resources from a user's environment. It's safer to ask a user to complete this action, than to let the application attempt it and delete a resource being used elsewhere by the user.

4.4.5 Other Limitations

Throughout the development of the application, various limitations had to be enforced in order to keep the main deliverable of the project on track. One of these limitations is that the application only supports the most primitive column types. These types include various sizes of integers, varchars, floats/doubles and boolean values. This subset of data types can be extended relatively easily but as the developer had limited development time, it was invested into more important aspects of the system.

Additionally, the user is limited to using the hard-coded values for some important configuration options e.g. snapshot fetch size and the maximum number of change events per Kafka topic poll. Optimum values may differ across environments, so it would have been best to provide the user with as much flexibility as possible, while still ensuring the system works as

intended.

Another key limitation is that only a single pipeline can run at a time. Allowing multiple pipelines at once would require an advanced pipeline management system to track and maintain the different pipelines, ensuring each individual pipeline is run in isolation. This feature would require a lot of development effort but wouldn't contribute much to the aims of the project.

Chapter 5

Testing

5.1 Unit and End-to-End Tests

Manual testing of the system quickly became infeasible as the codebase grew. To overcome the slow process of testing new features, a series of unit and end to end tests were written to confirm the behaviour of new functionality and ensure that no unexpected changes to existing behaviour were made. These tests were written using JUnit, a testing framework used primarily for unit testing. Additionally, the end to end tests utilised Testcontainers, spinning up PostgreSQL, MySQL and Kafka containers to test the complete workflow of a pipeline. Ultimately, 55 tests were written to provide an overall code coverage of 86%, which gave added confidence in the functionality of the system.

5.2 Performance Tests

The performance of a CDC system is incredibly important. If an application cannot provide a high enough end-to-end throughput, then an organisation may have no choice but to find a different solution which can handle the activity within their source.

To test the snapshot performance, a local instance of the application was ran via the provided docker script. The application ran a pipeline which was configured to use a local instance of PostgreSQL with no destination, using the `SINGLE` topic strategy. The pipeline snapshot 80 million rows across 4 tables (13.7 GB) in 655 seconds. This gives the application a snapshot throughput of 122,137 rows/sec.

A similar setup was used to test the applications streaming performance. The application was

ran using the provided docker script and the pipeline configuration was generally the same, only differing in the captured tables. Instead of the pipeline capturing an existing set of 80 million rows, the pipeline streamed 20 million rows from a single table (3.4 GB). These rows were generated using the SQL statement seen in Listing 20, which committed transactions of 5000 insert operations until 20 million rows were inserted. The pipeline processed this data in 332 seconds, giving a throughput of 60,240 rows/sec. However, this is not the streaming throughput upper limit. The pipeline managed to keep up with the looping SQL statement, maintaining a consistent source-producer time lag of less than 70 ms. The demonstrated performance is acceptable and further testing is not necessary unless streaming throughput becomes a bottleneck of the system.

```
DO $$
DECLARE
    total_records INT := 20000000;
    batch_size INT := 5000;
    inserted_records INT := 0;
BEGIN
    WHILE inserted_records < total_records LOOP
        BEGIN
            BEGIN
                INSERT INTO towns2 (code, article, name, department)
                SELECT
                    left(md5((inserted_records + i)::text), 10),
                    md5(random()::text),
                    md5(random()::text),
                    left(md5(random()::text), 4)
                FROM generate_series(1, batch_size) s(i);
            COMMIT;
            inserted_records := inserted_records + batch_size;
        END;
    END LOOP;
END $$;
```

Listing 20: SQL Statement to Generate Streamed Data.

To measure the consumers maximum throughput, a pipeline processed 10 million snapshot events (1.7 GB) from a single table using each of the sinks. First, the transactional sink applied the 10 million events in 1,433 seconds, processing them at a rate of 6,978 rows/sec. Whereas, the batching sink processed them in 655 seconds, achieving an average throughput of 15,267

rows/sec. The batching sink performed 2.18x better than the transactional sink in these optimal conditions.

Finally, to get a more realistic consumer throughput measure, the pipeline processed 10 million streamed events (1.7 GB), spread across 4 tables. A single row was inserted into each table, cycling around the 4 and committing every 1250 cycles, until 10 million new rows were created. The transactional sink managed to apply these events to the destination in 1,622 seconds, with a throughput of 6,165 rows/sec. However, the batching sink completed the same operation in 659 seconds, showcasing a throughput of 15,175 rows/sec. The batching sink performed 2.46x better than the transactional sink. This scenario highlights the disparity in performance between the two sinks when interweaving operations on different tables. When using the batching sink, the inter-weaved change events will be processed in a very similar manner to that of the best-case scenario test. Whereas, the transactional sink has to maintain the order of the individual operations, essentially applying each event one after another.

