

MongoDB and IBM zSystems

Modernizing Mainframe Applications and Data

May 2022

1633 Broadway, 38th Floor, New York, NY, 10019 United States | mongodb.com

Content

Executive Summary 3

The Role of MongoDB in Mainframe Data and Applications Modernization Projects 3

MongoDB: The Application Data Platform 4

Operational Data Layer 6

MongoDB on Linux for IBM Z 6

Current and Future Architecture 6

Options for Organizing Data Replication 9

Non Real-time and DIY scenarios 10

Change Data Capture (CDC) based scenarios 11

Apache Kafka 12

MongoDB Sink Connector 12

Replication Architectures 12

Implementation Details for the IBM CDC Based Replication 17

Architecture Description 17

IBM CDC Remote Source 18

Implementation Details 19

Test Results 23

Conclusion & Outlook 24

Legal Notice 28

Executive Summary

Modernization of data and applications on the mainframe is a necessary priority for modern organizations. Building a MongoDB Operational Data Layer (ODL) is a pattern widely used to make critical mainframe data available for modern mobile and web applications.

MongoDB and IBM have collaborated to explain how IBM replication technologies can be combined with Kafka and the MongoDB Sink Connector to set up real-time replication from Db2 to MongoDB ODL on Linux on IBM Z, the pros and cons of available Change Data Capture (CDC) tools, and their usability for different use cases.

Understanding possible replication architectures, their tradeoffs and implementation of best practices is essential to helping make stronger technical decisions and accelerate mainframe data and application modernization efforts.

The Role of MongoDB in Mainframe Data and Applications Modernization Projects

The IBM zSystems (aka mainframe) remains a widely used enterprise computing platform, hosting key IT systems for the world's top industries and organizations, such as banks, airlines, insurers, etc. IBM zSystems run z/OS based workloads that rely on Db2 and other middleware subsystems like CICS, IMS or MQ (IBM z/OS subsystems) to process business critical transactions. These systems were typically developed with another era of use cases in mind. As such, digital transformation becomes an absolute imperative for enterprises to drive innovation. Modernization of applications and data is essential for client-centric businesses.

Considering an ever-increasing digitized and interconnected world, customer experience and real-time expectations are the priority for modern applications. Users anticipate information such as bank account status, payment and billing info, customer personalization, media updates, etc. to be constantly available both through web and mobile media. In modern banking, for example, the possibility of real-time updates at any time without visiting a bank is a simple customer expectation, but on the backend, this leads to an explosion of the number of operations that the existing mainframe-based systems have to handle.



To meet those business needs, IBM zSystems servers offer the capability of running Linux alongside z/OS on the same system. This allows customers to benefit from traditional IBM zSystems strengths, like availability and security that they are accustomed to from years of production experience, while using a modern operating environment and technologies not prone to skill shortages.

This is where MongoDB comes into play. Linux on IBM Z combined with MongoDB's capability to significantly shorten the development cycles and ensure performance and scalability, means MongoDB on IBM Z is a foundation to accelerate new digital initiatives. Building a MongoDB Operational Data Layer (ODL) for mainframe data is a widely used pattern in which the mainframe data is replicated in real-time to MongoDB, thus making it available for modern mobile and web applications.

MongoDB: The Application Data Platform

Since MongoDB's first database release, it has intentionally evolved into a general-purpose application data platform capable of handling virtually any diverse use case, across industries.

Before the introduction of the document model, database technology had not evolved much in the prior 30+ years. Instead of the traditional relational model, of storing data in columns and rows, MongoDB organizes information using documents written in JSON, allowing developers to introduce changes to the document schema on demand. This flexibility speeds up development cycles, allows application teams to deal with upcoming or changing requirements in an agile manner, and scale as needed without traditional rigidity. The beauty of documents is that they can be shaped to serve any kind of data structures like key-value pairs, graphs or nested structures, allowing for ingestion of diverse data types.

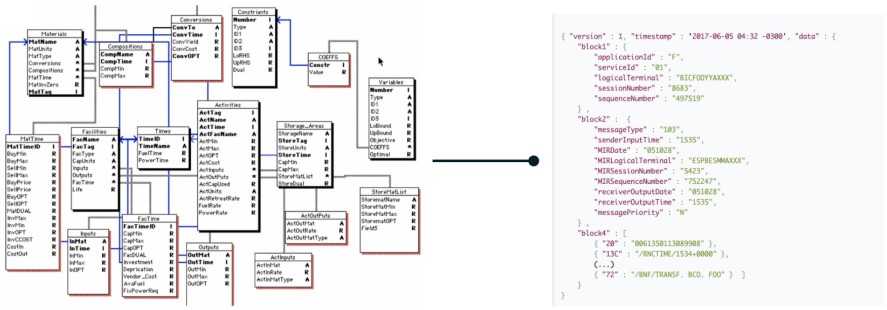
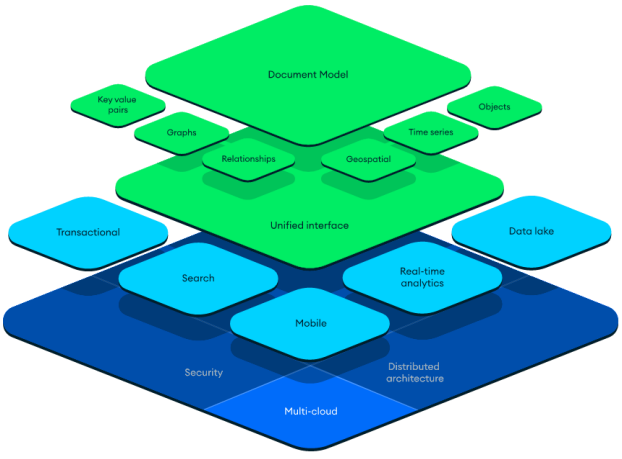


Figure 1: From relational to the document model

MongoDB provides drivers for all modern programming languages and allows developers to work with data as code, leveraging the MongoDB query API. This expressive language offers necessary CRUD (Create, Read, Update and Delete) operations and unique features, such as data aggregations and transformations, real-time analytics or full-text search capabilities. Whether MongoDB is run on a laptop, on a mainframe, or in the cloud or multi-cloud, the query API will stay the same and therefore empower seamless migrations across different platforms or cloud providers. In regard to availability, the underlying architecture of the MongoDB database assures it is available 24/7 and can be maintained and scaled to a globally available environment with ease.



MongoDB's Atlas platform offers MongoDB as a fully managed service across AWS, GCP and Azure with more than 80+ regions available worldwide that can be used to deploy MongoDB clusters. As of today, MongoDB manages 2 million+ clusters for their customers, taking care of typical maintenance tasks such as backups, updates or scaling in a fully automated way. MongoDB Atlas is a complete data platform that supports automated data tiering, full-text search based on Lucene, a built-in sync mechanism to mobile databases and many other features that help to further increase developer productivity.

The Operational Data Layer

An operational data layer (ODL) works as an intermediary between existing data sources and the consumers who need to access that data. It is an architectural data pattern that centrally integrates and organizes siloed data, making it available to consuming applications. ODLs are a key aspect of mainframe modernization for modern applications. It enables strategic initiatives, such as legacy modernization and data as a service, and use cases such as single view, real-time analytics, and mainframe data replication, that would otherwise be impossible relying on monolithic mainframe systems alone.

An ODL deployed on top of a mainframe works as a bridge to enable new, modern requirements that existing architectures cannot handle, without the risk and effort of fully eliminating or replacing legacy or mainframe systems. Adapting an ODL approach means reducing workloads on source systems, improving availability, reducing end-user response times, combining data from multiple systems into a single repository, and can serve as a base for restructuring a monolithic application as an array of microservices. The ODL is an answer to enterprises seeking modernization without starting over from zero.

