Analysis of Usage and Design of Change Data Capture for Implementing Multi-Master on RDBMS During Blue-Green Deployment

Prepared as a requirement for bachelor's degree graduation

By

RAHMAT WIBOWO

NIM: 18219040

INFORMATION SYSTEMS AND TECHNOLOGY STUDY PROGRAM

SCHOOL OF ELECTRICAL ENGINEERING & INFORMATICS

BANDUNG INSTITUTE OF TECHNOLOGY

202

STATEMENT SHEET

I hereby declare that:

1. The work and writing of this Final Assignment Report was carried out without the use of unauthorized assistance.
2. All forms of citations and references to other people's writings used in the preparation of this final assignment report have been written properly and correctly.
3. This Final Assignment Report has never been submitted to any educational program at any university.

If proven to have violated the matters above, I am willing to be subject to sanctions in accordance with the Academic and Student Affairs Regulations of the Bandung Institute of Technology in the Enforcement of Academic and Student Affairs Norms section, especially Article 2.1 and Article 2.2.

Bandung, September 17, 2023

Rahmat Wibowo
NIM 18219040

FOREWORD

Praise and gratitude be to God Almighty, for His blessings and mercy the author was able to carry out the final assignment as a graduation requirement for the undergraduate level in the Information Systems and Technology study program.

The author does not forget to thank the parties who have helped the author during the work on this final assignment, namely:

1) Mrs. Tricya Esterina Widagdo, S.T, M.Sc., as Final Assignment Supervisor for the knowledge, guidance, criticism and suggestions provided during the final assignment process.

2) Mr. Achmad Imam Kistijantoro, S.T, M.Sc., Ph.D, as Examiner Lecturer for Final Assignment Session, and Mrs. Dr. Yani Widyani, S.T, M.T. as Examining Lecturer II for the Final Assignment Session for his criticism and suggestions in working on the final assignment.

3) All lecturers in the Information Systems and Technology Study Program at the Bandung Institute of Technology for the knowledge and learning experiences they have provided.

4) The writer's family always supports the writer during the study period until writing the final assignment.

5) The writer's friends who always provided support to the writer during the study period.

6) Xendit colleagues who have taught us the use of Terraform to make tool testing easier

7) Amazon Web Services colleagues credit for testing the tool in an ideal environment

ABSTRACT

ANALYSIS OF USE AND DESIGN OF CHANGE DATA CAPTURE TO PERFORM MULTI-MASTER ON RDBMS DURING BLUE-GREEN DEPLOYMENT

By

Rahmat Wibowo

NIM: 18219040

Relational databases are services that are difficult to change because they are stateful. When the application is released with the latest version, with changes to the database schema, the consistency of the relational database is difficult to maintain if the blue-green deployment method is adopted, which requires the database to have two identical environments and must be in Multi Master condition. Blue-green deployment is a deployment strategy creating two separate but identical environments. One environment (blue) runs the current version of the application and one environment (green) runs the new version of the application. Once testing is complete in the green environment, application traffic is immediately redirected to the green environment and the blue environment is no longer used. Multi-master replication means there is more than one node acting as the master node. Change Data Capture (CDC) is a method for integrating data based on the results of identification, capture and delivery only for data changes in operational systems. Therefore, this final project aims to create tools that help implement this. Tools is designed with an event-based architecture by taking every database change data in the queue that has been captured by Debezium and changing the data structure according to the configuration and executing queries to synchronize. The tools designed were successful in helping the blue-green deployment process. The tools were tested by deploying in a Kubernetes environment and using Terraform as Infrastructure as a Code. In the test results, the average replication lag produced was 563 ms and the 99th percentile was 948 ms. This makes the tool successfully synchronize data in less than 1 second on 99% of requests.

Keywords: Kubernetes, Change Data Capture, Blue-Green Deployment

LIST OF CONTENTS

VALIDITY SHEET.....i

FOREWORD.....iii

ABSTRACT.....iv

LIST OF CONTENTS.....in

LIST OF FIGURES.....viii

LIST OF TABLES.....x

CHAPTER I Introduction.....1

I.1 Background 1

I.2 Problem Formulation 2

I.3 Objective 2

I.4 Limitation of the Problem 2

I.5 Methodology 3

I.6 Systematic Discussion 4

CHAPTER II LITERATURE STUDY.....5

II.1 Relational Databases 5

II.1.1 Database Schema 5

II.1.2 Database Schema Migration 6

II.1.3 Database Replication 8

II.2 Deployment Application 10

II.2.1 Blue-Green Deployment 10

II.3 Change Data Capture 12

II.3.1 Debezium 13

II.4 Write-Ahead Logging (WAL) 14

II.5 Logical Replication 14

II.6 Stream Processing 15
II.7 Infrastructure as a Code 16
II.8 Kubernetes 17
II.8.1 Primitives on Kubernetes 17
II.9 Schema Migration Using Expand-Contract Pattern 19
CHAPTER III PROBLEM ANALYSIS AND SOLUTION DESIGN..... 21
III.1 Problem Description and Analysis 21
III.2 Solution Analysis 21
III.2.1 Blue-Green Deployment Process on Databases 21
III.2.2 Module Plan 23
III.2.3 Solution Analysis Change Data Capture 25
CHAPTER IV..... IMPLEMENTATION AND TESTING 26
IV.1 System Requirements Analysis 26
IV.2 General Description of the System 26
IV.3 System Architecture Design 28
IV.4 Tools Architectural Design 30
IV.5 Implementation of Tools 35
IV.5.1 Configuration File Format 35
IV.6 Development Environment 47
IV.7 Testing Environment 48
IV.8 Testing Scenarios 50
IV.8.1 Blue-green Deployment Testing 50
IV.8.2 Tools Functional Testing 55
IV.8.3 Tool Performance Testing 65

CHAPTER V..... CONCLUSIONS AND RECOMMENDATIONS 70

V.1 Conclusion 70

V.2 Suggestion 71

BIBLIOGRAPHY..... 73

LIST OF FIGURES

Figure II.1Master-slave replication (PostgreSQL, 2022) 9
Figure II.2Multi-master replication (PostgreSQL, 2022) 10
Figure II.3 IllustrationBlue-Green Deployment 11
Figure II.4 CDC method forimmediate data extraction (Ponniah, 2010) 13
Figure II.5 Simplification ArchitectureLogical Replication 15
Picture III.1Activity Diagram Blue-Green Deployment On Database 22
Figure III.2 Tool Module Design in Changes to the Preparation SchemeBlue-Green Deployment 23
Picture III.3 Plan of the Appliance ModuleData Change Event PreparationBlue-Green Deployment 23
Picture III.4 Plan of the Appliance ModuleData Change Event Blue-Green Deployment 24
Picture IV.1Sequence Diagram Rooster 28
Figure IV.2 Tools Architectural Design 29
Figure IV.3 Detail Modul ddl_change_executor 31
Figure IV.4 Details of the config module 32
Figure IV.5 CDC Module Details 32
Figure IV.6 Details of the data_transformer module 33
Figure IV.7 Detail Modul SQL Builder 33
Figure IV.8 Detail Module SQL Executor 34
Figure IV.9 Details of the report_generator module 35
Figure IV.10 Tool Configuration File Format 37
Figure IV.11 Debezium Configuration File 39
Figure IV.12 CLI Interface Tools 40
Figure IV.13 Diagram Function Call Modul ddl_change_executor 41
Figure IV.14 Diagram Function Call Modul config 43
Figure IV.15 Function Call Diagram in the CDC Module 44
Figure IV.16 Function Call Diagram in the data_transformer Module 45
Figure IV.17 Diagram Function Call Pada Modul query_builder 46

Figure IV.18 Diagram Function Call Pada Modul query_executor 47
Figure IV.19 Solution Architecture in the Testing Process 48
Figure IV.20 Security Group Test Environment 49
Figure IV.21 Kubernetes Resources in Testing Scenarios 51
Figure IV.22 Test Environment Terraform Folder Structure 53
Figure IV.23 Cost Analysis of Testing Environment Requirements 54
Figure IV.24 Data Synchronization Status Results in the Database at the EndBlue-Green Deployment 55
Figure IV.25 Tool Performance Testing Environment 66

LIST OF TABLES

Table II.1 List of Relational Database Schema Entities (Delplanque et al., 2018) 6
Table II.2 List of Change Operations for Relational Database Schemes (Dijkstra, 2021) 7
Table II.3 List of Denormalization Change Operations (Dijkstra, 2021) 8
Table III.1 Comparison of MethodsChange Data Capture 25
Table IV.1 System Functional Requirements 26
Table IV.2 Mapping of Implemented Modules Against the Design 30
Table IV.3 Configuration File Description 37
Table IV.4 List of Tool Commands 40
Table IV.5 Description of the ddl_change_executor Module Function 41
Table IV.6 Description of Functions in the config Module 43
Table IV.7 Description of Functions in the CDC Module 44
Table IV.8 Description of Functions in the data_transformer Module 45
Table IV.9 Description of Functions in the query_builder Module 46
Table IV.10 Description of Functions in the sql_executor Module 47
Table IV.11 Tool Development Environment 47
Table IV.12 Tool Testing Environment 48
Table IV.13 Description of Kubernetes Resources Used 51
Table IV.14 List of Change Operation Tests 56
Table IV.15 Table of Rename Column Test Results When the Service Uses the Old Version 57
Table IV.16. Drop Table Test Results Table When the Service Uses an Old Version 57
Table IV.17. Testing Add Column When the Service Uses an Old Version 58
Table IV.18. Delete Test Using Primary Key When Service Uses Old Version 58
Table IV.19. Drop Column Test Results When the Service Uses an Old Version 58
Table IV.20 Table of Rename Table Test Results When the Service Uses an Old Version 59
Table IV.21 Horizontal Splitting Test Results When the Service Uses an Old Version 59

Table IV.22 Vertical Splitting Test Results When the Service Uses the Old Version 60
Table IV.23 Test Results for Adding Redundant Tables When the Service Uses an Old Version 61
Table IV.24. Table of Complex Case Testing Results in the Old Version 61
Table IV.25 Table of Rename Column Test Results When the Service Uses a New Version 62
Table IV.26. Add Column Testing When the Service Uses a New Version 63
Table IV.27 Rename Table Test Results Table When the Service Uses a New Version 63
Table IV.28 Horizontal Splitting Test Results When the Service Uses a New Version 64
Table IV.29 Vertical Splitting Test Results When the Service Uses a New Version 64
Table IV.30. Table of Complex Case Test Results When the Service Uses a New Version 65
Table IV.31 Tool Performance Test Results with Scheme Changes 67
Table IV.32 Tool Performance Test Results Without Schematic Changes 68
Table IV.33 Results of PostgreSQL Logical Replication Performance Testing 68

INTRODUCTION

I.1 Background

Relational databases are the most difficult services to change within Three-tier architecture, because the database which is the data layer is the only layer that is stateful. Database consistency is very important in application architecture. When the app is release with the latest version, with schema changes from the regional database, data consistency is difficult to maintain if using Blue-Green Deployment..

Blue-green deployment is deployment strategy which creates two separate but identical environments. One environment (blue) runs the current version of the application and one environment (green) runs the new version of the application. Using a blue/green deployment strategy increases application availability and reduces deployment risk by simplifying the rollback process if deployment fails. Once testing is complete in the green environment, application traffic is immediately redirected to the green environment and the blue environment is no longer used. On blue-green deployment if there is changes in data definition language (DDL) in an RDBMS ideally requires two databases that run simultaneously and both have the ability to store data, but the data must be maintained with each other.

This is intended if the DDL change results in a decrease in application performance, such as removing indexes on tables which aims to increase storage capacity efficiency for making queries running in the environment.production slow down and cpu utilization from the database becomes high. The database used can be: rollback returns to its original state without downtime resulting from changes returning to the original condition.

In addition, large companies use relational managed databases service on cloud computing, such as on Amazon Web Services there is Amazon RDS and on Google Cloud there is Cloud SQL.

I.2 Problem Statement

This final assignment contains the following problem statement:

1. How to implement blue-green deployment on a relational database?
2. What is the architecture of the implementation tool for blue-green deployment?
3. How does the implementation tool perform?
4. Can the tools be implemented in the production environment?

I.3 Objective

Until the time this research was written, no tool had been found that could synchronize the database during schema migration. Therefore, this final assignment aims to develop tools for doing blue-green deployment by having two versions of the database running simultaneously and minimizing the existence downtime when changing the database schema.

I.4 Limitation of the Problem

This final assignment has several problem limitations as follows.

1. The database used in design and implementation is PostgreSQL.

I.5 Methodology

The stages taken in preparing this Final Assignment are:

1. Problem Analysis

   Collection of problems that exist in blue-green deployment implementation changing the schema in the database and carrying out analysis to find solutions to these problems. This problem analysis includes the problems faced when making schema changes to the database.

2. Solution Design

Designing the architecture that will be used in the database schema migration process. The architecture will cover all aspects necessary to synchronize data across two different database versions in the blue-green deployment.

3. Solution Implementation

   Creation of tools that will be used in database processes in PostgreSQL in a cloud computing environment. Tools is a CLI-based application that has several parameters that can be set using a configuration file. The tool will process data changes via WAL Log and carry out the process of entering processed data into databases that have different versions and ensure data consistency.

4. Evaluation

   Carrying out tests on tools using benchmarking tools as a burden maker /load generator to simulate data changes in the database. There will be metrics that will be calculated such as the number of failed transactions and replication lag.

I.6 Systematic Discussion

The systematic discussion in this final research assignment consists of five chapters. The five chapters are Chapter I Introduction, Chapter II Literature Study, Chapter III Analysis and Design of Solutions, Chapter IV Implementation and Evaluation, and Chapter V Conclusions and Suggestions.

Chapter I Introduction discusses the background of the problem that will be the topic of the final assignment as well as the basis of the problem that will be raised including the problem, objectives, problem limitations, methodology and systematic discussion.

Chapter II Literature Study contains the theoretical basis used for implementing the final assignment. This chapter contains various book and journal references discussing relational databases, software architecture,deployment aplikasi, Write Ahead Log, logical replication, streaming processing, and related research.

Chapter III Problem Analysis and Solution Design contains a presentation of the architectural design and design of the system to be created. This chapter contains descriptions and analysis of problems, analysis of solutions.

Chapter IV Implementation and Evaluation contains the results of implementation and evaluation of the designed tools. This chapter contains descriptions and analysis of problems, analysis of solutions, andtimeline completion of the final assignment.

Chapter V Conclusions and Suggestions contains conclusions based on the results of testing and implementation of tools. In addition, this chapter will discuss the potential for further tool development.

LITERATURE REVIEW

I.7 Relational Databases

According to (Silberschatz et al., 2019), a database management system (DBMS) is a collection of interrelated data and a set of programs to access that data. A collection of data, usually referred to as a database, contains information relevant to a company. The main purpose of a DBMS is to provide a way to store and retrieve information from a database easily and efficiently.

Database systems are designed to manage large amounts of information. Data management involves both defining structures for storing information and providing mechanisms for manipulating information. In addition, the database system must ensure the security of the information stored, although the system goes down or there is an unauthorized access attempt. If data is to be shared among multiple users, the system must avoid the possibility of anomalous results.

A relational database is a type of database that uses tables to represent data and the relationship between these data. In addition, relational databases have a series of rules for data that are defined for each table.

I.7.1 Database Schema

The database schema is used to define the architecture of the database. Schemas are used to organize how the structure of data is created. In a relational database. A schema defines tables, other related database objects, and their relationships. Database schemas are defined through formal languages such as SQL. In a relational database the schema is defined inData Definition Language (DDL). In Table II.1, there is a list of several entities in the relational database schema, especially in PostgresSQL based on (Delplanque et al., 2018).