In conclusion, the performance of an end-to-end pipeline using an example MySQL sink is acceptable for a change data capture solution, provided that the source's operational throughput doesn't exceed that of the chosen sink. The two sinks are clear bottlenecks within the system, as they cannot reach the same level of performance achieved by the producer. It is suspected that the system would likely be able to achieve a much higher throughput when applying changes to a dedicated data warehouse e.g Snowflake. However, the producer showcases a great level of performance. Due to the optional nature of the consumer, the user can utilise the application's producer and build their own consumer if they require a higher throughput.

Chapter 6

Conclusion

The project was a great learning experience and resulted in a useful piece of software that performs at an impressive level. In this final chapter, we will discuss the key achievements of the project, along with some areas in which the application failed to meet expectations. Finally, the author will discuss some potential further work to build upon the research and development completed throughout the project.

6.1 Achievements

Before initiating the research phase of the project, four core objectives were set with the intention of measuring the project's success, see Section 1.2. The application met the majority of these objectives, achieving three out of the four.

The first two objectives define the core behaviour of the delivered application, while exceeding expectations by delivering two example sinks. In its current state, a user can define an end-to-end replication pipeline which produces and consumes source agnostic change events in near real-time. Despite the performance of the two example sinks being much less than the consumer, the application can handle relatively active sources without lagging behind, provided that the source's operational throughput does not exceed that of the sink's throughput.

Next, the system achieved the third objective by exposing a wide range of metrics about a user's pipeline. These metrics offer a fine-grained view over all aspects of the pipeline, providing a deeper insight into the overall process than that provided by other solutions. This enables the user to infer the correctness of their pipeline and view its current state. However, one limitation of the current implementation is the lack of a historical view. This could be collated through

regular requests but would be better served as a built-in feature.

Despite not being a core objective, the system was built to be very easy-to-use. The application provides a quick-start environment, providing an example postman collection and `docker-compose.yml` file. This enables the user to get familiar with the system before connecting to their actual environment, easing the on-boarding process. This is important as the on-boarding process of a CDC solution can often come with a lot of friction. Being able to showcase the system and its functionality quickly should mitigate the majority of early concerns from a potential user.

Finally, the application is highly flexible and extensible. First, its flexibility is shown through the various configuration options and optional consumer. A user can choose to forego the use of a provided example sink, in favour of building their own internal or external sink. This situation is very likely as the current system only supports one source (PostgreSQL) and one destination (MySQL). To accommodate for new internal connectors, the system was built to be highly extensible through abstract principles.

6.2 Potential Improvements

Despite achieving the majority of the the core objectives, there are some improvements which could be made to existing functionality.

Most importantly, the application failed to achieve the final core objective. The current system cannot resume a pipeline at the point of failure, meaning a user would have to restart the replication process and re-snapshot their source. Depending on the number of rows, this process could take days, meaning the missing feature would likely be a deal breaker in time sensitive use cases. Before any future work gets implemented into the system, this core issue should be resolved.

The current system doesn't allow for the configuration of many important values within a pipeline. The system could be improved by adding further configuration to values like the snapshot fetch size and the Kafka poll size, as the optimal values may differ between user environments. This would be straightforward to implement but wasn't a priority during development.

Finally, future developers could add support for new source/destination technologies. This would greatly increase the number of users that can actually use the application.

6.3 What Went Wrong?

One mistake that lost a great deal of development time was the initial attempt at utilising the Avro format for the application's change events. The developer failed to step back and evaluate other options when it wasn't working, resulting in about a month of development time being lost to the issue. This time could have been much better spent on a core feature like the missing resilience feature discussed earlier.

Additionally, although the two MySQL sinks showcased that replication was possible using the application, the performance was not at the same level as the producer. Upon reflection, I feel as though it would be more valuable to demonstrate a highly performant sink rather than use a familiar technology.

6.4 Further Work

The current application provides a solid foundation for a change data capture solution. However, there is some potential further work to expand upon the project.

The implementation of a user interface could further improve the user experience. For example, the pipeline creation experience would be greatly improved as the current API request isn't very intuitive and requires additional documentation. Furthermore, the metrics system could be enhanced to provide historical views via a graphical representation. This was out-of-scope for the initial development of the project, as the focus was on the core replication logic.

To further expand upon this user interface, a centralized platform could be created which could manage various instances of the current application via a dashboard view. The platform could even handle the creation/hosting of new application instances, foregoing the need for users to manage the required containers. This platform would allow for improvements to existing features of the system. The email notification system could be greatly improved by using an email specified by the platform to send the notifications, rather than require a user defined one. This transition would start to convert the system into more of a product, rather than an open-source free-to-use application.