MongoDB on Linux for IBM Z

MongoDB and IBM have a long and successful history of collaboration. MongoDB Enterprise Advanced versions 4.4 and 5.0 are fully supported on Linux on IBM Z by RedHat Enterprise Linux 7 and 8, allowing organizations that are heavily invested in IBM zSystems to modernize their workloads using existing infrastructure and skills investments. Having Linux instances with MongoDB close to the source z/OS transactional data also allows for lower network latencies, simplified network configuration and a better security posture as the data never leaves the secured mainframe environment. IBM zSystems high CPU frequencies, large RAM volumes and powerful I/O capacity make IBM zSystems a great choice for mission critical workloads. While MongoDB's Ops Manager fully supports managing and backing up MongoDB deployments running on IBM zSystems, it must be deployed on x86 infrastructure. MongoDB internal databases used by Ops Manager (application and backup) can also run outside of the IBM zSystems Infrastructure.

Current and Future Architecture

A commonly used architecture with all queries going against z/OS is shown in the figure below. In the architecture shown, the modern web or mobile application backends running on Linux are accessing the data in Db2 either directly or through an API Gateway for both read and write requests. The applications can also call CICS “transactions” (pre-programmed pieces of business logic typically implemented in Cobol) through the same API gateway, which in turn access data in Db2. No matter which method is used, the rise in the number of transactions directly translates into raising the “MIPS” costs on the z/OS side (the monthly software bill is defined by the maximum 4-hour rolling average CPU consumption within that month).

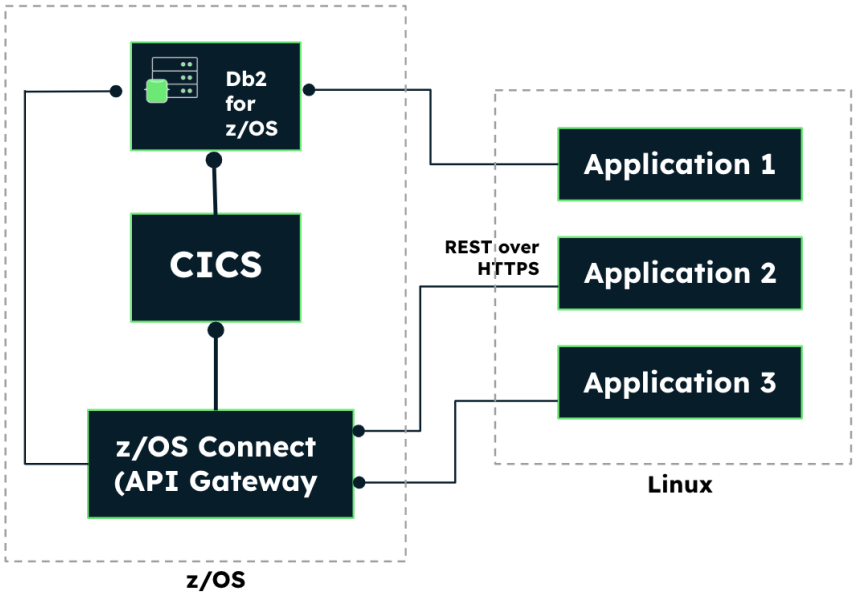


Figure 2: Queries going against z/OS

Now, the architecture of the MongoDB ODL based approach can be seen in the figure below. The data from Db2 for z/OS is replicated (the exact details of how this can be done will be addressed further along) in real time to MongoDB running on Linux for IBM Z. The read requests originating from the applications are routed to the MongoDB, while the write requests are still executed against the original Db2 for z/OS subsystem or forwarded to CICS through the z/OS Connect API gateway. Since the read requests typically represent the vast majority of all requests, this

architecture avoids the bulk of the “MIPS” costs generated by the rise in the number of requests.

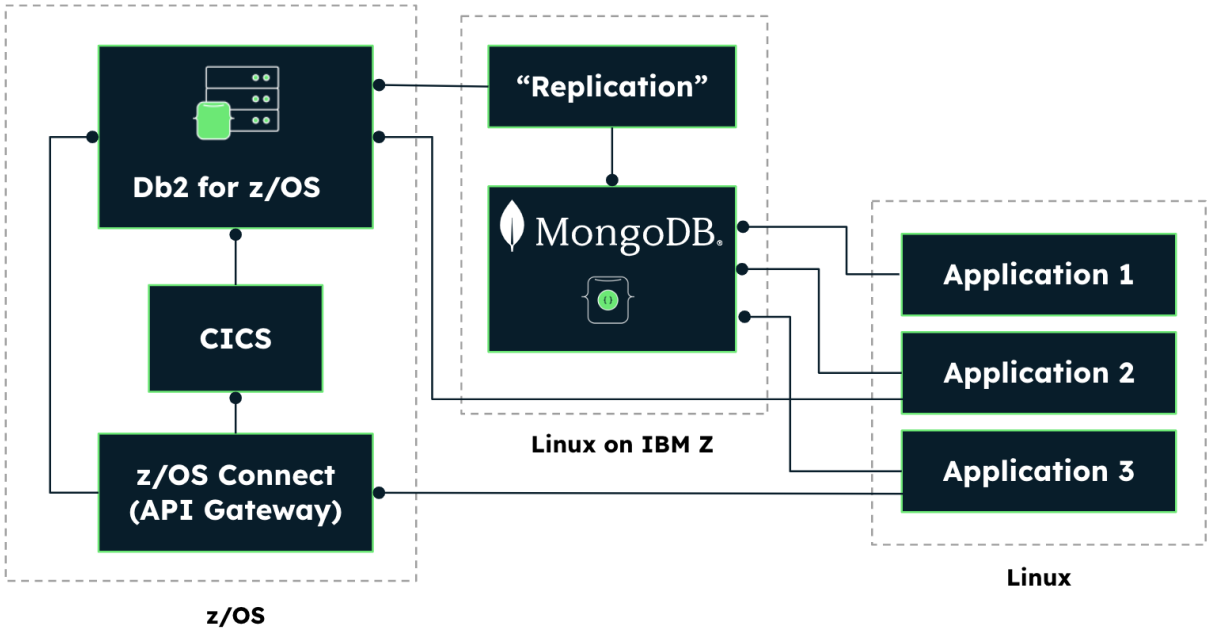


Figure 3: Architecture of the MongoDB ODL based approach

This architecture can be later extended by adding a “reverse” replication from MongoDB to Db2 for z/OS and replacing the CICS transactions using the strangler pattern, which puts the CICS system behind an intermediary facade. The target state architecture is shown in the figure below.