Table II.1 List of Relational Database Schema Entities (Delplanque et al., 2018)

No.	Entity Type	Description
1	Table	A collection of related data and consists of columns and rows.
2	Column	A collection of data values of a specific type on each row on the database.
3	Primary key constraint	Constraint to uniquely identify each record in the table.
4	Foreign key constraint	Constraint to identify relationships between database tables by referencing a column in the containing foreign key, to primary key in the parent table.
5	Unique constraint	Constraint to uniquely identify each record in the table.
6	Check constraint	Constraint to limit the range of values that can be placed in a column.
7	Default value constraint	Constraint to fill columns with default and fixed values.
8	View	Virtual tables are created by real tables from the database.
9	Trigger	Procedural code that executes automatically in response to certain events on certain tables or views in a database.

I.7.2 Database Schema Migration

Database schema migration is the process of changing the database schema, either adding column in the table, change the table name, drop in the table. This process is called change operations, there is some change operations that can be done in PostgreSQL, some of which are mentioned in the research(Dijkstra, 2021) in table II.2.

Table II.2 List of Change Operations for Relational Database Schemes (Dijkstra, 2021)

No.	Change Operation Type
1	addColumn
2	addNotNullConstraint
3	addForeignKey
4	addPrimaryKey
5	addUniqueConstraint
6	createIndex
7	createSequence
8	createTable
9	dropColumn
10	dropIndex
11	dropSequence
12	dropTable
13	dropNotNullConstraint
14	dropUniqueConstraint
15	modifyDataType
16	renameColumn
17	renameTable

Besides change operations There is also a scheme change operations which aims to to do performance tuning, using the denormalization method. Denormalisasi is a technique for changing the schema in a database with the aim of achieving better performance by minimizing the amount of data and reducing join operation in the table. There are several denormalization methods as follows:

Table II.3 List of Denormalization Change Operations (Dijkstra, 2021)

No.	Denormalization Operation Type	Description
1	Collapsing Tables	Combining related tables by operation join table
2	Horizontal Splitting	Placing data in two separate tables, depending on the data values in one or more columns
3	Vertical Splitting	Partition relationships to isolate existing data most frequently accessed, retrieving only the necessary information
4	Adding Redundant Columns	Reduces the join method by moving columns from one table to another table.
5	Adding Derived Columns	Reduce aggregate operations by computing new columns
6	Adding Redundant Tables	Speed up data retrieval by only using frequently accessed columns

I.7.3 Database Replication

Database replication is the process of storing data in more than one place or node. This is useful in increasing data availability. The database replication process basically copies data from a database from one server to another server so that node can share the same data without inconsistencies. The result is a distributed database where users can access data relevant to their tasks without disrupting anyone else's work. Data replication involves continuously duplicating transactions, so that the replica is in a consistently updated state and synchronized with its source(van Kampen, 2022).

Based on the official documentation from(PostgreSQL, 2022), the use of data replication aims to create database cluster for in-case of high availability.

I.7.3.1 ReplicationMaster-Slave

Master-slave replication in PostgreSQL involves using a server slave solely for performing read and query fail-over events. During a fail-over event, the server slave can assume the role of the master. Master-slave replication facilitates the replication of data from one database server (the master) to one or more other database servers (the slaves). The master logs updates, which are then transmitted to the slave. The slave acknowledges that it has received the update successfully, enabling the delivery of the next update. Master-slave replication can be either synchronous or asynchronous, depending on when the changes propagate. If changes are made to the master and slave simultaneously, it is considered synchronous. If changes are queued and written later, it is considered asynchronous (PostgreSQL, 2022). The visualization of master-slave replication can be observed in Figure II.1.

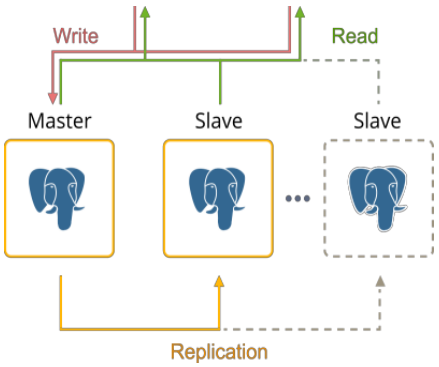


Figure II.1 Master-slave replication (PostgreSQL, 2022)

I.7.3.2 ReplicationMulti Master

Multi-master replication means that there are multiple nodes, each of which acts as a master node. Data is replicated between these nodes, allowing updates and inserts to be performed on any of the master nodes. In this scenario, there exist multiple copies of the data, and the system is responsible for resolving conflicts that may arise from concurrent changes (PostgreSQL, 2022). The visualization of multi-master replication can be observed in Figure II.2.

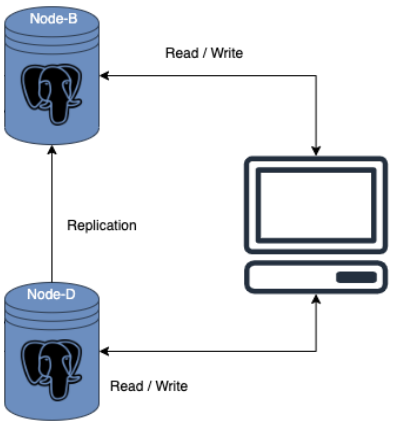


Figure II.2 Multi-master replication (PostgreSQL, 2022)

I.8 Deployment Application

Deployment is one of the phases in the software development life cycle (SDLC). The deployment process involves releasing the latest version of an application, allowing users to access and utilize it (van Kampen, 2022).

I.8.1 Blue-Green Deployment

Blue-green deployment is a deployment strategy that involves creating two separate but identical environments. One environment (blue) runs the current version of the application, while the other environment (green) runs the new version. This strategy enhances application availability and reduces deployment risk by simplifying the rollback process in case of deployment failures. After completing testing in the green environment, application traffic is immediately redirected to the green environment, and the blue environment is no longer utilized. One of the advantages of blue-green deployment is that users will not experience downtime. However, users may encounter some latency when transitioning to the green environment (Rudrabhatla, 2020). Figure II.3 illustrates the concept of Blue-Green Deployment.

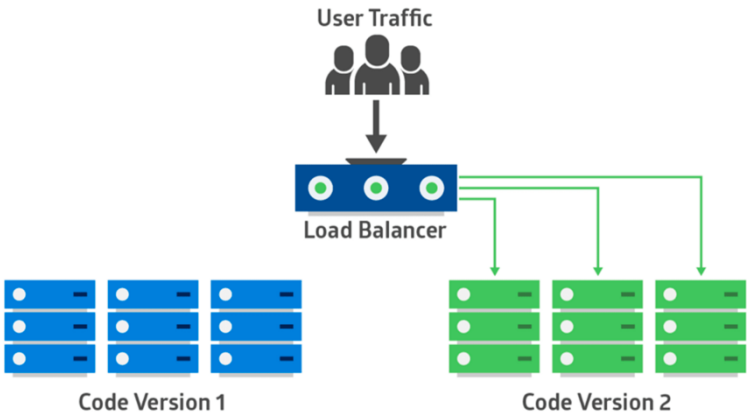


Figure II.3 IllustrationBlue-Green Deployment

Source: https://faun.pub/blue-green-deployment-strategy-7c0a07364e6

Blue-green deployment Suitable for system implementation that requires the following factors:

1. Critical Applications: These are systems vital to an organization's operations, where downtime cannot be compromised. They include platforms for e-commerce, financial systems, health applications, and emergency services.
2. Large-scale web-based applications, such as websites and web applications catering to a significant user base over extended periods, can experience revenue loss or user dissatisfaction due to even short periods of downtime.
3. Complex Microservice Architecture: In microservice based systems where multiple services are interconnected and need to be updated simultaneously or in coordination to maintain compatibility.
4. A/B Test: If you want to compare the performance and user experience of two versions of your app in a production environment, blue-green deployment allows you to switch between versions "blue" (current) and "green" (new) easily.
5. Upgrade Without Downtime: Applications that require upgrades or modifications without causing downtime or disruptions to end users' services.
6. Ability to Rollback: In systems where the capability to swiftly revert to a previous version is crucial in the event of issues or unexpected behavior.

7. Test Environment: Using the environment "green" as an areastaging to test not only applications but also infrastructure changes, configurations, and compatibility with various dependencies.
8. Geographically Distributed Systems: In scenarios where the application is implemented across multiple geographic regions,blue-green deployment can help ensure a smooth transition without impacting users in different locations.
9. Compliance and Regulatory Requirements: In industries such as finance or healthcare, where strict compliance and auditing are required,blue-green deployment can help maintain necessary controls when updating software.

I.9 Change Data Capture

Change Data Capture (CDC) has several definitions. One definition is that CDC is a method for integrating data based on the results of identification, capture and delivery only for data changes in operational systems(Tank dkk., 2010). Another definition of CDC is a technique for monitoring operational data sources that focuses on data changes (Kimball & Caserta, 2004). Based on these two definitions, it can be concluded that CDC is a method for knowing and detecting data changes that occur when transactions occur in operational systems.

In general, CDC applications can be categorized into immediate data extraction (immediate data extraction) and deferred data extraction (deferred data extraction) (Ponniah, 2010). Immediate data extraction enables extraction in real time when changes occur in the data source. Meanwhile, in the deferred data extraction approach, the extraction process is carried out periodically (in certain intervals). Therefore, the extracted data is the data that has been changed since the last extraction.

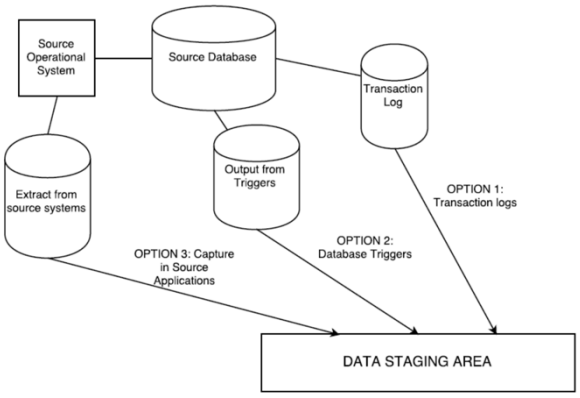


Figure II.4 Immediate data extraction using the CDC method (Ponniah, 2010).

There are three immediate data extraction methods for CDC applications. Figure II.4 illustrates these three methods, which involve transaction logs, database triggers, and capturing within the source application.The CDC method utilizing transaction logs relies on the logs recorded within the RDBMS. This approach leverages events such as Create, Update, and Delete (CRUD) operations recorded by the RDBMS in a log file. Systems employing this method scan the log file's contents to identify changed or added data.The second method involves the use of database triggers. Database triggers are procedural functions that an RDBMS typically performs when CRUD operations are executed on specific data. These functions can be used to apply updates when data changes occur.The third method is deep capturing within the source application. This approach harnesses an application or system as a data source capable of implementing CDC. However, it's important to note that this method is only applicable to applications that can be modified (Ponniah, 2010).

I.9.1 Debezium

Debezium is an open-source software platform used to capture data changes in databases (Change Data Capture or CDC) in real time. It is a useful tool in data integration, database monitoring and data analysis that allows you to identify, track and replicate data changes occurring in database management systems such as MySQL, PostgreSQL, MongoDB and others.

Debezium works by observing changes that occur in the database, including additions, updates, and deletions of data, and then sending these changes to other systems or other data stores for further analysis. This is especially useful in the context of developing data-driven applications that require real-time data updates or in monitoring database systems.

So, in short, Debezium is a tool for capturing data changes in a database and sending them elsewhere for further use, and it can be used in a variety of data development and analysis contexts.

I.10 Write-Ahead Logging (WAL)

Write-Ahead Logging (WAL) is the standard method for ensuring data integrity. The primary concept of WAL is that changes in file data, where tables and indexes reside, should only be written after the change has been logged—specifically, after the log record describing the change has been stored permanently. By following this procedure, there is no need to write data to disk with every transaction commit. This approach provides confidence that in the event of a crash, the database can be restored using the logs, allowing for the redoing of any changes not yet applied to data pages (a process known as roll-forward recovery or REDO) (PostgreSQL, 2022).

I.11 Logical Replication

Logical replication is a method for replicating data objects and their changes based on their replication identity, typically the primary key. It utilizes the principles of publishing and subscribing to one or more publications on an older version of the database. Subscribers pull data from the publications to which they are subscribed (PostgreSQL, 2022), as illustrated in Figure II.5.

The process of logical replication usually starts with taking a snapshot of the data in the publisher's database and copying it to the subscriber. Once completed, changes made in the publisher are sent to subscribers in real-time. Subscribers apply the data

in the same order as the publisher to ensure transaction consistency within a single subscription. This data replication method is referred to as transactional replication.

Whenever a row is modified in a PostgreSQL table, the change is logged in the Write-Ahead Log (WAL). The logical decoding module in PostgreSQL analyzes these records to produce plugin output, converting the WAL format into plugin format, such as a JSON object. These reformatted changes are then sent out of PostgreSQL via slot replication to consumers, which display the changed tuples. Below is an example of an executed query and a WAL record generated using the Wal2JSON plugin.

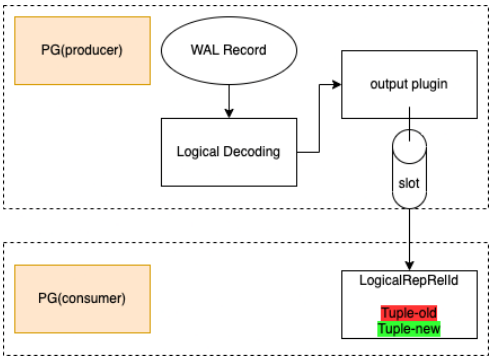


Figure II.5 Simplification ArchitectureLogical Replication

Logical decoding in Postgres can only provide information about DML (data manipulation) events, which include INSERT, UPDATE, and DELETE operations. DDL (data definition language) changes, such as CREATE TABLE, ALTER ROLE, and DROP INDEX, are not generated by logical decoding. In the case of INSERT and UPDATE operations, a new row is added to the table, and this new row's data is always sent to the output plugin. However, for DELETE and UPDATE operations, the primary key from the old row is sent from the Write-Ahead Logging (WAL) records."

I.12 Stream Processing

Stream processing is a continuous and real-time data processing method (Hesse & Lorenz, 2016). It involves collecting and processing new data as it arrives to meet the organization's immediate needs. The requirement for swift processing of large data

volumes and making real-time decisions underscores the use of streaming methods (Ali & Abdullah, 2018).

In traditional data processing (batch), transaction data accumulated over a period is extracted, loaded, and processed (including integration and transformation) for entry into analytical storage. When applied to large datasets, this traditional processing approach may result in slower processing times. To expedite data processing, it can be carried out automatically in a streaming or real-time manner, where incoming data is promptly processed and stored in analytical storage (Ali & Abdullah, 2018).

I.13 Infrastructure as a Code

According to Morris (2020) Infrastructure as Code is an infrastructure automation approach based on practices from software development. It emphasizes consistent and repeatable routines for provisioning and changing systems and their configurations. Every change to the code, automation will be used to test and apply the change to the system.

There are several benefits to using Infrastructure as Code (IaC) to manage software system infrastructure, including:

1. Self-service. By using IaC technology, the development team can write the necessary code themselves to prepare the system infrastructure they need because the code is not much different from the programming language code they use every day.
2. Speed. The operations required to manage infrastructure are much faster when executed by machines (via IaC technology) than when typed manually by humans.
3. Security. The majority of IaC technologies have mechanisms to ensure consistency in the code written. This minimizes the chance of fatal errors occurring when code is deployed to our infrastructure.

4. Documentation. As long as all our team members can read the code, then all team members can see clearly from the IaC code that we write how the infrastructure is created and what its components are.
5. Version Control. Like all code, the IaC code we write can be managed with version control tools such as git. In this way, we can always monitor the history of changes to the IaC code that we manage.
6. Validation. By using IaC, we can carry out code reviews, automated tests (both unit tests and integration tests), or other techniques that can reduce the risk of production defects in our IaC code.
7. Reuse. Most IaC technologies have modules that can be easily deployed and reused. For example, Ansible has playbooks, Chef has cookbooks, and Terraform has modules.