Bibliography

- [1] O. Abdo. The docker's impact: Transforming the landscape of application development and deployment. <https://blog.stackademic.com/the-dockers-impact-transforming-the-landscape-of-application-development-and-deployment-7116d4c86bee>. Accessed: March 21, 2024.
- [2] A. Andreakis and I. Papapanagiotou. Dblog: A watermark based change-data-capture framework, 2020.
- [3] Apache Software Foundation. Apache kafka. <https://kafka.apache.org/>. Accessed: March 19, 2024.
- [4] Baeldung. Introduction to debezium. <https://www.baeldung.com/debezium-intro>. Accessed: March 14, 2024.
- [5] T. Barrera. Understanding change data capture (cdc): Definition, methods and benefits. <https://airbyte.com/blog/change-data-capture-definition-methods-and-benefits>. Accessed: March 14, 2024.
- [6] T. Collins. What is maven in java? (framework and uses). <https://www.browserstack.com/guide/what-is-maven-in-java>. Accessed: March 21, 2024.
- [7] Confluent. Survey reveals apache kafka® will be mission-critical to 90 percent of data and application infrastructures in 2018. <https://www.confluent.io/press-release/survey-reveals-apache-kafka-will-be-mission-critical-to-90-percent-of-data-and-application-infrastructures-in-2018/>. Accessed: March 19, 2024.
- [8] Confluent. Why change data capture? <https://www.confluent.io/en-gb/learn/change-data-capture/>. Accessed: March 13, 2024.
- [9] davyam. pgeasyreplication. <https://github.com/davyam/pgEasyReplication>. Accessed: March 26, 2024.

- [10] Debezium. Debezium. <https://github.com/debezium/debezium>. Accessed: March 26, 2024.
- [11] Debezium. Debezium architecture. <https://debezium.io/documentation/reference/2.5/architecture.html>. Accessed: March 14, 2024.
- [12] Debezium. Debezium connector for postgresql. <https://debezium.io/documentation/reference/2.5/connectors/postgresql.html>. Accessed: March 14, 2024.
- [13] Debezium. Debezium connectors. <https://debezium.io/documentation/reference/2.5/connectors/index.html>. Accessed: March 14, 2024.
- [14] S. H. A. El-Sappagh, A. M. A. Hendawi, and A. H. El Bastawissy. A proposed model for data warehouse etl processes. *Journal of King Saud University - Computer and Information Sciences*, 23(2):91–104, 2011.
- [15] Fivetran. Change data capture: Definition, benefits, and how to use it. <https://www.fivetran.com/blog/change-data-capture-what-it-is-and-how-to-use-it>. Accessed: March 13, 2024.
- [16] Google Cloud. Sources. <https://cloud.google.com/datastream/docs/sources>. Accessed: March 18, 2024.
- [17] Google Cloud Tech. Introduction to datastream for bigquery. <https://www.youtube.com/watch?v=vMo6Zgkvt40>. Accessed: March 18, 2024.
- [18] IBM. What is a data warehouse? <https://www.ibm.com/topics/data-warehouse>. Accessed: February 16, 2024.
- [19] E. Levy. Debezium for cdc: Benefits and pitfalls. <https://www.upsolver.com/blog/debezium-vs-upsolver>. Accessed: March 14, 2024.
- [20] S. Maarek. Introduction to schemas in apache kafka with the confluent schema registry. <https://medium.com/@stephane.maarek/introduction-to-schemas-in-apache-kafka-with-the-confluent-schema-registry-3bf55e401321>. Accessed: March 19, 2024.
- [21] Matillion. Matillion change data capture (cdc). <https://docs.matillion.com/data-productivity-cloud/cdc/docs/31111/#core-concepts>. Accessed: March 14, 2024.

- [22] Matillion. Postgresql connector. <https://docs.matillion.com/data-productivity-cloud/cdc/docs/31624/>. Accessed: March 14, 2024.
- [23] Matillion. What is change data capture and why is it important? <https://www.matillion.com/blog/what-is-change-data-capture-and-why-is-it-important>. Accessed: February 16, 2024.
- [24] P. S. S. D. Mayuri B. Bokade and P. H. R. Vyavahare. Framework of change data capture and real time data warehouse. *International Journal of Engineering Research & Technology (IJERT)*, 02(04), 04 2013.
- [25] H. Nassour. Spring boot vs spring : Discovering the magic of spring boot. <https://medium.com/javarevisited/discovering-the-magic-of-spring-boot-a-before-after-story-293ee5b3be7f>. Accessed: March 22, 2024.
- [26] A. Pandey and S. Mishra. Moving from traditional data warehouse to enterprise data management: A case study. *Issues in Information Systems*, 15:133–140, 01 2014.
- [27] PostgreSQL. Create publication. <https://www.postgresql.org/docs/current/sql-createpublication.html>. Accessed: March 26, 2024.
- [28] PostgreSQL. Issuing a query and processing the result. <https://jdbc.postgresql.org/documentation/query/>. Accessed: March 26, 2024.
- [29] A. Prakash. What is avro?: Big data file format guide. <https://airbyte.com/data-engineering-resources/what-is-avro>. Accessed: March 19, 2024.
- [30] Spring. Spring initializr. <https://start.spring.io/>. Accessed: April 16, 2024.
- [31] Stack Overflow. 2023 developer survey. <https://survey.stackoverflow.co/2023/>. Accessed: March 20, 2024.