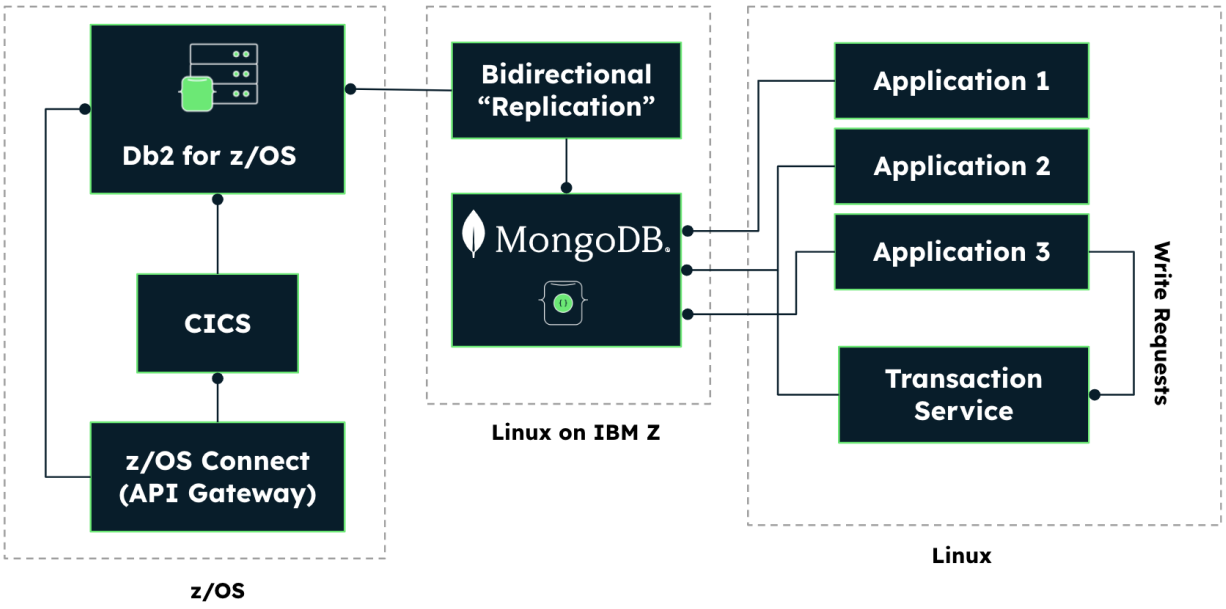


Figure 4: “Reverse” replication from MongoDB to Db2 for z/OS

In this architecture, the applications are no longer constrained by the z/OS environment and are free to either write directly against MongoDB or use the CICS transaction replacing service, whichever approach best fits the use case. Then, the MongoDB data is replicated back to Db2 for z/OS to be used by other services/applications that depend on Db2.

Options for Organizing Data Replication

The target architecture described in the previous section heavily depends on an efficient way to set up replication from Db2 for z/OS to MongoDB. The following metrics play a key role in replication and influence the choice of the replication architecture:

Latency is the time that passes from the moment that the changed row has been committed in the Db2 for z/OS database until this row is inserted into the target MongoDB database.

Throughput is the number of changes that have been replicated from Db2 for z/OS to MongoDB in a given time period.



CPU overhead measures the additional CPU resources that are required on the z/OS side to capture changes while meeting the latency goal. This parameter is extremely important since the licensing model used for z/OS based software is heavily influenced by the CPU costs. The amount that a customer is paying for mainframe software is dependent on their maximal 4-hour rolling average CPU consumption in a given month.

Ultimately the goal in building the replication pipeline is to minimize **latency** and **CPU overhead** while meeting the **throughput** requirements.

The replication pipeline from Db2 for z/OS to MongoDB can be broken down into the following steps:

1. Extract the information from Db2 on z/OS. Latency and CPU overhead are very important here as the changes need to be captured on the z/OS side, which can lead to additional costs for all software components installed on the source operating system.

2. Transform the extracted information and route it to the target MongoDB database. Often the extracted information needs to be transformed in some way. For example, data type conversions could be required. Since embedding is typical in MongoDB, there is often a need to join change streams from multiple tables to form a single stream of documents. For instance, we could prefer to store all a customer's orders within the customer document rather than having two separate document collections, one for customers and another for orders.

3. Sink the record to the target MongoDB database.

Writing the information to the target MongoDB database is not the only way to utilize the captured changes. The stream of changes (potentially transformed) can also act as a base for event driven applications.

Non Real-time and DIY Scenarios

Non Real-time Replication



Periodic unloads/loads can be used to organize non-real time replication. IBM provides efficient tools for unloading large amounts of data from Db2, and **`mongoimport`** could be used to load the data into target MongoDB. ETL tools, both open source (Talend) and software vendor provided (Informatica, Hitachi Vantara, etc.) can also be used to implement non real-time replication.

The main challenge in implementing such unloads is to identify the changed rows that need to be unloaded. One possible approach is to add an additional column to each replicated table and implement an insert and update trigger to populate this column on every mutation. Identifying and replicating deletes is a bit more involved, as a new table needs to be created to store at least the primary key columns of the deleted rows with the trigger that would copy the deleted rows into this table. Executing a trigger on every mutation causes additional CPU overhead and maintaining replication-related database objects is an additional burden for Db2 DBAs.

DIY Streaming

Since Db2 for z/OS supports Java stored procedures, one could implement a trigger that would call a Java stored procedure that would push the changes to MongoDB (either directly or through an intermediate system like Kafka). The major downside of this approach is the necessity to implement and maintain the stored procedures and triggers. Having a trigger in the synchronous code path also poses the threat of negatively affecting the original transactions. If the trigger fails the source transaction will not be executed. Triggers and stored procedures also create CPU overhead, which makes this method expensive on z/OS. The upside of this method is that it can be implemented without any additional products from any vendor and can guarantee zero latency.

Change Data Capture (CDC) Based Scenarios

IBM Db2 for z/OS uses write ahead logging in which all changes are written to the transaction logs for recovery purposes. Db2 for z/OS also provides APIs that allow listening applications to be notified when the changes have been written to transaction logs and APIs to read the transaction log entries. These APIs are used by IBM replication solutions to efficiently extract the stream of changes. Extracting them from the transaction logs has the advantage of being outside the synchronous code path and so does not affect the source transactions. Reading



the transaction logs is also less resource intensive than triggers, which makes log-based approaches the optimal choice for most customers. However, log-based replication can be built only with special tools, for example, those offered by IBM.

IBM currently offers three replication products that can be used to extract Db2 transaction log entries to build near-real time replication pipelines:

> **SQL Replication**: A capture process running on z/OS reads the Db2 transaction logs, extracts the changes and stores them in special change data capture tables (one per source table). From these tables the data can be picked up by the apply processes that transfer it to the target database.
>
> **Q Replication**: Works similarly to SQL replication with the exception that the read changes are transferred to the IBM MQ Queues with every change record being encoded as an XML document. Apply processes read from the queues and transfer the information to the target system.
>
> **CDC Replication**: Based on source and target agents that communicate directly. CDC supports not only a wide set of relational databases, but also other systems, for example Apache Kafka.

Apache Kafka

Apache Kafka is a publish/subscribe based durable messaging system that has massively gained in popularity over the past years. Due to its distributed nature, it efficiently scales horizontally and vertically and is used as a base for a wide range of integration use cases. Kafka Connect API provides a framework for developing importers (source connectors) and exporters (sink connectors) from Kafka. Many hundreds of open source and non open source connectors for Kafka Connect have been developed over the years, including for Db2 and MongoDB. Combined with Kafka's Streaming APIs and ksqlDB, which enable powerful transformation capabilities within Kafka, Kafka is a perfect base for replicating data from Db2 to MongoDB.

Kafka is not only the transportation and transformation mechanism, but can serve a much larger role by enabling event driven architectures for the organization. The changes that are replicated from Db2 for z/OS can be treated as business events that open a set of new possibilities to build reactive, real-time applications.

MongoDB Sink Connector

MongoDb Kafka Sink Connector is a Confluent-verified Kafka connector that allows users to sink data from Kafka topics to MongoDB databases. The connector is developed and supported by MongoDB and uses the Kafka Connect APIs. Kafka Connect APIs provide the required primitives to build scalable and reliable integrations between various data sources based on Kafka.

Replication Architectures

Depending on the available IBM products and existing infrastructure, the following replication architectures could be used:

IBM Q Replication based replication: IBM Q Replication captures the changes from Db2 transaction logs and forwards them to an MQ Queue. The Confluent MQ Source Connector extracts the change records from the MQ queue and forwards them to a Kafka topic. As soon as the data is in a Kafka topic, the full power of the Kafka and Confluent ecosystem can be used to transform the data. MongoDB Sink Connector for Kafka ensures efficient sink of Db2 data from a Kafka topic to MongoDB. This method is a good choice when existing IBM MQ based replication infrastructure is already in place. The architecture of this approach is shown in the following figure.