I.14 Kubernetes

Kubernetes is an orchestration tool used to manage a group of containers (J. Shah and D. Dubaria, 2019). A container is a self-contained environment for a piece of software, containing all the necessary files and libraries required for the software to run smoothly (J. Shah and D. Dubaria, 2019).

I.14.1 Primitives on Kubernetes

In Kubernetes, there are the newest primitives that compose an application that runs on Kubernetes according to Bilgin Ibryam and Roland Huß (2019).

1. Container

   That is a building block for applications running on top of Kubernetes. A container image serves as a functional unit, being self-contained and defining and carrying runtime dependencies. Container images are also immutable and cannot be changed once they have been built. Containers also have an API to expose their functionality.

2. Pods

Pod is the smallest execution unit of an application in Kubernetes. One pod may contain one or more containers. Using multiple containers is done when one container requires another as a helper. Each pod has an IP address, name, and port range that are shared among all containers within that pod. A pod can be likened to an atom in Kubernetes, serving as the environment where the application resides.

3. Services

   In an impermanent (ephemeral) environment, it means that resources can be easily created and destroyed. This flexibility is attributed to factors such as health checks, node migration, scaling up, and scaling down, which can trigger resource provisioning or de-provisioning. Additionally, the IP addresses assigned to pods are dynamic. Therefore, a service is required to address this challenge. A service acts as an entry point to resources, binding an IP address to the name of the service. This approach allows the service to function as an entry point for one or multiple resources. Services also serve as DNS (Domain Name Service) for facilitating service discovery within a Kubernetes cluster.

4. Labels

   There are various microservice applications, each serving different purposes. To address this diversity, Kubernetes offers Labels and Namespaces. Labels serve to distinguish between different applications. For instance, there are applications in the form of frontends, backends, customer services, and buyer services. By using labels such as 'app' and 'type,' these can be grouped together, providing a means to differentiate between pods."

5. Namespaces

   With namespaces, a Kubernetes cluster can be divided into various resources. A namespace provides a scope for Kubernetes resources and mechanisms for deploying policies differently on the cluster based on its namespace. For example, for development, production, and testing purposes.

6. Jobs

   A job creates one or several subtasks, ensuring that the specified number of subtasks is successfully executed as previously defined. When a subtask is completed, the job marks it as successfully executed. Once the predetermined number of successful executions is achieved, the job is considered complete. Deleting a job will also remove all subtasks created by that job.

   A simple use case involves creating a job object to ensure the successful completion of a subtask. This job object generates a subtask only when the initial subtask fails or is deleted, such as due to hardware failure or a node reboot event.

I.15 Schema Migration UsingExpand-Contract Pattern

In a thesis authored by Van Kampen (2022), titled "Zero Downtime Schema Migrations in Highly Available Databases," the examination of the "expand-contract" pattern in database management to align with blue-green deployment is discussed. When database schema changes occur, it becomes necessary to maintain two versions of the application, corresponding to two versions of the domain model, simultaneously. This necessity gives rise to the requirement for a database in a mixed state, referred to as an "in-state database."

The "expand-contract" pattern comprises two sequential steps: expansion and contraction. For instance, if there is a need to rename a column, the expansion step involves creating a new column with the updated name. At this point, the database is in a mixed state, accommodating both old and new versions of the application. Subsequently, the new and old columns are synchronized by applying update and insert operations to both columns and transferring the remaining data from the old column to the new one. This synchronization can be achieved through application code or trigger-based data handling.

However, this method has limitations, as it only supports schema changes related to table structure modifications, such as column renaming, dropping a column, or adding a column. It does not support other options, such as altering constraints or index changes. Additionally, it may lead to a decrease in database performance due to the utilization of triggers.

To address these limitations and maintain proper database performance, the thesis suggests using the "change data capture" method in conjunction with "extract-transform-load" processes. This approach allows for a more comprehensive review of performance changes during the mixed-state period and supports alterations beyond just table structure modifications, including changes to the logical aspects of the database. It is important to note that this method cannot be directly compared to the "blue-green deployment" method for databases, as it lacks the concept of a new database instance and a feasible database version rollback mechanism.

BAB II

PROBLEM ANALYSIS AND SOLUTION DESIGN

II.1 Problem Description and Analysis

Chapter I.1 explains that, up to this point, there have been no tools available for implementing blue-green deployment in the database. The database itself is discussed in subsection II.1, followed by an explanation of deployment definitions, especially blue-green deployment, in subchapter II.3, and software architecture based on microservices in subchapter II.2.

The main challenge with blue-green deployment in the database is the schema change process. Research in this area adopts a change-data-capture approach, as discussed in subsection II.4, utilizing transaction logs recorded within the RDBMS. This method leverages every Create, Update, Delete (CUD) event logged by the RDBMS in a file known as the Write Ahead Log (WAL) in PostgreSQL, explained in subchapter II.5. The data processing occurs automatically in real-time, as described in subsection II.6. The various types of change operations are detailed in subsection II.1.2.

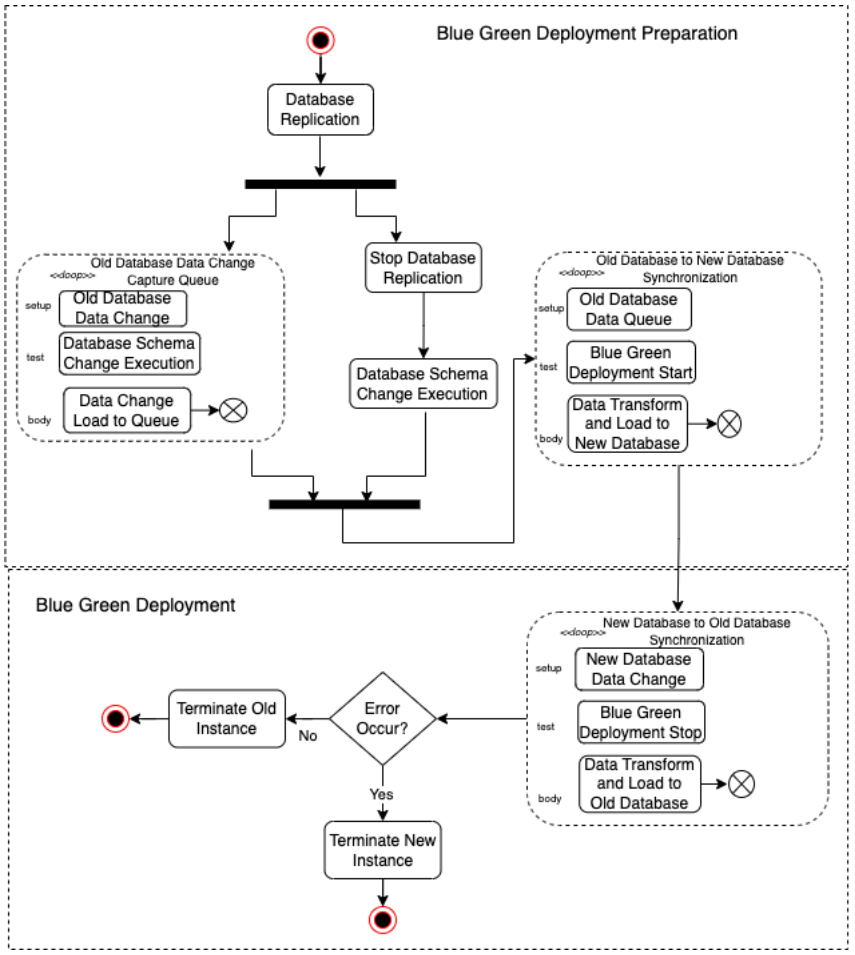
II.2 Solution Analysis

II.2.1 ProcessBlue-Green Deployment On Database

The Blue-Green Deployment process commences by replicating the database to generate a new database. Once the data from the replication process matches that of the old database, the replication process for the new database version will halt. Any changes to the data in the old database must then be queued. Subsequently, a query is executed on the new database to modify the database schema. Following this, data that underwent changes in the old database, previously queued, must have its schema adapted to align with the new database schema. A query is created to insert this data into the new database version, all of which occurs prior to the initiation of the actual Blue-Green Deployment.

During the active Blue-Green Deployment phase, the application's traffic shifts, causing the application to utilize the new version of the database. Consequently, data that changes in the new database must undergo schema modifications to align with the old database schema. A query is crafted to insert this data into the old database version until the Blue-Green Deployment is completed.

The duration of the Blue-Green Deployment period depends on the company's decision. If this transition period proceeds without issues, the old database will be terminated, and the application will run with the new database. However, in case of any errors, the application will revert to the old version.Figure III.1 shows activity diagram of blue-green deployment in database.

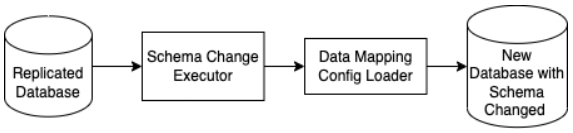


Picture III.1 Activity Diagram Blue-Green Deployment On Database

II.2.2 Design of Tool Module

II.2.2.1 Design of Tool Modules in the Blue-Green Deployment Preparation Phase

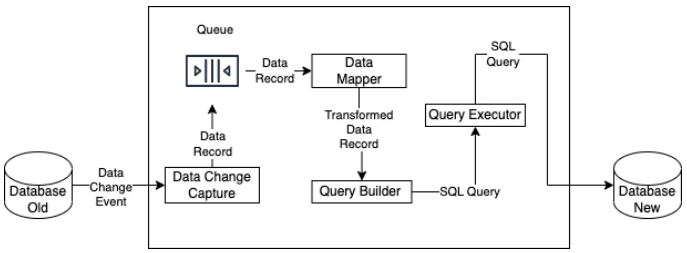
As explained in subsection III.2.1, there are several processes in the preparation phase of blue-green deployment. During the preparation phase of blue-green deployment, Tools does not handle database replication; instead, database replication is carried out by the user using methods supported by the database. Once the new database has successfully created a replica, the tool will make schema changes using the module depicted in Figure III.2.



Picture III.2 Module Design for Tools in Preparation for Blue-Green Deployment Schema Changes

Below we can identify several main modules of tools in the database schema change phase.

1. The "Destination Database Check" component serves to verify the status of the new database version, determining whether it is in "read-only" mode or ready for "write" operations, and confirming the data's synchronization and overall integrity.
2. The "Schema Change Executor" component serves the function of executing queries necessary for implementing schema changes, such as "ALTER TABLE" statements.
3. The "Data Mapping Config Loader" component, on the other hand, is responsible for inputting configuration data required for mapping schemas that have undergone changes to the Data Mapper Component in other processes.



Picture III.3 Module Design for Tools in Data Change Event Preparation Blue-Green Deployment

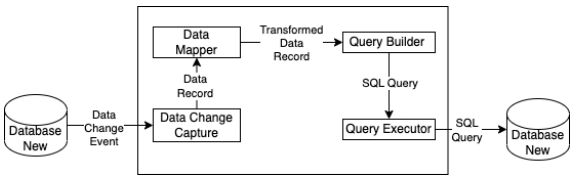
After changes have been made to the database schema, several main modules of the tool can be identified in the preparation phase of blue-green deployment, which runs every time a data change event occurs in the database, as illustrated in Figure III.3.

1. Data Change Capture component: This component is responsible for performing change data capture by capturing every data change in the database. Data changes are received through this component and queued for processing.
2. Data Mapper component: This component handles mapping related to schema changes made during the blue-green deployment. It becomes active after the Schema Change Executor component successfully executes and updates the database schema to the new version. The Data Mapper component pulls data from the queue previously populated by the Data Change Capture Component.
3. Query Builder component: This component is responsible for creating SQL queries based on data changes that have been mapped to the data schema in the Data Mapper component.
4. Query Executor component: This component executes queries created by the Query Builder on the new version of the database.

II.2.2.2 Design of Tool Modules in Blue-Green Deployment Phase

The modules used in the blue-green deployment phase are no different from those used in the preparation phase. However, a distinction arises with regard to the database. In the previous phase, data was sourced from the old version of the database, whereas in this phase, it is sourced from the new version of the database. Additionally, within the Data Mapper Component, data is no longer drawn from the

queue, as the old version of the database is in a writable state and does not undergo changes in the schema, as illustrated in Figure III.4."



Picture III.4 Plan of the Appliance ModuleData Change Event Blue-Green Deployment

II.2.3 Solution AnalysisChange Data Capture

There are several methods available for retrieving data for each database change. One method that can be employed is using transaction logs and triggers within the database. Below, we'll explore the advantages and disadvantages of each of these methods.

Table III.1 Comparison of MethodsChange Data Capture

Method	**Advantage**	**Deficiencies**
Transactio n Log	There isn't any overhead on the system because logs are already integrated into the database process.	It requires the development of tools to retrieve data from each transaction log.
Trigger Basis Data	Creating procedures and triggers with comprehensive documentation in the database makes it easier to develop programs.	Creating a trigger can lead to overhead on the database, and using HTTP to send data is not recommended.

Because the method selection prioritizes minimal overhead on the database, transaction logs are chosen as the method used in the tool's development.

BAB III
IMPLEMENTATION AND TESTING

In this chapter, the system requirements analysis and general description of the system will be explained. This chapter also explains the system architecture design and implementation of the modules contained in the tool. Apart from that, in this chapter, test scenarios, test results and analysis of the test results are also explained.

III.1 System Requirements Analysis

The tools developed will serve as middleware used to synchronize data in a blue-green deployment microservice. The following are the functional requirements for the tools, as presented in Table IV.1.

Table IV.1 System Functional Requirements

Requirement ID	**Description of Needs**
FR-1	The tool can subscribe to any data changes.
FR-2	The tool can modify the schema of data in different versioned databases.
FR-3	The tool can receive configuration changes for database schema modifications.
FR-4	The tool can rebuild SQL queries according to the schema of the target database version.
FR-5	The tool can determine when replication of the new database version has caught up with the old database version and stop the replication process.
FR-6	The tool can modify data flow in blue-green deployment.
FR-7	The tool can generate reports on data replication status between the old and new database versions.

III.2 General Description of the System

The developed system focuses on implementing blue-green deployment in microservices using PostgreSQL databases, involving changes to the database schema. In general, the developed system follows a workflow consisting of several steps.

First, the user creates a read replica in PostgreSQL using the logical replication method. Logical replication is required because, with this method in PostgreSQL, changes to Data Modification Language (DML) are recorded at the row level.

Second, when the database is approaching the catch-up state, the user is required to run Debezium to perform change data capture at the row level changes in the database. The Debezium used requires configuration to establish connections with the old and new database versions as well as Redis.

Third, the user is required to run the change data capture tool designed by inputting commands to execute it.

Fourth, when the user wants to initiate the database schema change process, they enter a command to start the database schema change process.

Fifth, the tool will begin the data sync process to synchronize data between the old and new database versions.

Sixth, when the user wants to perform blue-green deployment, they are required to run another Debezium to capture data changes in the new database version.

Seventh, during the blue-green deployment process, the user enters a command into the tool to switch data traffic from the new database version to the old one. When the blue-green deployment process is complete, the user can stop the tool. Figure IV.1 is a sequence diagram of the tool.

III.3 System Architecture Design

The following is the process design carried out by the chief in carrying out the process for each data change which is explained in Figure IV.2.