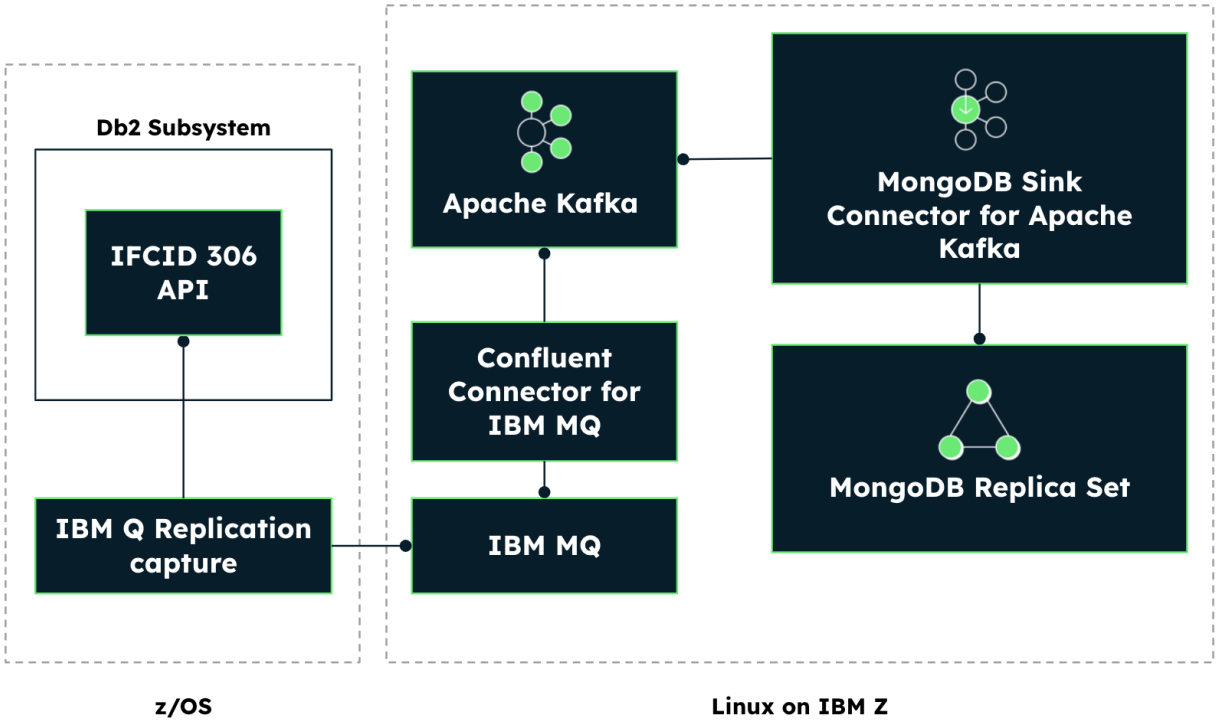


Figure 5: IBM Q Replication based replication

IBM SQL Replication based replication: The capture agent of SQL replication reads the Db2 transaction logs and saves the change data in special change data (CD) tables that resemble the structure of original tables that are being replicated. Unit of work information is then joined with CD tables to filter out the rollbacks and produce consistent change data (CDD) tables. The data in CDD tables can be used by the standard Kafka JDBC Connector (pull based) to feed the changes to a Kafka topic. This option can be useful for customers that already use IBM SQL Replication. The architecture of this approach is shown in the following figure.

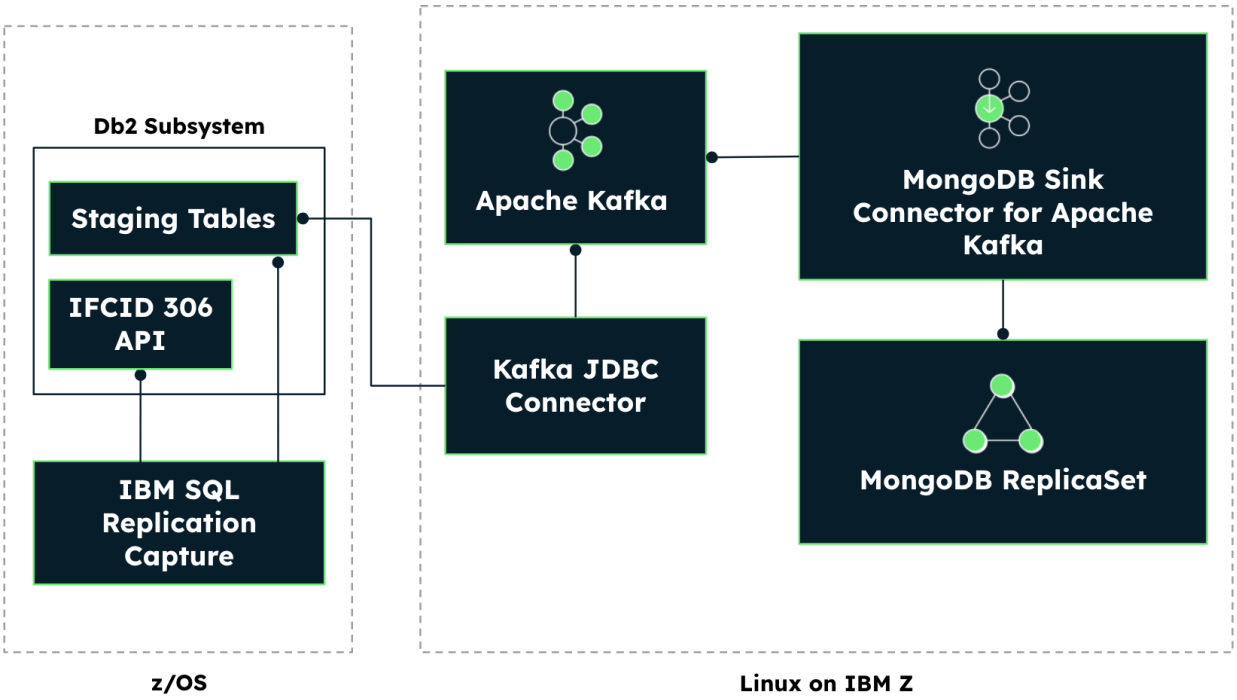


Figure 6: IBM SQL Replication based replication

IBM CDC Replication based replication: The CDC Db2 source agent reads the Db2 transaction logs and transfers the changes to the CDC Kafka target agent, which in turn pushes the changes to a Kafka topic. MongoDB Sink Connector reads the changes from the Kafka topic and applies them to MongoDB. This is the most direct (i.e., near real-time) route to Kafka. An important concept to understand is that CDC reads the individual operations (inserts, updates, deletes) from the log and then builds the transactions (URs) on the source system. No data is sent to the target system until the full transaction is built. Multiple concurrent transactions can be built; whenever a transaction is complete it will be sent to the target. CDC will process the data in a transactionally consistent manner. The architecture of this approach is shown in the following figure.