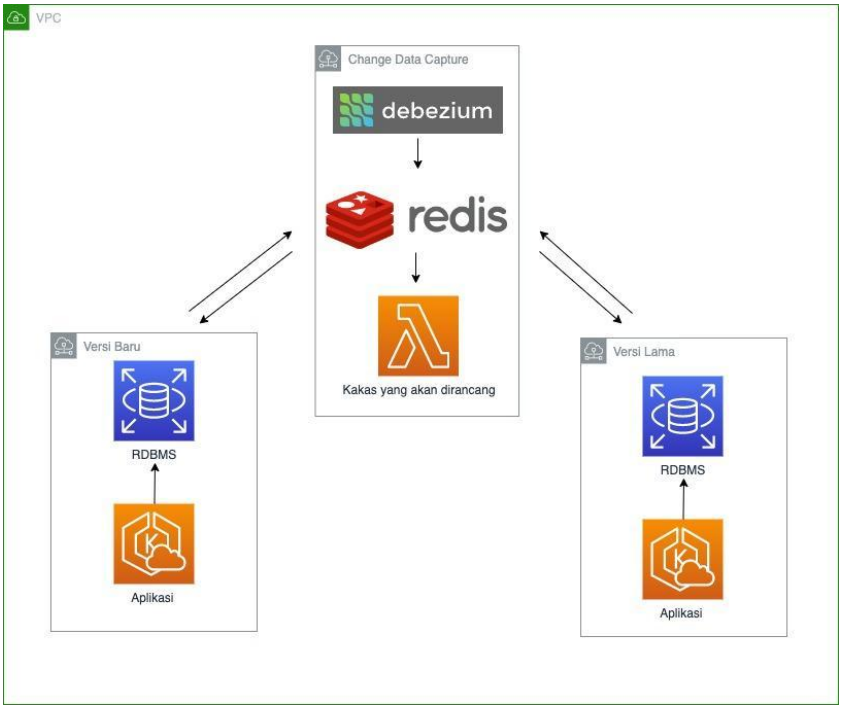


Figure IV.2 Tool Architectural Design

Every data change that occurs in the RDMS, including Create, Update, and Delete events, will be captured by Debezium and formatted into messages. By making modifications to Debezium, this data will be queued into Redis.

Debezium is an open-source distributed tool for capturing changes in a database, allowing the database to observe and respond to these changes. Debezium records all row-level changes in every database table in a change stream, and applications can simply read this stream to see changes in the same order.

In its usage, Debezium requires the use of messaging services such as Redis Stream and Kafka. Redis Stream is chosen because it stores data in RAM, resulting in lower data latency and resource usage. Redis is also used as a database for inter-process messaging.

III.4 Tool Architectural Design

Based on the identified functional requirements, a system architecture design for the tool can be formulated. The implemented system consists of five modules, namely, the stop replication module, schema change executor module, data config loader module, data change capture module, data mapper module, SQL builder module, SQL executor module, database module, and a command-line interface (CLI) module serving as the interface for the tool as described in Table IV.2.

Table IV.2 Mapping of Implemented Modules Against the Design

No	Module	Components involved in Subchapter III.2
1	CLI	Additional components frontend interface
3	ddl_change_executor	Schema Change Executor
4	config	Data Mapping Config Loader
5	cdc	Data Change Capture
6	data_transformer	Data Mapper
7	query_builder	SQL Builder
8	query_executor	SQL Executor
9	report_generator	Additional components are optional to check replication status

The explanation for each module is as follows.

1. The CLI module serves as the interface for the developed tool. This module receives commands from users to perform various tasks, including initiating schema changes and starting the replication process from the old database version to the new database version. It also handles replication changes during blue-green

deployment and initiates the replication process from the new database version to the old database version. This module utilizes the github.com/spf13/cobra library to create the CLI interface.

2. The ddl_change_executor module is responsible for stopping the replication process on the database, intending to create a replica of the old database. This is done as part of the full data load process. In this module, the tool will read from Redis until it obtains an accessible key. When a user issues a command to make schema changes, the tool will check the replication lag of the replica database to determine if it has caught up with the old database version. If the replica database is in a catch-up state, the tool will record the timestamp of the replication process's termination in Redis, halt the replication on the replica database, and execute schema changes on the replica database, transforming it into the new database version as depicted in Figure IV.3.

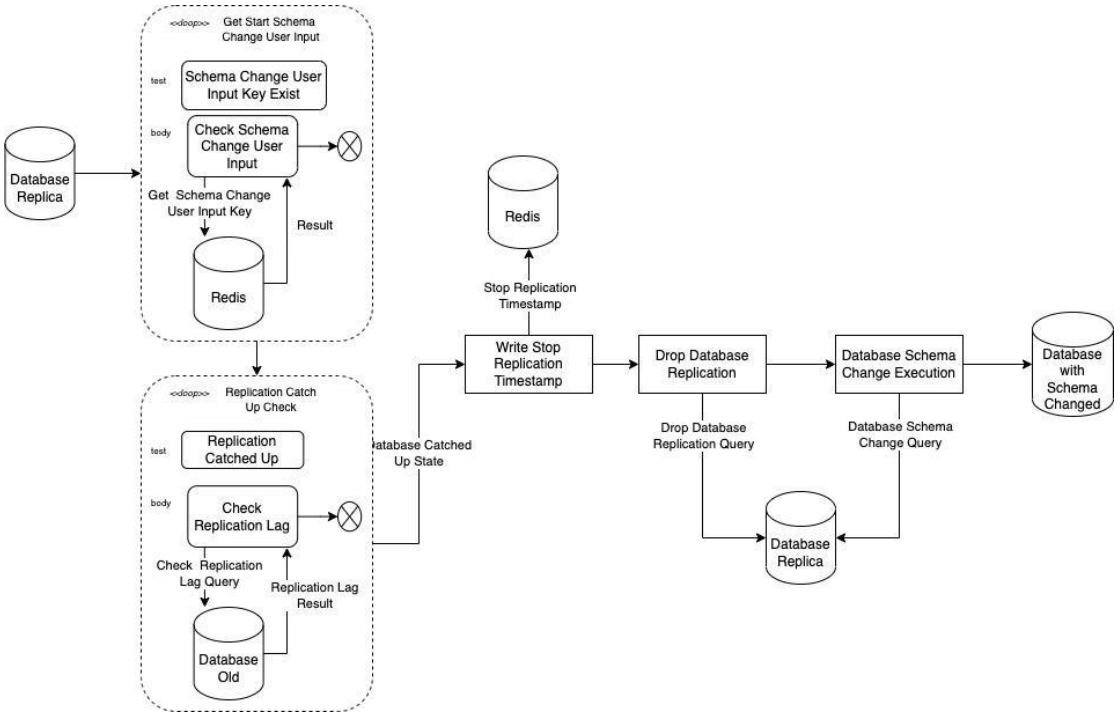


Figure IV.3 Detail of Module ddl_change_executor

3. The config module serves to input data from the configuration file into the tool and initialize the tool according to the configuration. Initialization involves mapping change operations to the relevant tables and initializing the search for the primary key used as a pointer during delete and update operations. Following that, topic names are initialized in the Redis Stream for each table, resulting in a tool with the updated configuration as described in Figure IV.4.

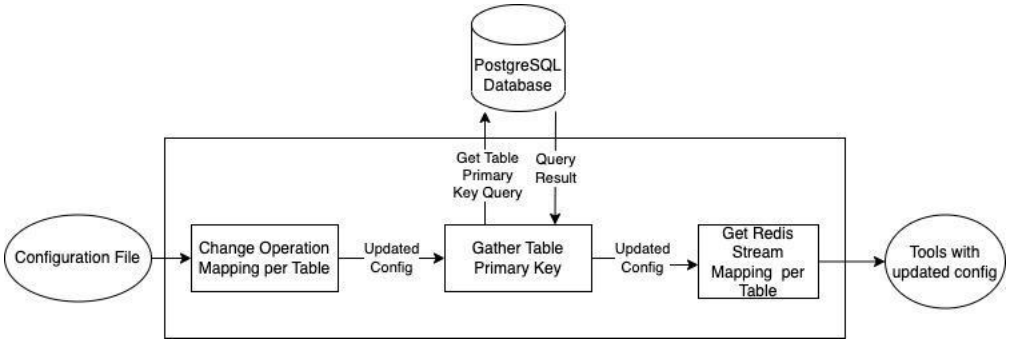


Figure IV.4 Details of the config module

4. The CDC module functions to receive data changes from the database that have previously been captured by Debezium based on WAL events and stores them in a Redis Stream. Upon initial use of the module, it performs a GET operation on Redis to obtain the timestamp from the start time of data synchronization. This is done to avoid data duplication, especially if the table lacks unique constraints. The module then enters a loop to read the Redis Stream, and reading the stream requires a list of stream names obtained from previous configurations. The data obtained from the stream is then checked for its timestamp. If the timestamp of the data is after the start time of data synchronization, it will be forwarded to the data_transformer for schema transformation, as described in Figure IV.5.

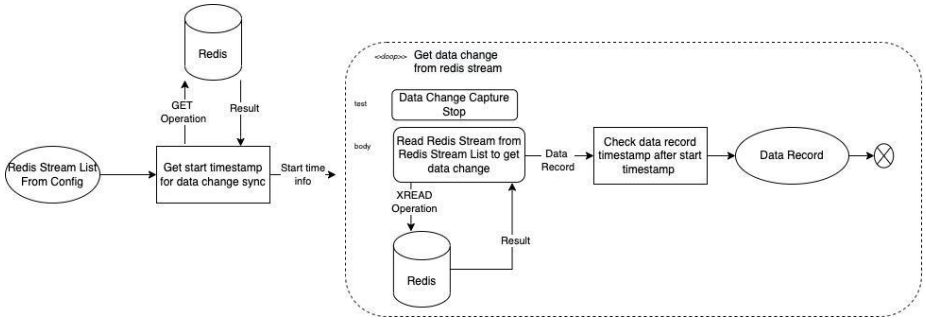


Figure IV.5 CDC Module Details

5. The data_transformer module is responsible for making changes to the data received in the cdc module. It retrieves the table's name and, based on the configuration, obtains the schema change configuration for that table. Then, the module transforms the data based on the configuration. The data changes in the configuration include change operations related to data schema, such as: addColumn, createTable, dropColumn, dropTable, renameColumn, renameTable, horizontalSplit, and verticalSplit. The result of the data record is identified based on the table name, Data Modification Language (DML) operation, and column information. The information obtained is then matched with the configuration file, and the data changes are mapped as described in Figure IV.6.

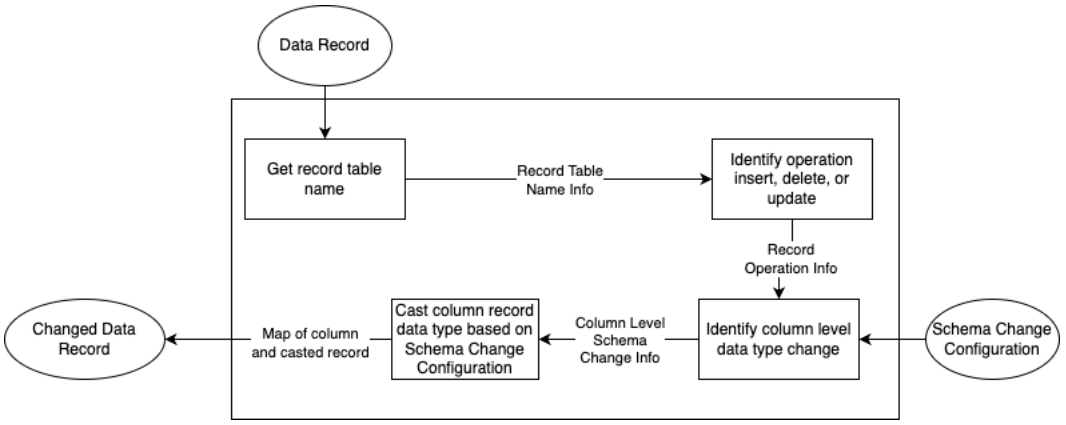


Figure IV.6 Details of the data_transformer module

6. The query_builder module functions to create SQL queries derived from data that has previously undergone modifications in the data_transformer module, as described in Figure IV.7.

Figure IV.7 Details of the SQL Builder module

7. The query_executor module serves to execute queries for making modifications to another version of the database. This module utilizes the github.com/lib/pq library for executing queries on PostgreSQL, as described in Figure IV.8.

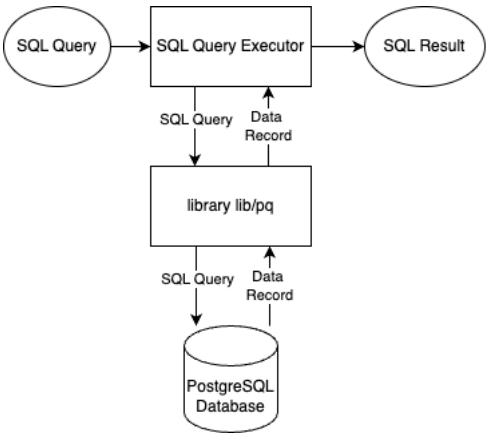


Figure IV.8 Details of the SQL Executor Module

8. The report_generator module functions to generate a data replication status report for both the old and new database versions. This module will execute queries to obtain data such as table names and the number of records in each table. It will also perform join operations to combine data from the databases using the previously created query_executor module.

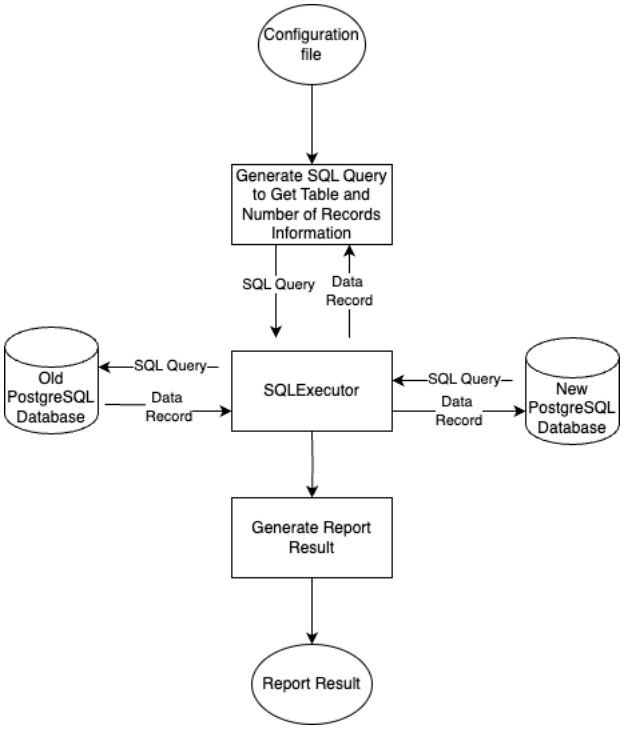


Figure IV.9 Details of the report_generator module

III.5 Implementation of Tools

This tool is implemented in the Go programming language. This choice was made because, on average, many tools are implemented in Go due to its relatively small executable size. Additionally, Go is a strongly typed language, which can facilitate future improvements or feature additions. Although Python is a commonly used language in machine learning experiments, this tool is not tied to specific features only available in Python, so using other languages is not a problem. When using the tool, input in the form of a configuration file in JSON format is required.

III.5.1 Configuration File Format

The tools used in this process require input in the form of a configuration file that contains credentials or authentication information for accessing both the old and new database versions. This configuration file plays a crucial role in facilitating the tools' access to and manipulation of data in both databases.

Within this configuration file, essential information is provided, such as the database server address, username, password, and other relevant details needed to identify and establish connections to both the old and new database versions.

Additionally, the configuration file also contains information regarding changes at the Data Definition Language (DDL) level in the old database. DDL is a part of the SQL language used for managing table structures and schemas in the database. In this context, DDL changes encompass various operations like creating, altering, or deleting tables, columns, indexes, and more.

The tools have implemented an SQL parser capable of identifying DDL changes from the provided SQL queries. However, there are cases that cannot be automatically identified by the tools, such as:

Data Type Changes: Changes involving alterations to the data type of table columns. When data type changes occur, they can impact the validity of existing data and require careful adjustment in the data conversion process.

In the tool's configuration file format, specific rules or explanations are provided regarding the handling of such cases. Potential changes that cannot be automatically identified by the SQL parser may necessitate manual steps or additional adjustments to ensure the proper functioning of the tools.

Overall, this configuration file serves as a vital bridge between the tools and both the old and new database versions, ensuring smooth access to both databases and effective management of DDL changes. With this configuration file in place, the process of migrating or updating the database becomes more structured, secure, and reliable, thus supporting the successful use of the tools in handling changes in the database environment as described in Figure IV.10.


```
{
  "ddlTransform":{"modifyDataType":[{"column":"","newType"
    :"","oldType":"","schema":"","table":""}]},"dest"
    :{"database":"","debeziumPublication":"","host":""
    ,"maxIdleConn":5,"maxOpenConn":5,"password":"postgres"
    ,"port":5432,"redisTopicPrefix":"","subscriptionName":""
    ,"username":""},"redis":{"database":0,"host":"localhost"
    ,"port":6379},"source":{"database":""
    ,"debeziumPublication":"","host":"","maxIdleConn":5
    ,"maxOpenConn":5,"password":"","port":5432
    ,"redisTopicPrefix":"","username":""},"sqlFile":"/cdc
    /test_sql.sql"
}
```


Figure IV.10 Configuration File Format

The following is an explanation of the variables contained in the configuration file which are explained in Table IV.3.

Table IV.3 Configuration File Description

Example	Description
{ "modifyDataType": [{	This change type is used to change the data type of a column in a table. The following is an explanation of the variables:

Example	Description
"column": "age", "newType": "integer", "oldType": "varchar", "schema": "public", "table": "person" }] }	1. column: The name of the column that will change the data type. In the example above, the column "age" will be changed from type "varchar" to type "integer". 2. newType: The new data type that will be applied to the column in question. 3. oldType: The old data type of the column before changes were made. In this example, the previous data type of the "age" column is "varchar". 4. schema: The schema name of the table that will undergo a data type change. 5. table: The name of the table that will change the data type.

"dest"/"source": {
"database": "",
"debeziumPublication": "",
"host": "",
"password": "",
"port": 5432,
"redisTopicPrefix": "",
"subscriptionName": "",
"username": ""
}
	This section contains information related to the database that will be used.
	
	1. database: Nodatabase the destination that will be used to store the data resulting from the changes.
	2. debeziumPublication: The name of the Debezium publication to be used for consuming data changes from the target database registered in PostgreSQL, with the default value of 'dbz_publication.
	3. host: Addresshost or IP of the destination database.
	password: Password (password) to access the database.

Table IV.3 Configuration File Description (Continued)

Example	Description
	4. port: Numberport which is used to connect to database objective. 5. redisTopicPrefix: Topic prefix used when data is sent to Redis. Topics are used to organize data in Redis. 6. subscriptionName: Nosubscription which is used when consuming data changes from to create a replica database. 7. username: Username (username) which is used to access the database.

In using Debezium as a supporting tool for change data capture, it also comes with its own configuration file. The Debezium configuration file is a crucial component in the implementation and utilization of this tool. This file contains various settings and details that define the data source to be monitored, the data format generated, synchronization mechanisms, and more. By using this configuration file, Debezium can establish connections and interact with the database, as well as organize changed data into a format that can be further accessed and understood. Below is the Debezium configuration file used in Figure IV.11.

```
debezium.sink.type=redis
debezium.sink.redis.address=${redis_host}:6379
debezium.source.connector.class=io.debezium.connector.postgresql.PostgresConnector
debezium.source.offset.storage.file.filename=data/offsets.dat
debezium.source.offset.flush.interval.ms=0
debezium.source.database.hostname=${db_host}
debezium.source.database.port=5432
debezium.source.database.user=${db_username}
debezium.source.database.password=${db_password}
debezium.source.database.dbname=${db_name}
debezium.source.database.server.name=${db_name}
debezium.source.plugin.name=pgoutput
debezium.source.topic.prefix=${topic_prefix}
debezium.source.snapshot.mode=never
debezium.source.decimal.handling.mode=double
```

Figure IV.11 Debezium Configuration File

Based on the system architecture described in Section III.2.2, five modules are implemented. The functions implemented in each module are as follows.

1. CLI module

   The Command-Line Interface (CLI) module plays a crucial role in user interaction with this tool. The CLI serves as an interface that allows users to implement and utilize the previously discussed modules more easily and efficiently. By using the CLI, users can perform various tasks and operations related to the tool directly through the command line or terminal.

   This CLI interface provides a set of pre-programmed commands by the tool's developers. These commands enable users to access and execute the tool's functions in an intuitive and straightforward manner. Below is an image of the tool's CLI interface. The CLI interface can be seen in Figure IV.12.

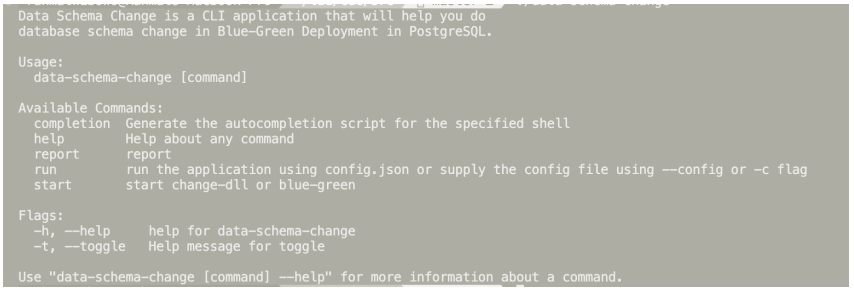


Figure IV.12 CLI Interface Tools

The following are several commands that can be used in Table IV.4.

Table IV.4 List of Tool Commands

Command Name	**Description**
run	This command functions to run the tool, this command is the main command of the tool and must be executed during the tool initiation process
start	This command is used to initiate the processes of both DDL changes and the beginning of the blue-green deployment. There are two subcommands:

Command Name	Description
	"change-ddl," which is used for schema changes, and "blue-green," which is used to execute the blue-green deployment process and switch traffic from the old version to the new version of the database.
report	This command is used to generate a report on the replication status of the database by comparing the data counts in tables between the old and new database versions.
help	This command is used to display a list and descriptions of available commands that the tool can use.

2. The ddl_change_executor module is designed to stop the replication process in the database, with the goal of creating a replica of the old database with a new schema. In the implementation of this module, there are several functions explained in Table IV.5, along with a diagram in Figure IV.13.

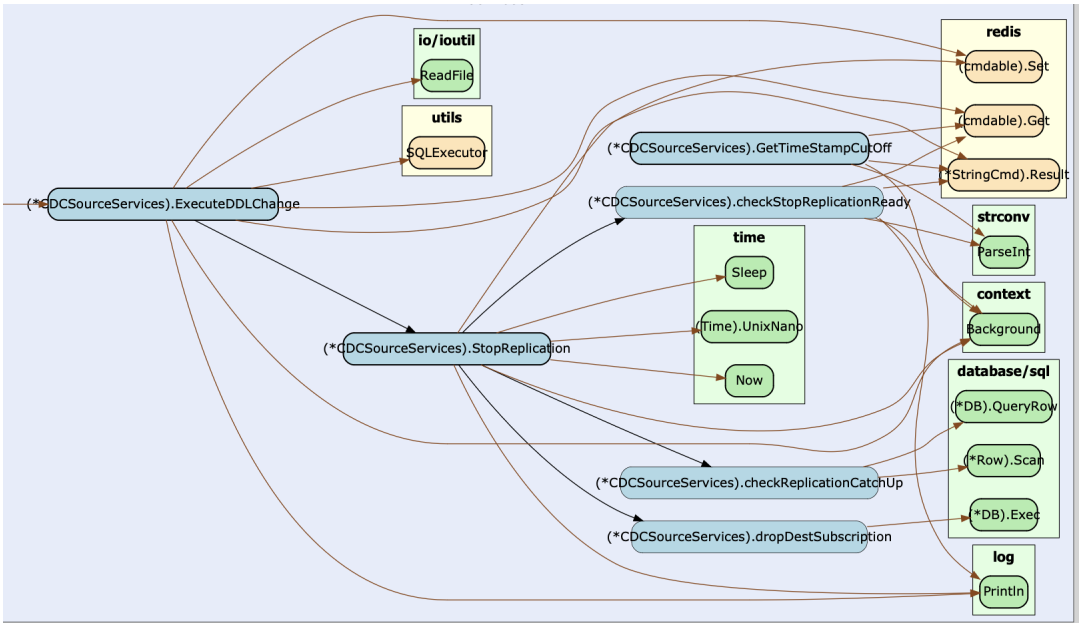


Figure IV.13 Diagram Function Call Modul ddl_change_executor

Table IV.5 Description of the ddl_change_executor Module Function

Function Name	**Description**
dropDestSubscription	Performs a query execution to stop the replication process. In this function, a query is used to drop replication on the PostgreSQL database of the new version.
checkReplicationCatchUp	Performs a select query by checking the log serial number (LSN) in the PostgreSQL database of the new version to determine if the data replication is in a state that allows it to be stopped.
checkStopReplicationReady	Determines whether the user has entered a command to immediately start the DDL change process if replication has reachedcatch up state.
getTimestampCutOff	Do a checktimestamp replication termination time stored in redis. This aims to reduce the potential for duplicate data during the replication termination period.

3. The config module is responsible for inputting data from the configuration file into the tool and initializing the tool according to the configuration. Initialization involves mapping change operations to the relevant tables and initializing the search for the primary key, which is used as a pointer during delete and update operations. The module utilizes the github.com/pganalyze/pg_query_go/v4 library to parse DML queries in PostgreSQL and transform them into tool configurations. It also employs the viper library for reading and parsing

configuration files. The implementation of this module includes several functions, which will be explained in Table IV.6 with a diagram in Figure IV.14.

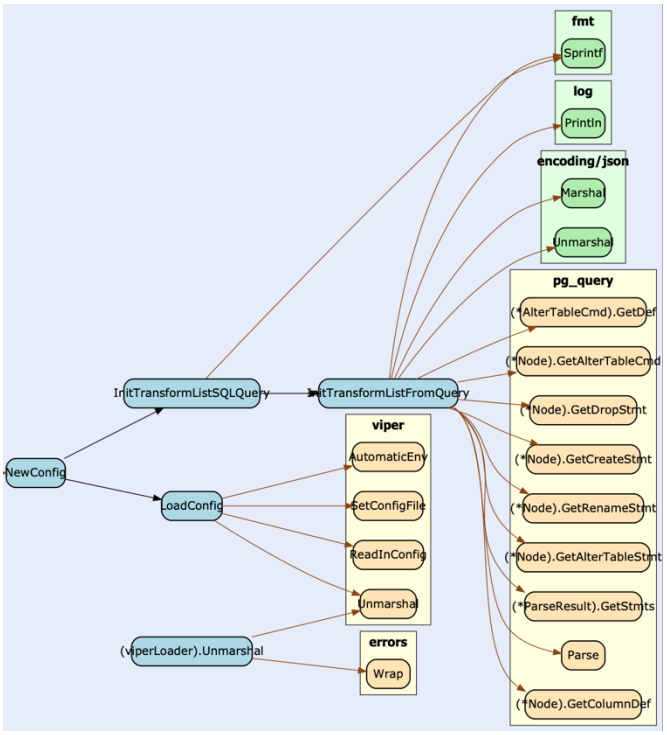


Figure IV.14 Function Call Diagram of config Module

Table IV.6 Description of Functions in the config Module

Function Name	Description
InitTransformListFromSQLQuery	Parsing the SQL statements within the SQL file to identify the set of change operations performed.
LoadConfig	Initiate configuration based on configuration file

4. The CDC module is designed to receive data changes from a database previously processed by Debezium, based on WAL events, and store them in a Redis Stream. Upon initial usage of the module, it performs a GET operation on Redis to retrieve a timestamp representing the start time of data synchronization. This is done to prevent data duplication during synchronization, especially when tables lack

unique constraints. Debezium reads WAL events in PostgreSQL and maps them to various topics in Redis based on the database tables. Therefore, the 'makeStream' function is used to identify and consolidate the Redis stream. The implementation of this module includes several functions, which will be explained in Table IV.7, accompanied by a diagram in Figure IV.15.

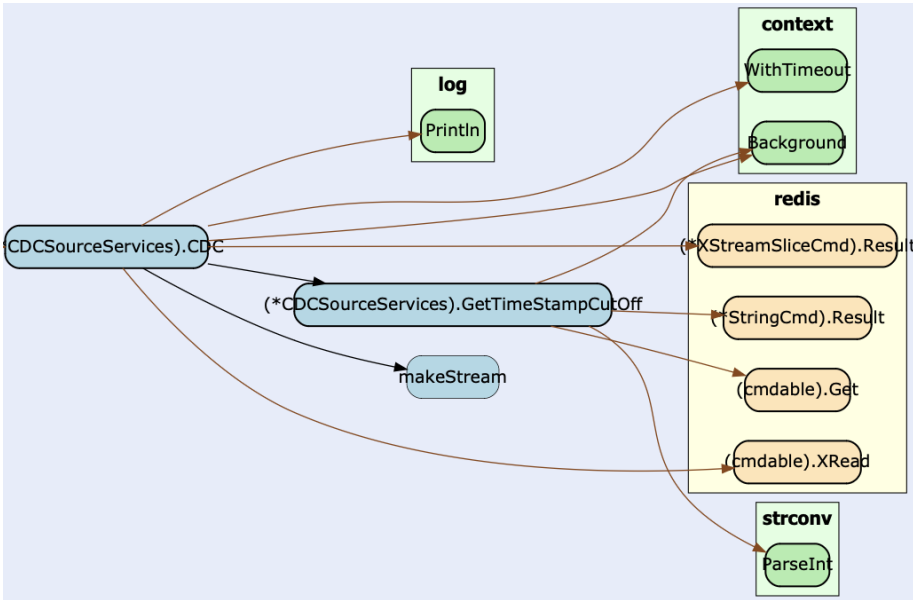


Figure IV.15 Function Call Diagram in the CDC Module

Table IV.7 Description of Functions in the CDC Module

Function Name	Description
GetTimestampCutOff	Parsing the SQL statements within the SQL file to identify the performed change operation sets
LoadConfig	Initiate configuration based on configuration file

5. The data_transformer module functions to make changes to the data received in the cdc module. In this module there are two child functions to change data originating from the old version of the database and the new version of the database. In the implementation of this module there are several functions which will be explained in table IV.8 with the diagram in Figure IV.16.

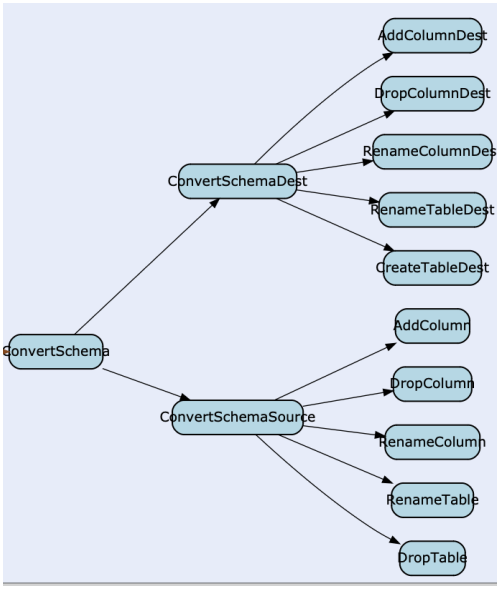


Figure IV.16 Function Call Diagram in the data_transformer Module

Table IV.8 Description of Functions in the data_transformer Module

Function Name	Description
ConvertSchemaDest	Make schema changes to data originating from the new version of the database.
ConvertSchemaSource	Make schema changes to data originating from an old version of the database.