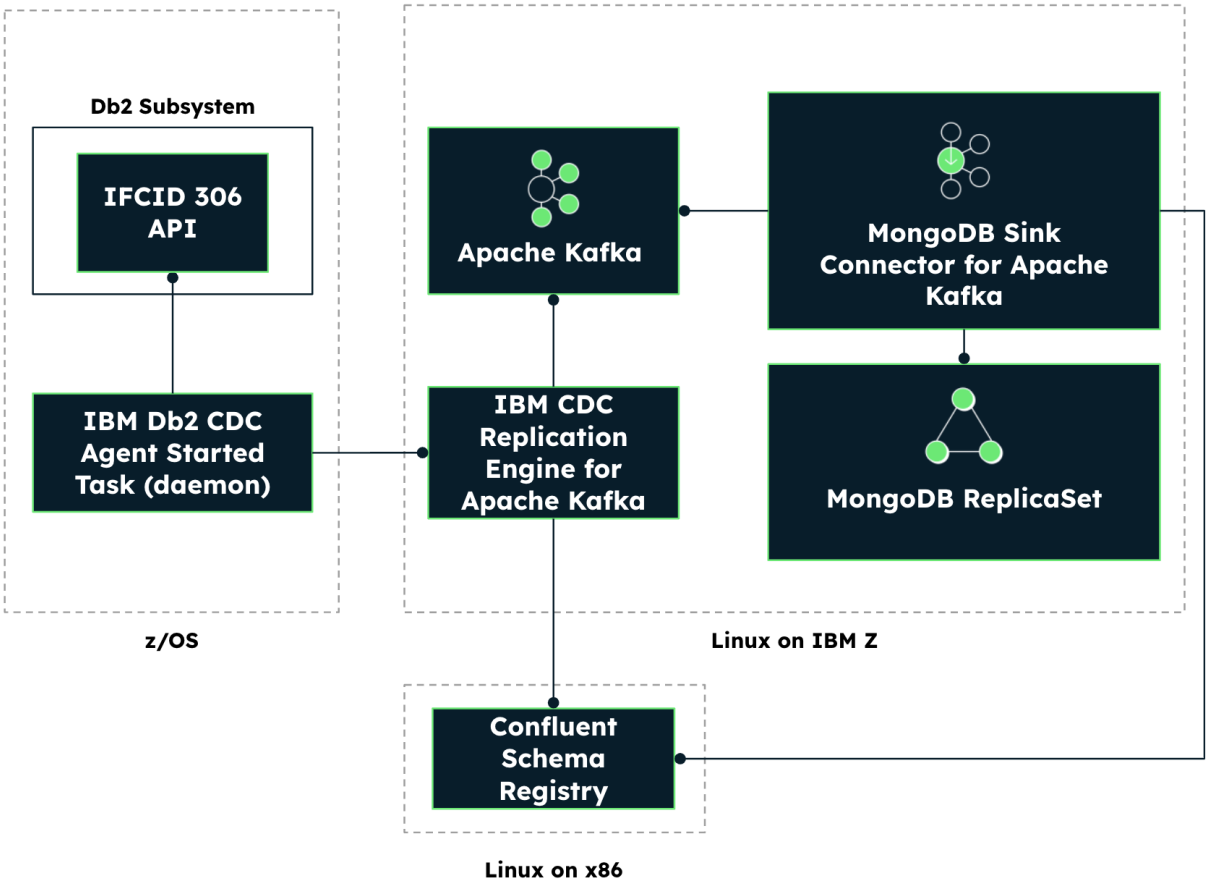


Figure 7: IBM CDC Replication based replication

IBM CDC with user exit based approach: The capture part is similar to the CDC based approach, but rather than pushing the changes to the Kafka topic, the target agent can call a custom written user exit that does a direct insert into MongoDB. Sample code for such a user exit can be obtained from IBM. However, there is no official support provided for it. The architecture of this approach is shown in the following figure.

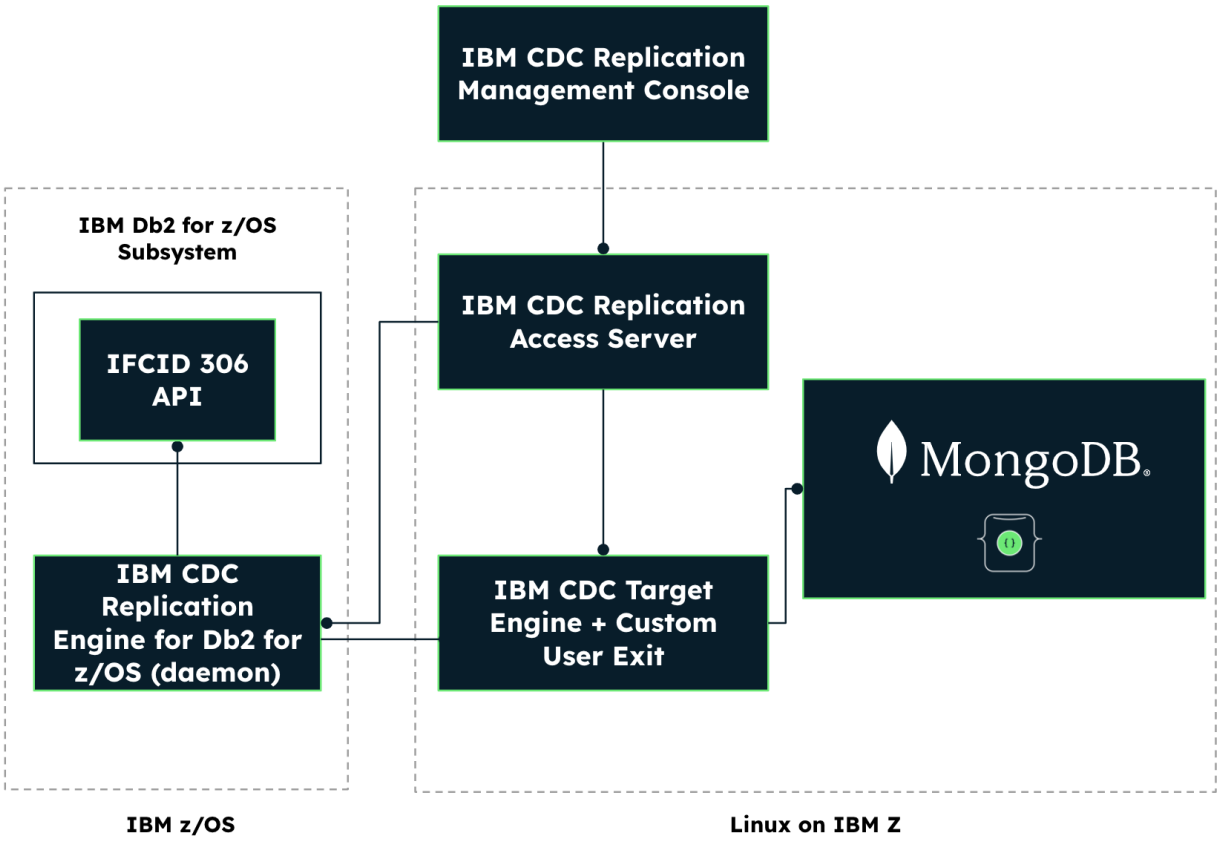


Figure 8: IBM CDC with user exit approach

A comparative table for various replication methods is shown in the following table:

Method	Push or Pull	Latency	Complexity	Officially supported?
Q Replication	Push	Low	High	Yes
SQL Replication	Pull	High	Medium	Yes
CDC Replication	Push	Low	Low	Yes
CDC with user exit	Push	Lowest	Low	No

The IBM CDC Replication based approach is the simplest to implement and offers low latency and highest flexibility by pushing the changes to Kafka topics. Commercial

support is available from the software vendors for all parts of this solution, which makes it a good choice for new replication implementations.

Implementation Details for the IBM CDC Based Replication

Architecture Description

As shown in the previous section, the most universal architecture to organize the real time replication from Db2 for z/OS to MongoDB is based on IBM InfoSphere Data Replication for CDC and Apache Kafka as transport. The components required to set up this architecture are shown in the following diagram.