6. The query_builder module is used to generate SQL queries based on data that has been previously modified in the Data Mapper module. To create queries, there are two different functions: ConvertDataToSQLNormal and ConvertDataToSQLVerticalSplit. This is because for normal SQL queries and horizontalSplit data, the necessary query information can be extracted only from the data records captured by Debezium. Special handling is required for temporal data because the data format captured by Debezium is in the form of integers originating from UNIX time. In the implementation of this module, there are several functions explained in Table IV.9 with a diagram shown in Figure IV.17.

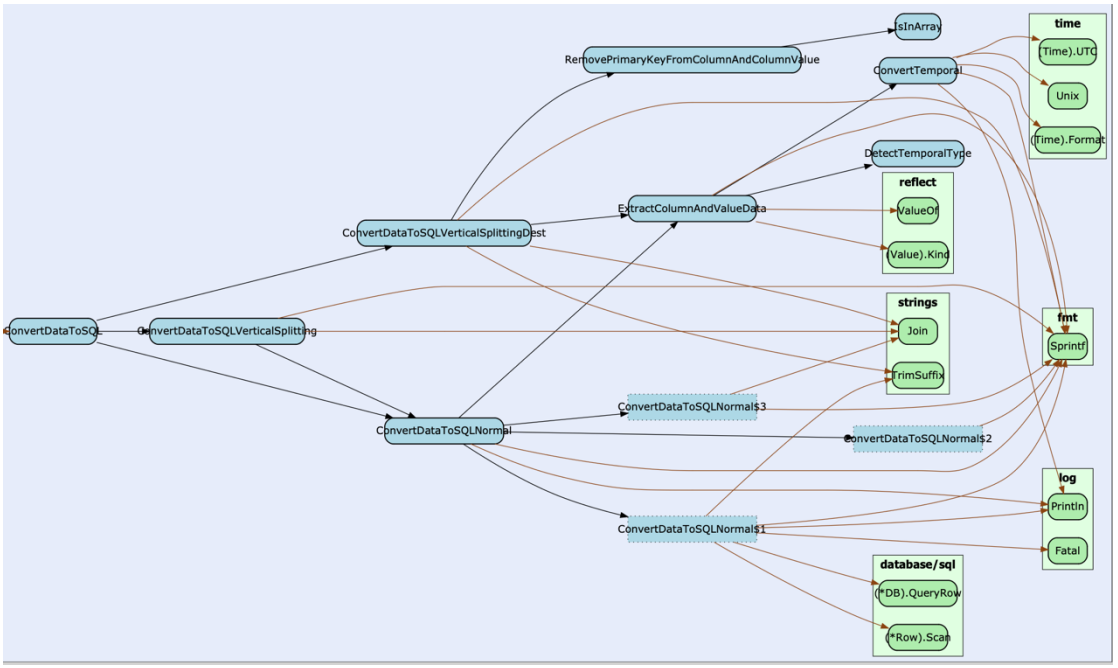


Figure IV.17 Diagram Function Call Pada Modul query_builder

Table IV.9 Description of Functions in the query_builder Module

Function Name	Description
ConvertToSQLVerticalSplitting	Performing schema changes on data intended for vertical splitting.
ConvertToSQLNormal	Performing schema changes on data undergoing modifications other than vertical splitting.
ConvertTemporal	Converting temporal data with an integer format into a string.

7. The query_executor module is used to execute queries for data modification in another version of the database. This module utilizes the github.com/lib/pq library to execute queries on PostgreSQL. In the implementation of this module, several functions are described in Table IV.10, with a diagram shown in Figure IV.18.

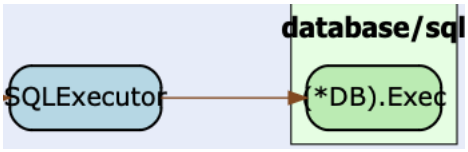


Figure IV.18 Diagram Function Call Pada Modul query_executor

Table IV.10 Description of Functions in the sql_executor Module

Function Name	Description
DB.Exec	Executequery SQL

III.6 Development Environment

In this final project a number of tools (tools) and library (library) which is used to implement the Change Data Capture tool This. The development environment used during implementation is explained in table IV.11.

Table IV.11 Tool Development Environment

No	**Specification**
Operating system	Mac OS Ventura 13.4.1 (22F82)
Processor	Apple M1
Programming language	Go 1.19
Relational DBMS	PostgreSQL 14.7
Streaming Platform	Redis Stream Using Redis version 6
Change data capture tools	Debezium Server 2.3
Container Tools	Orbstack 0.14.1

III.7 Testing Environment

The testing environment used to test tools in working on this final project is explained in Table IV.12.

Table IV.12 Tool Testing Environment

No	Specification
Cloud Service Provider	Amazon Web Services
Operating system	Amazon Linux 2 x86_64
Roosterchange data capture	Debezium Server 2.3
Container orchestration	Kubernetes 1.27
Kubernetes Worker Node	2 vCPUs, 2.0 GiB
Managed Node Group Kubernetes Count	3
Relational DBMS	PostgreSQL 14.7, 2 vCPUs, 1..0 GiB
Servicestreaming	Return 6, 2 vCPUs, .0.5 GiB
Region	Us-east-1
Infrastructure as Code	Terraform 1.3.0

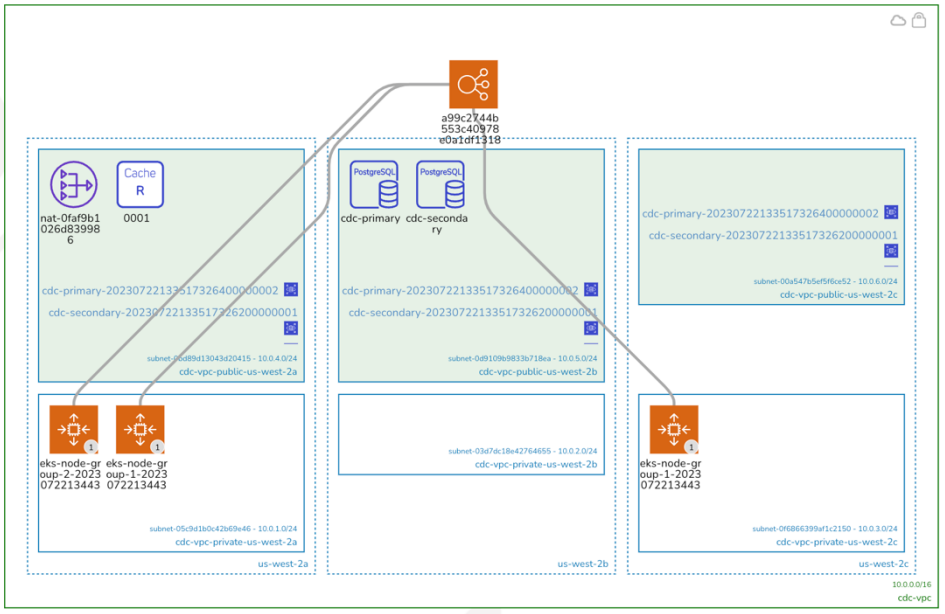


Figure IV.19 Solution Architecture in the Testing Process

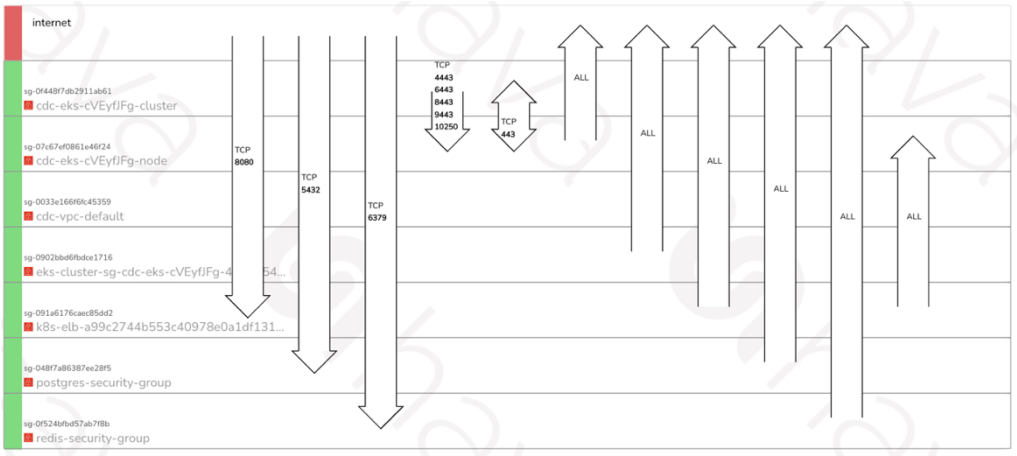


Figure IV.20 Security Group Test Environment

In the solution architecture design implemented on Amazon Web Services (AWS) in Figure IV.19, a number of services have been implementedmanaged service which aims to test various software and applications in an environment that is more ideal for the company that uses itcloud computing in business processes. Use of servicesmanaged service This enables companies to optimize and speed up testing processes, reduce infrastructure complexity, and improve system scalability and performance.

In this test, opening is carried outport on several services used to be accessible from the Internet. However, it needs to be emphasized that the openingport to the Internet is not recommended in terms of security, because it can increase the potential for attacks and provide an opportunity for unauthorized parties to try to access the system described in Figure IV.20.

As a more secure alternative, connections to AWS services should be private, where resources can only be accessed via a Virtual Private Network (VPN) or through a bastion host. The use of VPN ensures that every connection established with resources in the AWS environment is made through an encrypted path, thereby increasing communications security and protecting sensitive data from potential unauthorized parties.

Additionally, usebastion host (bridge hosts) provide a more secure way to access resources in an AWS environment.Bastion host is a server that is on a public network and functions as an entry gate to a private network. By usingbastion host, the user must first connect tohost this is over a public network, and only from there can they access resources in the highly regulated AWS environment.

By implementing a private connection via VPN orbastion host, companies can ensure that testing environments on AWS remain secure and protected from potential security threats. Additionally, this method also helps reduce the chances of security incidents occurring and minimize potential risksattack surface which can be exploited by attackers.

Openingport mentioned in the architectural design aims to simplify the testing process by allowing the initiation of connections toresource which exists. However, it is important to note that this access should always be managed carefully and only during the testing phase, not for production environments.

III.8 Testing Scenarios

In tool testing, three types of testing are carried out, namely: implementation testingblue-green deployment, tool functional testing, and tool performance testing

III.8.1 Blue-green Deployment Testing

The purpose of this testing is to ensure that the tool can function optimally and effectively in supporting the processblue-green deployment on the environmentmicroservice which requires changes to the database schema. In this test, the method usedblue-green deployment which has become a popular choice in ensuringzero-downtime and application continuity during updates or migration. Testing is performed in a Kubernetes environment. The following isKubernetes resource used in testing in Figure IV.21 and Table IV.13.

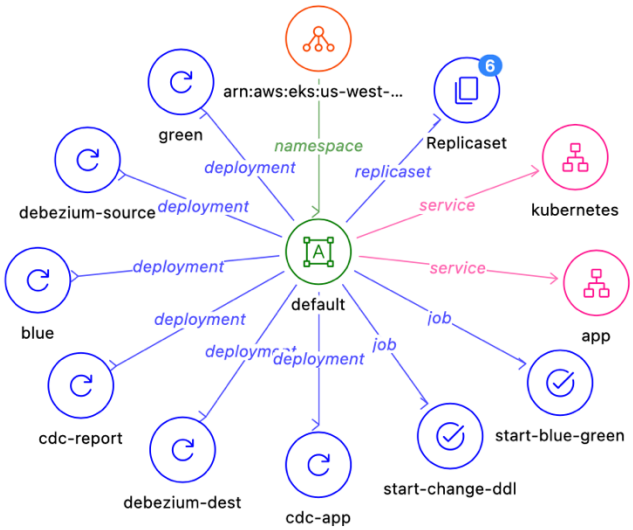


Figure IV.21 Kubernetes Resources in Testing Scenarios

Table IV.13 Description of Kubernetes Resources Used

No	Type	Description
default	namespace	Namespace in the deployed Kubernetes
app	service	Service which is used toexpose example application to the Internet.
cdc-app	deployment	Deployment which contains tools in starting condition
cdc-report	deployment	Deployment containing tools in producing conditionreport database replication status, aims to test the replication status when the tool is used
blue	deployment	Deployment which contains an old version of the sample application
green	deployment	Deployment which contains the new version of the sample application
debezium-source	deployment	Deployment which contains Debezium whichcapture data that comes from an old version of the database
debezium-hand	deployment	Deployment which contains Debezium whichcapture data coming from the new version of the database
start-change-ddl	job	Job which performs the command to initiate schema changes

Table IV.13 Description of Kubernetes Resources Used (Continued)

No	Type	Description
start-blue-green	job	Job which carries out the command to startblue-green deployment

In this final project, manufacturing automation is implementedresource using Terraform as Infrastructure as a Code. The decision to use Terraform has the main goal of simplifying and speeding up the creation processresource which is required in testing.

Terraform makes it possible to define infrastructure as code, which makes it easy to structure and manage resources through written scripts and configurations. Without using Terraform, implementationresource manually throughweb console oncloud provider will require many steps and may increase

With Terraform, you can quickly create Amazon EKS-based Kubernetes clusters, provision Elasticache for Redis, and manage PostgreSQL databases using RDS. Apart from that, Terraform is also used to do thisdatabase migration at the initial stage of the application by makingpublication andsubscription on the database.

The application of Terraform is not only limited to infrastructure, it also applies to provisioning applications and tools within a Kubernetes cluster. This provides consistency and efficiency across testing environments. Through the use of Terraform, the creation processresource can be automated and the risk of human error can be reduced. The following is the folder structure of Terraform used in Figure IV.22.

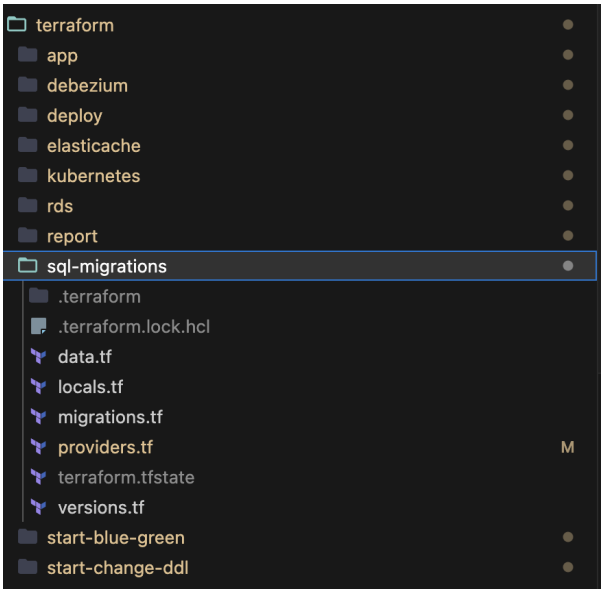


Figure IV.22 Test Environment Terraform Folder Structure

In order to test the tools that have been developed, several were maderesource which consists of AWS cloud computing services (Amazon Web Services). Makingresource cloud This was done to simulate implementation cases in companies that use the service a lotmanaged service from cloud computing and allows evaluation of the performance and efficiency of the tools that have been designed. This test environment requires several typesresource cloud which has the following cost details in Figure IV.23.

Name	Monthly Qty	Unit	Monthly Cost
aws_elasticache_replication_group.redis			
↳ ElastiCache (on-demand, cache.t3.micro)	730	hours	$12.41
module.blue.module.db_instance.aws_db_instance.this[0] Tags: Environment=dev, Owner=user			
↳ Database instance (on-demand, Single-AZ, db.t3.micro)	730	hours	$13.14
↳ Storage (general purpose SSD, gp2)	5	GB	$0.58
module.eks.aws_cloudwatch_log_group.this[0] Tags: Name=/aws/eks/cdc-eks-cluster_name-mock/cluster			
↳ Data ingested	Cost depends on usage: $0.50 per GB		
↳ Archival Storage	Cost depends on usage: $0.03 per GB		
↳ Insights queries data scanned	Cost depends on usage: $0.005 per GB		
module.eks.aws_eks_cluster.this[0]			
↳ EKS cluster	730	hours	$73.00
module.eks.module.eks_managed_node_group["one"].aws_eks_node_group.this[0] Tags: Name=node-group-1			
↳ module.eks.module.eks_managed_node_group["one"].aws_launch_template.this[0]			
↳ Instance usage (Linux/UNIX, on-demand, t3.small)	1,460	hours	$30.37
↳ EC2 detailed monitoring	14	metrics	$4.20
module.eks.module.eks_managed_node_group["two"].aws_eks_node_group.this[0] Tags: Name=node-group-2			
↳ module.eks.module.eks_managed_node_group["two"].aws_launch_template.this[0]			
↳ Instance usage (Linux/UNIX, on-demand, t3.small)	730	hours	$15.18
↳ EC2 detailed monitoring	7	metrics	$2.10
module.eks.module.kms.aws_kms_key.this[0]			
↳ Customer master key	1	months	$1.00
↳ Requests	Cost depends on usage: $0.03 per 10k requests		
↳ ECC GenerateDataKeyPair requests	Cost depends on usage: $0.10 per 10k requests		
↳ RSA GenerateDataKeyPair requests	Cost depends on usage: $0.10 per 10k requests		
module.green.module.db_instance.aws_db_instance.this[0] Tags: Environment=dev, Owner=user			
↳ Database instance (on-demand, Single-AZ, db.t3.micro)	730	hours	$13.14
↳ Storage (general purpose SSD, gp2)	5	GB	$0.58
module.vpc.aws_nat_gateway.this[0] Tags: Name=cdc-vpc-us-west-2a			
↳ NAT gateway	730	hours	$32.85
↳ Data processed	Cost depends on usage: $0.045 per GB		
Project total			**$198.54**

Overall total	**$198.54**

Figure IV.23 Cost Analysis of Testing Environment Requirements

In order to evaluate the performance of the developed tool, testing was carried out using an example application with two versions to simulateblue-green deployment.

Testing is carried out by following the work flow that has been determined in accordance with sequence diagrams that have been carefully designed. This workflow allows to simulate scenariosblue-green deployment by ensuring that new versions of the application can bedeploy and tested in parallel with the running version.

In this test, the "k6" tool is used asload generator to impose a load on the application's API at runtimeblue-green deployment taking place. "k6" acts as a load testing tool to perform requests constantly at a rate of 10request per second on the application API.

Report generated

Schema	Table	Source Records	Dest Records	Difference	Change Operation	Expected
public	day_menus	922	922	0		true
public	menus	922	922	0		true
public	transactions	919	919	0	Vertical Splitting	true
public	transaction_amounts	0	919	-919		true

Figure IV.24 Data Synchronization Status Results in the Database at the EndBlue-Green Deployment

The final results of this test show the success of the tool in carrying outblue-green deployment. Processdeployment running smoothly and without significant service disruption (downtime). Use the tool "k6" asload generator also proves that the application is able to handle the load well and provides a fast and consistent response during the processdeployment taking place can be seen in Figure IV.24.

III.8.2 Tools Functional Testing

Functional testing of tools in changing the data schema is a critical stage in validating and ensuring the quality and reliability of the tools that have been built. This test aims to test the tool's ability to manage data schema changes accurately and without compromising data integrity.

In this test, a series of trials are carried out by enteringquery which includes some changes to the data schema into the tool. Various types of data schema changes, such as adding columns, changing column data types, or deleting indexes, are simulated to test the reaction and response of the tools to the changes. Tests carried out to ensure that the tool can recognize and apply data schema changes correctly can be seen in Table IV.14.

Table IV.14 List of Change Operation Tests

No.	Change Operation Type	Testing Status	Information
1	addColumn	√	-
2	addNotNullConstraint	√	-
3	createTable	√	-
4	dropColumn	√	-
5	dropTable	√	-
6	modifyDataType	√	-
7	renameColumn	√	-
8	renameTable	√	-
9	HorizontalSplitting	√	-
10	VerticalSplitting	√	-
11	Adding Derived Columns	√	-
12	Adding Redundant Tables	√	-

In the test case it is focused on change operations that change the structure of the data schema. For changes that do not change the data schema, such as creating indexes and procedures, functional testing is not carried out. The following are several functional test cases of the tool.

1. Testing Rename Column When the Service Uses an Old Version

In this test case, there is a schema change in the form of changing the column names from previously there was a column in the rename_column table whose name was changed to message. The test results can be seen in Table IV.15.

Table IV.15 Table of Rename Column Test Results When the Service Uses the Old Version

No.	Query Enter	Query Output
1.	INSERT INTO rename_column (created_at,text,id) VALUES ('2023-02-08T08:19:41.816962Z','test',270077);	INSERT INTO public.rename_column (created_at,message,id) VALUES ('2023-02-08T08:19:41.816962Z','test',270077);
2.	DELETE FROM rename_column WHERE id=270077 AND created_at='2023-02-08T08:19:41.816962Z' AND text='test';	DELETE FROM public.rename_column WHERE id=270077 AND created_at='2023-02-08T08:19:41.816962Z' AND message='test';
3.	UPDATE rename_column WHERE id=270077 SET message='test'	DELETE FROM public.rename_column WHERE id=270077 AND created_at='2023-02-08T08:19:41.816962Z' AND message IS NULL; INSERT INTO public.rename_column (id,message,created_at) VALUES (270077,NULL,'2023-02-08T08:19:41.816962Z');

2. Drop Table Testing When the Service Uses an Old Version

In this test case, there is a schema change in the form of a change to delete the previous t table, which is changed to none. You can see the test results in Table IV.16.

Table IV.16. Drop Table Test Results Table When the Service Uses an Old Version

No.	Query Enter	Query Output
1.	INSERT INTO public.t (created_at, message, id) VALUES ('2023-02-08T08:19:41.816962Z', 'test', 270077);	

3. Testing Add Column When the Service Uses an Old Version

In this test case there is a schema change in the form of additional columns. However, adding a column at the timeservice in the old version there were no changesquery The test results can be seen in Table IV.17.

Table IV.17. Testing Add Column When the Service Uses an Old Version

No.	Query Enter	Query Output
1.	INSERT INTO public.add_column (id,message,created_at) VALUES (270077,'test','2023-02-08T08:19:41.816962Z');	INSERT INTO public.add_column (id,message,created_at) VALUES (270077,'test','2023-02-08T08:19:41.816962Z');
2.	DELETE FROM public.add_column WHERE message='test' AND created_at='2023-02-08T08:19:41.816962Z' AND id=270077;	DELETE FROM public.add_column WHERE message='test' AND created_at='2023-02-08T08:19:41.816962Z' AND id=270077;

4. Delete Test Using Primary Key When Service Uses Old Version

In this test case, if the table has a primary key then the primary key will be used in the delete process. You can see the test results in Table IV.18.

Table IV.18. Delete Test Using Primary Key When Service Uses Old Version

No.	Query Enter	Query Output
1.	DELETE FROM public.add_column WHERE id=270077;	DELETE FROM public.add_column WHERE id=270077;

5. Column Drop Testing When the Service Uses an Old Version

In this test case there is a schema change in the form of deleting a column from previously there was a column in the drop_column table called message. The test results can be seen in Table IV.19.

Table IV.19. Drop Column Test Results When the Service Uses an Old Version

No.	Query Enter	Query Output
1	INSERT INTO public.drop_column (id,message,created_at) VALUES (270077, 'test','2023-02-08T08:19:41.816962Z');	INSERT INTO public.drop_column (id,created_at) VALUES (270077,'2023-02-08T08:19:41.816962Z');
2	DELETE FROM public.drop_column WHERE id=270077 AND created_at='2023-02-08T08:19:41.816962Z' and message='test';	DELETE FROM public.drop_column WHERE id=270077 AND created_at='2023-02-08T08:19:41.816962Z';

6. Testing Rename Table When the Service Uses an Old Version

In this test case there is a schema change in the form of changing the table name from previously having a table called rename_table to rename_table_new. The test results can be seen in Table IV.20.

Table IV.20 Table of Rename Table Test Results When the Service Uses an Old Version

No.	Query Enter	Query Output
1.	INSERT INTO public.rename_table (id,message,created_at) VALUES (270077,'test','2023-02-08T08:19:41.816962Z');	INSERT INTO public.rename_table_new (id,message,created_at) VALUES (270077,'test','2023-02-08T08:19:41.816962Z');
2.	DELETE FROM public.rename_table WHERE id=270077 AND message='test' AND created_at='2023-02-08T08:19:41.816962Z';	DELETE FROM public.rename_table_new WHERE id=270077 AND message='test' AND created_at='2023-02-08T08:19:41.816962Z';

7. Horizontal Splitting Testing When the Service Uses an Old Version

In this test case there is a schema change in the form of:horizontal splitting In this schema change, the table initially has a table name, but a horizontal split is carried out so that all data with an ID of more than 2000 is entered in the table_id_after_2000 table. The test results can be seen in Table IV.21.

Table IV.21 Horizontal Splitting Test Results When the Service Uses an Old Version

No.	Query Enter	Query Output
1.	INSERT INTO public.table (id, message, created_at) VALUES (270077, 'test', '2023-02-08T08:19:41.816962Z');	INSERT INTO public.table_id_after_2000 (created_at, message, id) VALUES ('2023-02-08T08:19:41.816962Z', 'test', 270077);
2.	DELETE FROM public.table WHERE id=270077 AND created_at='2023-02-08T08:19:41.816962Z' AND message='test';	DELETE FROM public.table_id_after_2000 WHERE id = 270077 AND created_at = '2023-02-08T08:19:41.816962Z' AND message = 'test';
3.	INSERT INTO public.table (id, message, created_at) VALUES (10, 'test', '2023-02-08T08:19:41.816962Z');	INSERT INTO public.table (created_at, message, id) VALUES ('2023-02-08T08:19:41.816962Z', 'test', 10);
4.	DELETE FROM public.table WHERE id = 10 AND created_at =	DELETE FROM public.table WHERE id = 10 AND created_at =

No.	Query Enter	Query Output
	'2023-02-08T08:19:41.816962Z' AND message = 'test';	'2023-02-08T08:19:41.816962Z' AND message = 'test';

8. Vertical Splitting Testing When the Service Uses an Old Version

In this test case there is a schema change in the form of:vertical splitting in this schema change, the table initially contained a vertical_splitting table which had columns in the form of id, message, and message2. Then it's donevertical splitting and produces two tables vertical_splitting_derived_1 which contains the id and message, and vertical_splitting_derived_2 which contains the id and message2. The test results can be seen in Table IV.22.

Table IV.22 Vertical Splitting Test Results When the Service Uses the Old Version

No.	Query Enter	Query Output
1.	INSERT INTO public.vertical_splitting (id,message,message2) VALUES (270077,'test','test');	INSERT INTO public.vertical_splitting (id,message,message2) VALUES (270077,'test','test');INSERT INTO public.vertical_splitting_derived_1 (id,message) VALUES (270077,'test');INSERT INTO public.vertical_splitting_derived_2 (id,message2) VALUES (270077,'test');
2.	DELETE FROM public.vertical_splitting WHERE id=270077;	DELETE FROM public.vertical_splitting WHERE message2='test' AND id=270077 AND message='test';DELETE FROM public.vertical_splitting_derived_1 WHERE message='test' AND id=270077;DELETE FROM public.vertical_splitting_derived_2 WHERE id=270077 AND message2='test';

9. Testing Adding Redundant Tables When the Service Uses an Old Version

In this test case there is a schema change in the form of:adding redundant tables in this schema change, the table initially contained a vertical_splitting table which had columns in the form of id, message, and message2. Then it's doneadding redundant tables and produces two tables vertical_splitting_derived_1 which contains id and

message, and vertical_splitting_derived_2 which contains id and message2. The test results can be seen in Table IV.23.

Table IV.23 Test Results for Adding Redundant Tables When the Service Uses an Old Version

No.	Query Enter	Query Output
1.	INSERT INTO public.vertical_splitting (id,message,message2) VALUES (270077,'test','test');	INSERT INTO public.vertical_splitting (id,message,message2) VALUES (270077,'test','test'); INSERT INTO public.vertical_splitting (id,message,message2) VALUES (270077,'test','test');INSERT INTO public.vertical_splitting_derived_1 (id,message) VALUES (270077,'test');INSERT INTO public.vertical_splitting_derived_2 (id,message2) VALUES (270077,'test');
2.	DELETE FROM public.vertical_splitting WHERE id=270077;	DELETE FROM public.vertical_splitting WHERE id=270077; DELETE FROM public.vertical_splitting WHERE message2='test' AND id=270077 AND message='test';DELETE FROM public.vertical_splitting_derived_1 WHERE message='test' AND id=270077;DELETE FROM public.vertical_splitting_derived_2 WHERE id=270077 AND message2='test';

10. Testing Complex Cases When Services Use Old Versions

In this test case there are schema changes in the form of changing the data type from time(3) to time in the time_mili column, changing the name of the time_mili column to time2, changing the name of timestamo_mili to timestamp2, changing the data type of timestamp2 to timestamp (3), changing the name of the timestamp column to timestamp_with_time_zone2 , changing the data type timestamp_with_time_zone2 to timestamptz can be seen in the test results in Table IV.24.

Table IV.24. Table of Complex Case Testing Results in the Old Version

No.	Query Enter	Query Output
1.	INSERT INTO public.test_temporal (id, time, time_mili, time_with_time_zone, timestamp, timestamp_mili, timestamp_with_time_zone, date)	INSERT INTO public.test_temporal (date,timestamp2,timestamp_with_time_zone2,time,time_with_time_zone,timestamp_with_time_zone,time2,id) VALUES

No.	Query Enter	Query Output
	VALUES (45, 07:21:22.98069', 07:21:22.798', '02:47:09.863519', 07:21:23.682798069', 07:21:23.682798', '2023-02-10 07:21:22.798069', '2023-02-06');	('2023-02-10','2023-02-10T07:21:22.798Z'::timestamp, '2023-02-10T07:21:22.798069Z'::timestamptz,'07:21:22.798069','02:47:09.863519Z','2023-02-10T07:21:22.798069Z','07:21:22.798000'::time,45);
2.	DELETE FROM public.test_temporal WHERE id = 45 AND time = '07:21:22.798000' AND time_mili = 26482798 AND time_with_time_zone = 2023-02-10T07:21:22.798069Z' AND timestamp = 2023-02-10T07:21:22.798069Z' AND timestamp_mili = 2023-02-10T07:21:22.798069Z' AND timestamp_with_time_zone = '2023-02-10T07:21:22.798069Z' AND date = '2023-02-10;	DELETE FROM public.test_temporal WHERE time2='07:21:22.798000'::time AND timestamp2='2023-02-10T07:21:22.798Z'::timestamp AND id=45 AND time='07:21:22.798069' AND timestamp_with_time_zone2='2023-02-10T07:21:22.798069Z'::timestamptz AND time_with_time_zone='02:47:09.863519Z' AND timestamp_with_time_zone='2023-02-10T07:21:22.798069Z' AND date='2023-02-10';

11. Testing Rename Column When the Service Uses a New Version

In this test case there is a schema change in the form of changing the column names from previously there was a column in the rename_column table whose name was changed to message. The test results can be seen in Table IV.25.

Table IV.25 Table of Rename Column Test Results When the Service Uses a New Version

No.	Query Enter	Query Output
1.	INSERT INTO public.rename_column (created_at, text, id) VALUES ('2023-02-08T08:19:41.816962Z', 'test', 270077);	INSERT INTO public.rename_column (created_at,message,id) VALUES ('2023-02-08T08:19:41.816962Z','test',270077);
2.	DELETE FROM public.rename_column WHERE id = 270077 AND created_at = '2023-02-08T08:19:41.816962Z' AND text = 'test';	DELETE FROM public.rename_column WHERE id=270077 AND created_at='2023-02-08T08:19:41.816962Z' AND message='test';

12. Add Column Testing When the Service Uses a New Version

In this test case there is a schema change in the form of additional columns. However, adding a column at the timeservice in the old version there were no changesquery The test results can be seen in Table IV.26.