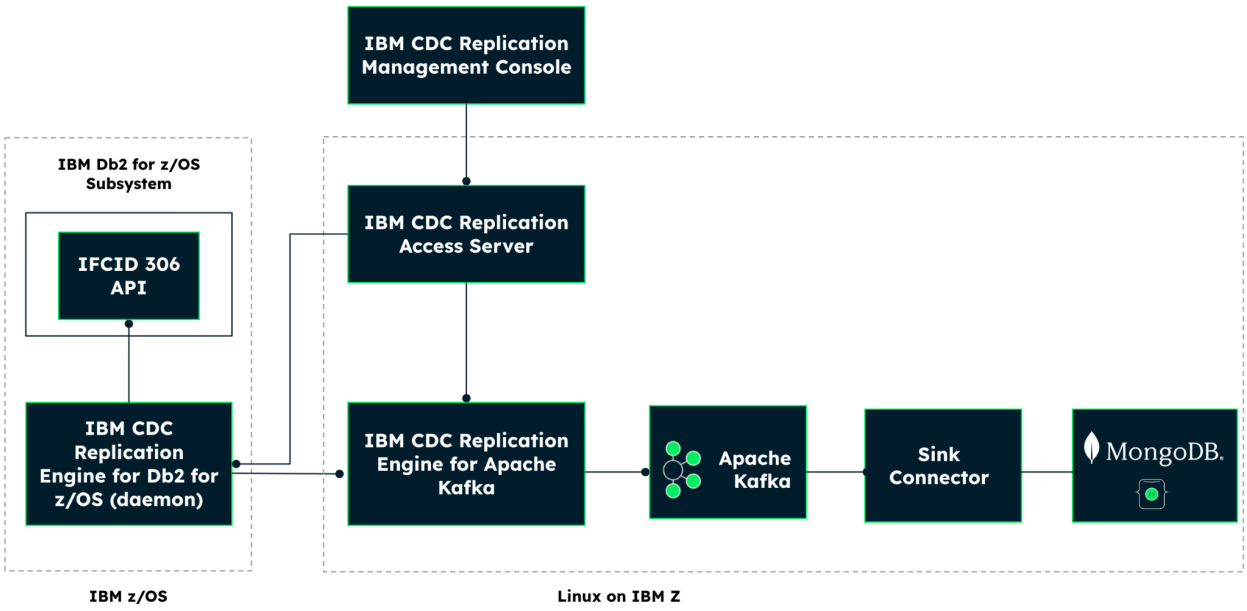


Figure 9: IBM InfoSphere Data Replication for CDC and Apache Kafka as transport

IBM CDC Replication Access Server: Stores all metadata and manages access to the replication agents. CDC works with subscriptions that link the source Db2 tables to target Kafka topics. Source agent definitions, target agents definitions, CDC administrators and their permissions are also stored on the access server.



IBM CDC Replication Management Console: A central point of control for replication. Allows defining replication agents, replication administrators, their permissions on agents and finally the subscriptions. The management console also enables the event log analysis as well as replication monitoring.

IBM CDC Replication Engine for Db2 for z/OS (Source Agent): Captures the data on the z/OS side and transfers the captured changes to the IBM CDC Replication Engine for Kafka.

IBM CDC Replication Engine for Kafka (Target Agent): Receives the changes captured on the source agent and pushes them to the Kafka topics. Changes for each source table are inserted to a separate Kafka topic.

Apache Kafka Cluster: An instance of Apache Kafka that acts as the sink for IBM replication. As soon as the captured changes are in Kafka they can be transformed using the Kafka built in capabilities, like Kafka Streams or ksqlDB. For example, multiple streams (topics) can be merged into one, data type conversions can be executed and so forth.

MongoDB: The target database that receives the final stream of documents.

In this architecture, every component besides the CDC source agent runs outside of z/OS. The CDC source agent runs as a started task (daemon) on z/OS and uses Db2 IFI APIs to get notified when the new changes have been made available in the transaction logs and to read these changes. The agent must be SMPE installed, configured and maintained by the z/OS administrators, which is an additional effort for often understaffed teams. More importantly, the CDC source agent is responsible for the capture, caching the captured log records, filtering, encoding conversions, etc. The associated CPU costs are paid on z/OS causing the MIPS bill to grow. In order to alleviate these challenges, IBM offers CDC Remote Source, which processes the majority of capture processing on Linux.

IBM CDC Remote Source

With IBM CDC Remote Source, the Db2 source agent, called the Remote Agent, is deployed outside of z/OS (on Linux). The actual capture is organized through an external stored procedure within Db2. The compiled code for this external stored procedure is deployed to an APF-authorized library (trusted library with code) on z/OS. The capture

agent installation is fully automated and is done on Linux, **`ssh`** is used to copy the APF-authorized library to z/OS.

The only actions that are required on z/OS are APF-authorizing the target library for the stored procedure executable and defining a WLM application environment (a separate process to execute the capture stored procedure). These actions need to be executed only once; future updates of the agent executed on the Linux environment replace the stored procedure executable in z/OS using **`ssh`**.

The architecture with the remote agent is shown in the following figure.

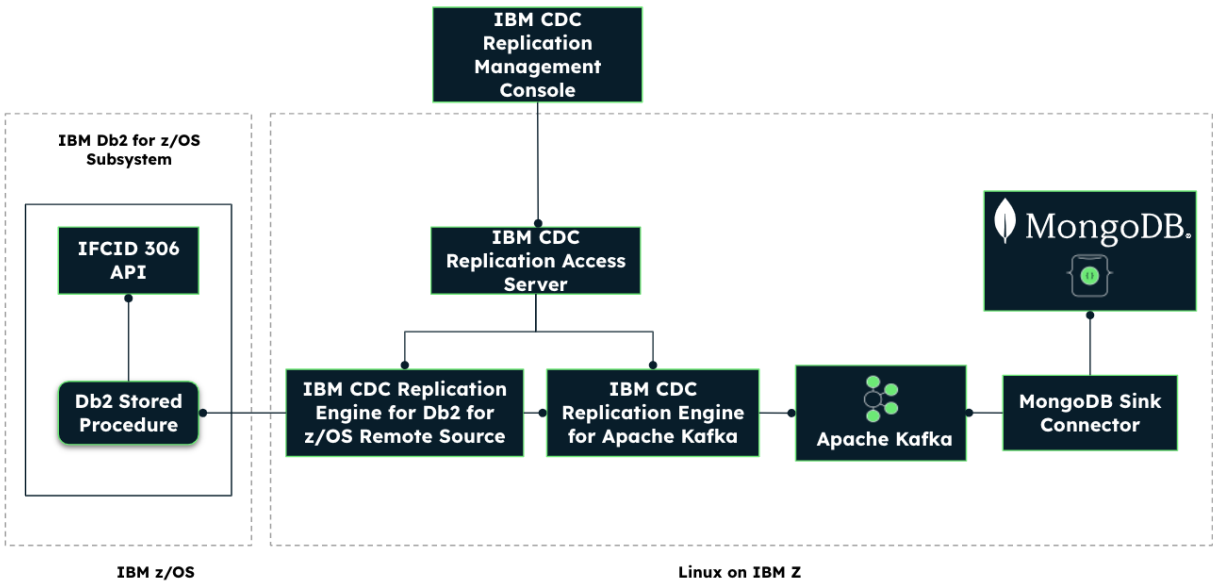


Figure 10: IBM's alternative approach where the Remote Agent is deployed outside of z/OS

Implementation Details

The implementation of the CDC Remote Source architecture provides the simplest approach that requires the fewest changes on the z/OS side and consumes minimal CPU resources there. We assume that the source Db2 subsystem and the target MongoDB deployment are already set up.