Table IV.26. Add Column Testing When the Service Uses a New Version

No.	Query Enter	Query Output
1.	INSERT INTO public.add_column (id, message, created_at) VALUES (270077, 'test', '2023-02-08T08:19:41.816962Z');	INSERT INTO public.add_column (id,message,created_at) VALUES (270077,'test','2023-02-08T08:19:41.816962Z');
2.	DELETE FROM public.add_column WHERE message = 'test' AND created_at = '2023-02-08T08:19:41.816962Z' AND id = 270077;	DELETE FROM public.add_column WHERE message='test' AND created_at='2023-02-08T08:19:41.816962Z' AND id=270077;

13. Rename Table Testing When the Service Uses a New Version

In this test case there is a schema change in the form of changing the table name from previously having a table called rename_table to rename_table_new. The test results can be seen in Table IV.27.

Table IV.27 Rename Table Test Results Table When the Service Uses a New Version

No.	Query Enter	Query Output
1.	INSERT INTO public.rename_table (id, message, created_at) VALUES (270077, 'test', '2023-02-08T08:19:41.816962Z');	INSERT INTO public.rename_table (id,message,created_at) VALUES (270077,'test','2023-02-08T08:19:41.816962Z');
2.	DELETE FROM public.rename_table WHERE id=270077 AND message='test' AND created_at='2023-02-08T08:19:41.816962Z';	DELETE FROM public.rename_table WHERE id=270077 AND message='test' AND created_at='2023-02-08T08:19:41.816962Z';

14. Horizontal Splitting Testing When a Service Uses a New Version

In this test case there is a schema change in the form of:horizontal splitting In this schema change, the table initially has a table name, but a horizontal split is carried out so that all data with an ID of more than 2000 is entered in the table_id_after_2000 table. The test results can be seen in Table IV.28.

Table IV.28 Horizontal Splitting Test Results When the Service Uses a New Version

No.	Query Enter	Query Output
1.	INSERT INTO public.table_id_after_2000 (id, message, created_at) VALUES (270077, 'test', '2023-02-08T08:19:41.816962Z');	INSERT INTO public.table (created_at,message,id) VALUES ('2023-02-08T08:19:41.816962Z','test',270077);
2.	DELETE FROM public.table_id_after_2000 WHERE id=270077 AND created_at='2023-02-08T08:19:41.816962Z' AND message='test';	INSERT INTO public.rename_table_new (id,message,created_at) VALUES (270077,'test','2023-02-08T08:19:41.816962Z');

15. Vertical Splitting Testing When a Service Uses a New Version

In this test case there is a schema change in the form of:vertical splitting in this schema change, the table initially contained a vertical_splitting_2 table which had columns in the form of id, message, and message2. Then it's donevertical splitting and produces two tables vertical_splitting_2_derived_1 which contains id and message, and vertical_splitting_2_derived_2 which contains id and message2. The test results can be seen in Table IV.29.

Table IV.29 Vertical Splitting Test Results When the Service Uses a New Version

No.	Query Enter	Query Output
1.	INSERT INTO public.vertical_splitting_2_derived_1 (id, message) VALUES (270077, 'test');	INSERT INTO public.vertical_splitting_2 (id,message) SELECT 270077,'test' WHERE NOT EXISTS (SELECT * FROM public.vertical_splitting_2 WHERE id=270077);UPDATE public.vertical_splitting_2 set id=270077,message='test' WHERE id=270077;
2.	UPDATE public.vertical_splitting_2_derived_1 SET message = NULL WHERE id = 270077;	UPDATE public.vertical_splitting_2 set id=270077,message=NULL WHERE id=270077;
3.	DELETE FROM public.vertical_splitting_2_derived_1 WHERE id = 270077;	UPDATE public.vertical_splitting_2 set message ISNULL WHERE id=270077;

16. Testing Complex Cases When Services Use New Versions

In this test case there are schema changes in the form of changing the data type from time (3) to time in the time_mili column, changing the name of the time_mili column to time2, changing the name of timestamo_mili to timestamp2, changing the data type of timestamp2 to timestamp (3), changing the name of the timestamp column to timestamp_with_time_zone2 , changing the data type timestamp_with_time_zone2 to timestamptz can be seen in the test results in Table IV.30.

Table IV.30. Table of Complex Case Test Results When the Service Uses a New Version

No.	Query Enter	Query Output
1.	DELETE FROM public.test_temporal WHERE date='2023-02-10' AND id=45 AND time_mili='07:21:22.798000'::time (3) AND timestamp='2023-02-10T07:21:22.798069Z'::timestamp AND timestamp_with_time_zone='2023-02-10T07:21:22.798069Z' AND time='07:21:22.798069' AND timestamp_mili='2023-02-10T07:21:22.798Z'::timestamp (3) AND time_with_time_zone='02:47:09.863519Z';	DELETE FROM public.test_temporal WHERE date='2023-02-10' AND id=45 AND time_mili='07:21:22.798000'::time (3) AND timestamp='2023-02-10T07:21:22.798069Z'::timestamp AND timestamp_with_time_zone='2023-02-10T07:21:22.798069Z' AND time='07:21:22.798069' AND timestamp_mili='2023-02-10T07:21:22.798Z'::timestamp (3) AND time_with_time_zone='02:47:09.863519Z';

III.8.3 Tool Performance Testing

Tool performance testing with gradual load to test,replication lag tools. To test the tool against transaction load in stages, the “pgbench” tool is used, which can be used to test performancequery on PostgreSQL. To create an ideal environment, testing the three scenarios is carried out in different environments, in this case, different database instances are used, andnamespace Kubernetes is different. The following is a diagram of the tool testing environment architecture in Figure IV.25.

Figure IV.25 Tool Performance Testing Environment

In the testing process the load was increased gradually from 10 RPS to 50 RPS and 100 RPS within 60 seconds. The transactions tested are operationsinsert, update, anddelete. During the testing process, every time you add a load, a 20 second pause is made so that there is a period for the tool to rest so that the results obtained are maximum. At each trial the RPS data will be recorded for further analysis. In addition to testing the tool with database schema changes, comparison tests were also carried out using the tool without any schema changes, and withlogical replication PostgresSQL default. The reason 100 RPS is the maximum limit in testing is because previous tests have shown a constant number and the limitations of the testing environment in the form of Redis which only has 500 MB RAM.

Table IV.31 Tool Performance Test Results with Scheme Changes

RPS	Replication Lag (ms)								
	Mean	Min	Max	STD Dev	50th pct	75th pct	90th pct	95th pct	99th pct
10	492	98	964	181	501	616	732	806	879
20	509	39	991	189	518	646	756	806	920
30	563	49	977	183	564	697	808	871	940
40	518	44	999	200	524	657	783	853	945
50	569	72	1389	207	567	717	833	885	1004
100	568	65	1031	186	569	700	815	874	966

Based on the data provided, the following are the conclusions from the performance testing results in Table IV.31:

1. RPS (Requests Per Second) is a measure of the level of transaction load tested on the system.
2. Replication Lag (ms) is the time delay in milliseconds between data replication on the system.
3. Column "Mean" represents the average replication delay observed at each RPS level. The average replication delay tends to stabilize around 536 ms.
4. Column "Min" and "Max" represents the smallest and largest replication delay values observed at each RPS level. The lowest replication delay varied between 39 ms to 98 ms, while the highest replication delay ranged from 964 ms to 1389 ms.
5. Column "STD Dev" (Standard Deviation) measures the spread of replication delay values around the average value. A lower standard deviation (191 ms) indicates that the values are relatively close to the average value.
6. The column "50th pct" (50th Percentile) or median represents the replication delay value below which 50% of the data points fall. The median replication delay consistently hovered around 540 ms.
7. The columns “75th pct,” “90th pct,” “95th pct,” and “99th pct” represent the replication delay values below which 75%, 90%, 95%, and 99% of the data points lie, respectively. As the percentile increases, the replication delay

value also increases, indicating that the replication delay tends to be higher for higher percentages of data.

Table IV.32 Tool Performance Test Results Without Schematic Changes

RPS	Replication Lag (ms)								
	Mean	**Min**	**Max**	**STD Dev**	**50th pct**	**75th pct**	**90th pct**	**95th pct**	**99th pct**
10	492	98	964	181	501	616	732	806	879
20	509	39	991	189	518	646	756	806	920
30	563	49	977	183	564	697	808	871	940
40	518	44	999	200	524	657	783	853	945
50	569	72	1389	207	567	717	833	885	1004
100	568	65	1031	186	569	700	815	874	966

From the data in the test results table in Table IV.32, the process of changing the schema on the data takes place very quickly so that the differencesreplication lag did not show significant or within-domain differencessub millisecond. This shows that the algorithm for changing the schema on the data is efficient.

Table IV.33 Results of PostgreSQL Logical Replication Performance Testing

RPS	Replication Lag (ms)								
	Mean	**Min**	**Max**	**STD Dev**	**50th pct**	**75th pct**	**90th pct**	**95th pct**	**99th pct**
10	1	1	40	3	1	1	1	2	12
20	2	1	349	10	1	1	1	2	7
30	2	1	34	1	1	2	2	2	5
40	2	1	34	2	2	2	2	3	9
50	3	2	157	6	2	2	3	3	8
100	3	2	47	1	2	2	3	4	8

However, the performance of tools is still far behindlogical replication which is owned by PostgreSQL. Visible to a minimumreplication lag the tools in Table IV.33 without any schema changes have an average of 61 ms, but with logical replicationreplication lag has an average of 1 ms. For the 99th percentile value, which is the ideal unit in testing because it can guarantee 99% performance of requests on

the tool, it has an average of 942 ms compared to postgresql logical replication which has a value of 8 ms. This is because the tool has two dependencies, namely Debezium and Redis which slow it downreplication lag. Debezium takes time to performformatting on change data in the database. Meanwhile, Redis, even though it stores data in RAM, still has bottlenecks in terms of I/O. Apart from that, the method used to enter data is to use executionquery SQL per-row data is certainly not efficient, especially if it existsbulk update nordelete using conditions that are notprimary key from the table.

BAB IV
CONCLUSIONS AND RECOMMENDATIONS

IV.1 Conclusion

The following are several points that can be drawn from the design and implementation of this tool:

1. For implementationblue-green deployment on the database the method can be usedchange data capture on the database.
2. Tools is designed with a based architectureevent by retrieving any database change data onqueue who Debezium had captured and change the data structure according to the configuration and execute the query to synchronize
3. The performance of the tool is fairly good with a 99 percentile of 849 ms, with this it can be guaranteed 99 percentreplication lag is below 849 ms.
4. Overall, the tools succeeded in fulfilling the purpose of the tools being made to help the processblue-green deployment applicationmicroservice. Performance testing results show that replication delays generally increase with higher RPS levels. The system tends to cope with lower RPS rates more efficiently, resulting in lower replication delays. However, as the load increases, the replication delay also increases, and the percentage of data experiencing higher replication delays also gets higher. Monitoring replication delays and addressing potential bottlenecks at higher RPS levels can help improve system performance and ensure timely data replication.

Whether or not replication delays are acceptable for a production database depends on the application being used. For some applications, a replication delay of 535 ms may be unacceptable, while for others it may be acceptable.

Here are some factors to consider in determining whether replication delays are acceptable:

1. The severity of the application. If the application is critical to business continuity, thenreplication lag the low becomes very important. However, if the application is not critical, then a higher delay may be acceptable.

2. Replicated data type. If the data is very sensitive or has a certain time limit, thenreplication lag the low becomes very important. However, if the data is not very sensitive or does not have a specific time limit, thenreplication lag higher may be acceptable.
3. Load ondatabase. Ifdatabase facing a heavy burden, thenreplication lag maybe it will be higher. However, if the load is ondatabase is not too heavy, then the replication delay will probably be lower.

Ultimately, the decision is whether the replication delay is acceptable or notdatabase production must be made on a case-specific basis. If the applications used are not critical and the data is not sensitive or has a specific time limit, then replication delays may be acceptable. However, if the application is critical or the data is sensitive or has a specific time limit, then replication delays may be unacceptable. The best way to determine whether replication delays are acceptable is to test the application with the specific data and load that will be used in production.

IV.2 Suggestion

There are several things that can be used as suggestions regarding the development of this tool. The following are several points taken in the tool development process.

1. For further development, it is recommended to use the methodchange data capture which ispush based. In developing this tool, Redis Stream is usedpull based. This causes an addition toreplication lag andresource who always works.
2. If you implement it further, you can use other methods to make the tool into somethingworker group so that the load received by the tool can be distributed to severalserver.
3. UsageInfrastructure as a Code like Terraform, really helpsdeploy-an applications that have many dependencies like this tool and reducehuman error.

BIBLIOGRAPHY

Ali, A. H., & Abdullah, M. Z. (2018). Recent Trends in Distributed Online Stream Processing Platform for Big Data: Survey. 2018 1st Annual International Conference on Information and Sciences (AiCIS), 140–145. https://doi.org/10.1109/AiCIS.2018.00036

Chen, L. (2018). Microservices: Architecting for Continuous Delivery and DevOps. 2018 IEEE International Conference on Software Architecture (ICSA), 39–397. https://doi.org/10.1109/ICSA.2018.00013

Delplanque, J., Etien, A., Anquetil, N., & Auverlot, O. (2018). Relational Database Schema Evolution: An Industrial Case Study. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 635–644. https://doi.org/10.1109/ICSME.2018.00073

Dijkstra, J.-J. (2021).Zero-downtime schema changes. University of Twente.

Gos, K., & Zabierowski, W. (2020).The Comparison of Microservice and Monolithic Architecture. 150–153. https://doi.org/10.1109/MEMSTECH49584.2020.9109514

Hesse, G., & Lorenz, M. (2016). Conceptual survey on data stream processing systems. Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS, 2016-January, 797–802. https://doi.org/10.1109/ICPADS.2015.106

K. Morris (2020). Infrastructure as Code, 2nd Edition. O'Reilly Media, Inc,.

Kimball, Ralph., & Caserta, J. (2004). The data warehouse ETL toolkit : practical techniques for extracting, cleaning, conforming, and delivering data. Wiley. https://www.wiley.com/en-us/The+Data+Warehouse%C2%A0ETL+Toolkit%3A+Practical+Techniques+for+Extracting%2C+Cleaning%2C+Conforming%2C+and+Delivering+Data-p-9780764567575

Ponniah, Paulraj. (2010). Data warehousing fundamentals for IT professionals (2nd ed.). John Wiley & Sons. https://www.wiley.com/en-us/Data+Warehousing+Fundamentals+for+IT+Professionals%2C+2nd+Edition-p-9780470462072

PostgreSQL. (2022). PostgreSQL: Documentation: 15: PostgreSQL 15.0 Documentation. https://www.postgresql.org/docs/current/index.html

Rudrabhatla, C. K. (2020). Comparison of zero downtime based deployment techniques in public cloud infrastructure. 2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 1082–1086. https://doi.org/10.1109/I-SMAC49090.2020.9243605

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). Database System Concepts. Dalam Database System Concepts Seventh Edition (Seventh Edition). McGraw-Hill. https://db-book.com/slides-dir/index.html

Tank, D. M., Ganatra, A., Kosta, Y. P., & Bhensdadia, C. K. (2010). Speeding ETL Processing in Data Warehouses Using High-Performance Joins for Changed Data Capture (CDC). 2010 International Conference on Advances in Recent Technologies in Communication and Computing, 365–368. https://doi.org/10.1109/ARTCom.2010.63

van Kampen, C.M. (2022).Zero Downtime Schema Migrations in Highly Available Databases. http://essay.utwente.nl/92098/