Installation Instructions



The following steps need to completed for the most basic setup:

1. Install the IIDR CDC Replication Access Server and Management Console using the official documentation.
2. Prepare the z/OS Environment for the remote capture stored procedure using the following instructions.
3. Install the IIDR CDC Replication Engine for Db2 z/OS Remote Source and IIDR CDC Replication Engine for Kafka using the official documentation.
4. Install Apache Kafka using, for example, the official Kick-Start.
5. Install the MongoDB Kafka Connector using the official documentation.

Configuring the Subscriptions

IIDR CDC Subscription defines the mapping of the source tables to target Kafka topics. Before defining the subscription, source and target datastores (agents) need to be defined in the management console. The exact steps can be found in official documentation.

Defining a subscription that maps source Db2 tables to Kafka topics is straight forward. The name of the topic in Kafka will be generated based on the subscription and table name. Key columns should be defined to be used as a key in the corresponding Kafka topic.

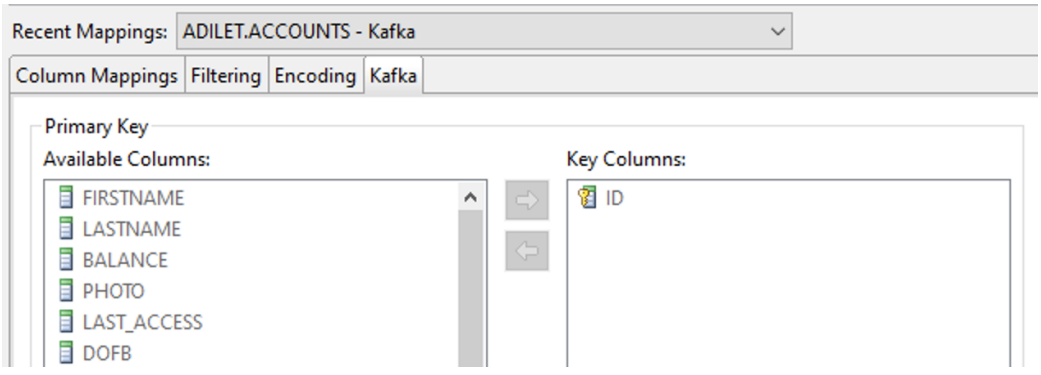


Figure 11: Defining key columns to be used as key value in every Kafka message

One Kafka agent can work with multiple Kafka destinations, but a single subscription can work with only one target Kafka cluster. It can be defined by right clicking on the subscription and selecting the 'Kafka Properties' menu item.

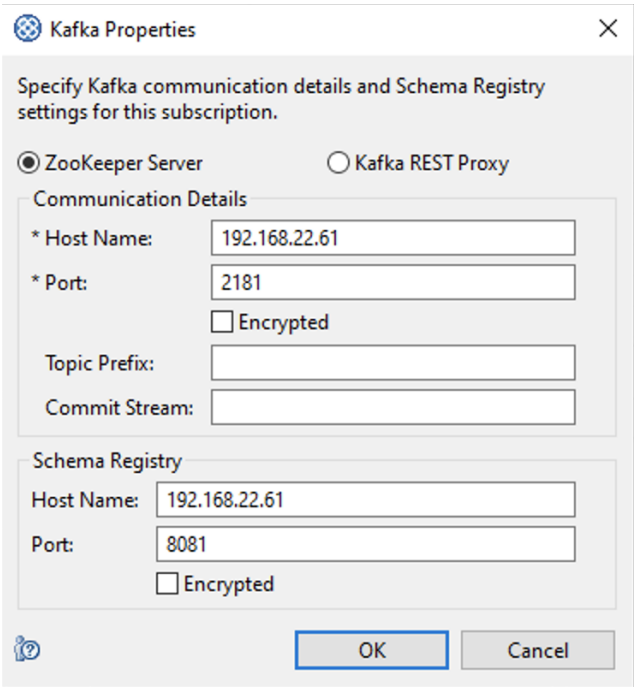


Figure 12: Defining the Kafka properties for a subscription

There are several formats in which the IIDR CDC Kafka engine can place changes to the target Kafka topic. The format used is controlled by setting the Kafka Custom Operation Processors (KCOP) for a subscription. KCOP is a java class that defines how the CDC message will be formatted before it is inserted into a Kafka topic. A set of KCOPs is delivered as part of IIDR CDC product, but customers have the possibility of implementing custom KCOPs. KcopJsonFormatIntegrated is the most basic option that formats the changes captured on the Db2 side as JSON messages and requires no schema. KcopDefaultBehaviorIntegrated is a more advanced option that formats the changes as avro messages and thus requires a compatible schema registry. There are several schema registries available on the market, including ones capable of running on IBM zSystems. The exact KCOP used is defined by right clicking on the subscription and selecting the 'User exits...' menu item.

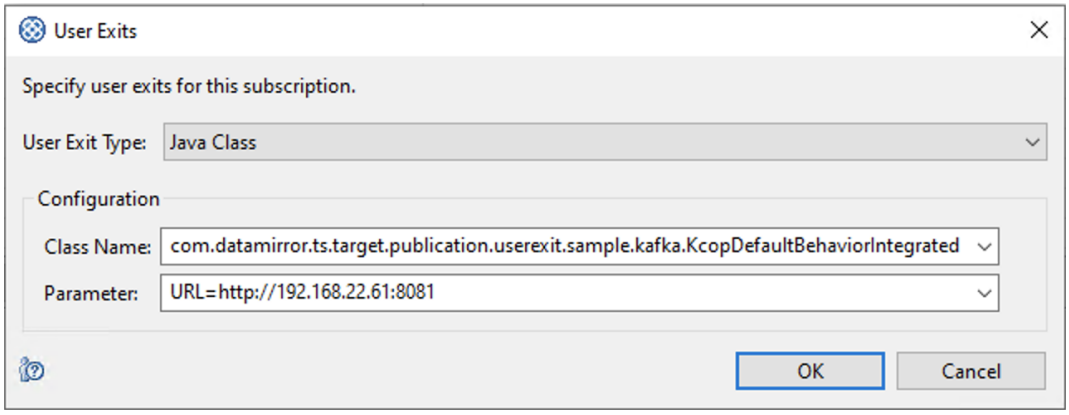


Figure 13: Specifying the KCOP for the subscription.

The KcopDefaultBehaviorIntegrated is the best option, since the avro format preserves the data type information and is more efficient compared to a string based JSON format.

MongoDB Sink Connector Configuration

The MongoDB Sink Connector is responsible for writing the messages out of the Kafka topics into target collections. Since the connector is based on Kafka Connect APIs it can be scaled to multiple physical processes running on multiple machines. The official documentation is available to learn more about the steps required to install the connector and the possibilities of configuring the connector, including running multiple instances of it to handle peak workloads. The installation consists of downloading the code (jar files) of the MongoDB Kafka connector and the MongoDB driver. The most basic sink connector configuration is shown below.

```
name=mongo-transactions-sink
connector.class=com.mongodb.kafka.connect.MongoSinkConnector
tasks.max=1
topics=kafka185.test1.sourcedb.mongosrc.adilet.transactions (1)
database=kafka_sink (2)
collection=transactions_avro (3)
connection.uri=mongodb://<username>:<password>@node1:27017,node
2:27017,node3:27017/myFirstDatabase?ssl=true&replicaSet=rs1&aut
hSource=admin&retryWrites=true&w=majority (4)
delete.on.null.values=true (5)
```

```
document.id.strategy=com.mongodb.kafka.connect.sink.processor.i
d.strategy.ProvidedInKeyStrategy (6)
# transformations (7)
transforms=renamekey,renamevalue
transforms.renamekey.type=org.apache.kafka.connect.transforms.R
eplaceField$Key
transforms.renamevalue.type=org.apache.kafka.connect.transforms
.ReplaceField$Value
transforms.renamekey.renames=ID:_id
transforms.renamevalue.blacklist=ID
transforms.convert.type=org.apache.kafka.connect.transforms.Tim
estampConverter$Value (8)
transforms.convert.field=EXECUTION_TIME
transforms.convert.target.type=Timestamp
transforms.convert.format=yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSSSS
```

The `topics` parameter **(1)** specifies the name of the topic that the sink connector should read, while the `database` **(2)**, `collection` **(3)** and `connection.url` **(4)** specify the target. The `delete.on.null.values` **(5)** and `document.id.strategy` **(6)** combination shown above enables the sink connector to correctly handle insert, update and delete operations. The transformations specified in **(7)** make sure that the ID column values are correctly inserted as `_id` fields in MongoDB and that the ID field is not inserted separately. By default the CDC converts the Db2 timestamps to strings and the transformation shown in **(8)** is required to convert the timestamp values to BSON datetime type upon insert.

Test Results

The architecture described in the previous section has been implemented and tested on IBM zSystems. A special workload generator based on locust.io has been developed for Db2 on z/OS and a workload of 1000 inserts per second was generated into a table with the DDL below. The padding columns were filled with random data.

```
CREATE TABLE TRANSACTIONS
(
   ID                  INTEGER NOT NULL,
   ACCOUNT_ID          INTEGER NOT NULL,
   OP_TYPE             VARCHAR (100),
```

```
    AMOUNT              DECIMAL,
    EXECUTION_TIME      TIMESTAMP,
    PAD_INT_1           INTEGER,
    ...
    PAD_INT_15          INTEGER,
    PAD_VARCHAR_1       VARCHAR (100),
    ...
    PAD_VARCHAR_10      VARCHAR (100),
    PAD_TIMESTAMP_1     TIMESTAMP,
    ...
    PAD_TIMESTAMP_5     TIMESTAMP
)
IN DATABASE MONGOSRC;
```

All components of this architecture (with the exception of Confluent Schema Registry) have been implemented on IBM zSystems. IBM Event Streams Schema can run on IBM zSystems and could be used as an alternative to Confluent Schema Registry, which would make this architecture run exclusively on IBM zSystems.

The tests have shown that even with the limited shared resources that were available during the testing, we were able to easily push through 1000 inserts per second with sub-second delays using the configuration defaults. This is by no means any indication of the maximal performance that can be achieved with this architecture. IBM has published a report showing as many as 125K requests per second using multiple subscriptions. IBM CDC replication also offers ways to reduce latency and CPU overhead by providing numerous configuration options. The official documentation and the IBM publication here are recommended for more details.

Conclusion & Outlook

MongoDB Operational Data Layer is a commonly used pattern to enable modern applications to use mainframe data while keeping the associated z/OS software costs in check. In the first step, the read queries can be replicated to MongoDB, significantly reducing infrastructure costs. In a second step, the MongoDB Operational Data Layer can be used as the main database for modern applications. The data from MongoDB can be replicated back to Db2 to enable other use cases that depend on Db2 data. Both steps require an efficient near real-time replication that can feed data from Db2

to MongoDB and back. Utilizing IBM CDC Replication and Apache Kafka is a universal approach to organize bidirectional replications between different Db2 for z/OS and MongoDB. Sub-second latencies can be achieved while minimizing z/OS software costs by using the Remote Source capability of IBM CDC Replication. Overall, modernizing mainframe applications and data doesn't need to be complex. MongoDB and IBM are collaborating to make this path simpler, more cost-effective and efficient. To learn more about MongoDB's initiatives, connect with the MongoDB team here.

About the authors



Adilet Sabyrbaev is a solution architect working for MongoDB in Germany. In his current role, he is helping the largest German customers realize the full potential of MongoDB. Prior to joining MongoDB, he has spent 10 years with IBM in different roles spanning solution architecture, performance benchmarking and optimization, and development.



Hauke Melius is a part of the solution architects team in Germany and supports customers from different industries on their NoSQL database journey. Before he joined MongoDB in early 2020, he was working as a cloud presales consultant for Hewlett Packard Enterprise and ownCloud.



Gabriela Preiss is a manager for the Industry Solutions team at MongoDB. Her expertise is smart banking, personalization and data enrichment. Gabriela joined MongoDB in 2020 after holding teaching positions in the USA, as well as programming and QA roles in Europe.



Resources

For more information, please visit mongodb.com or contact us at sales@mongodb.com.
Case Studies (mongodb.com/customers)
Presentations (mongodb.com/presentations)
Free Online Training (university.mongodb.com)
Webinars and Events (mongodb.com/events)
Documentation (docs.mongodb.com)
MongoDB Atlas database as a service for MongoDB (mongodb.com/cloud)
MongoDB Enterprise Download (mongodb.com/download)
MongoDB Realm (mongodb.com/realm)

Legal Notice

This document includes certain "forward-looking statements" within the meaning of Section 27A of the Securities Act of 1933, as amended, or the Securities Act, and Section 21E of the Securities Exchange Act of 1934, as amended, including statements concerning our financial guidance for the first fiscal quarter and full year fiscal 2021; the anticipated impact of the coronavirus disease (COVID-19) outbreak on our future results of operations, our future growth and the potential of MongoDB Atlas; and our ability to transform the global database industry and to capitalize on our market opportunity. These forward-looking statements include, but are not limited to, plans, objectives, expectations and intentions and other statements contained in this press release that are not historical facts and statements identified by words such as "anticipate," "believe," "continue," "could," "estimate," "expect," "intend," "may," "plan," "project," "will," "would" or the negative or plural of these words or similar expressions or variations. These forward-looking statements reflect our current views about our plans, intentions, expectations, strategies and prospects, which are based on the information currently available to us and on assumptions we have made. Although we believe that our plans, intentions, expectations, strategies and prospects as reflected in or suggested by those forward-looking statements are reasonable, we can give no assurance that the plans, intentions, expectations or strategies will be attained or achieved. Furthermore, actual results may differ materially from those described in the forward-looking statements and are subject to a variety of assumptions, uncertainties, risks and factors that are beyond our control including, without limitation: our limited operating history; our history of losses; failure of our database platform to satisfy customer demands; the effects of increased competition; our investments in new products and our ability to introduce new features, services or enhancements; our ability to effectively expand our sales and marketing organization; our ability to continue to build and maintain credibility with the developer community; our ability to add new customers or increase sales to our existing customers; our ability to maintain, protect, enforce and enhance our intellectual property; the growth and expansion of the market for database products and our ability to penetrate that market; our ability to integrate acquired businesses and technologies successfully or achieve the expected benefits of such acquisitions; our ability to maintain the security of our software and adequately address privacy concerns; our ability to manage our growth effectively and successfully recruit and retain additional highly-qualified personnel; the price volatility of our common stock; the financial impacts of the coronavirus disease (COVID-19) outbreak on our customers, our potential customers, the global financial markets and our business and future results of operations; the impact that the precautions we have taken in our business relative to the coronavirus disease (COVID-19) outbreak may have on our business and those risks detailed from time-to-time under the caption "Risk Factors" and elsewhere in our Securities and Exchange Commission ("SEC") filings and reports, including our Quarterly Report on Form 10-Q filed on December 10, 2019, as well as future filings and reports by us. Except as required by law, we undertake no duty or obligation to update any forward-looking statements contained in this release as a result of new information, future events, changes in expectations or otherwise.