### Pilot Guide

### MQ Source Connector

### IBM Mainframe on zIIP 5/4/2022

Prepared by

## Damon Wild

Account Director

[damon@confluent.io](mailto:damon@confluent.io)

## Sam Morana

Senior Systems Engineer

[smorana@confluent.io](mailto:smorana@confluent.io)

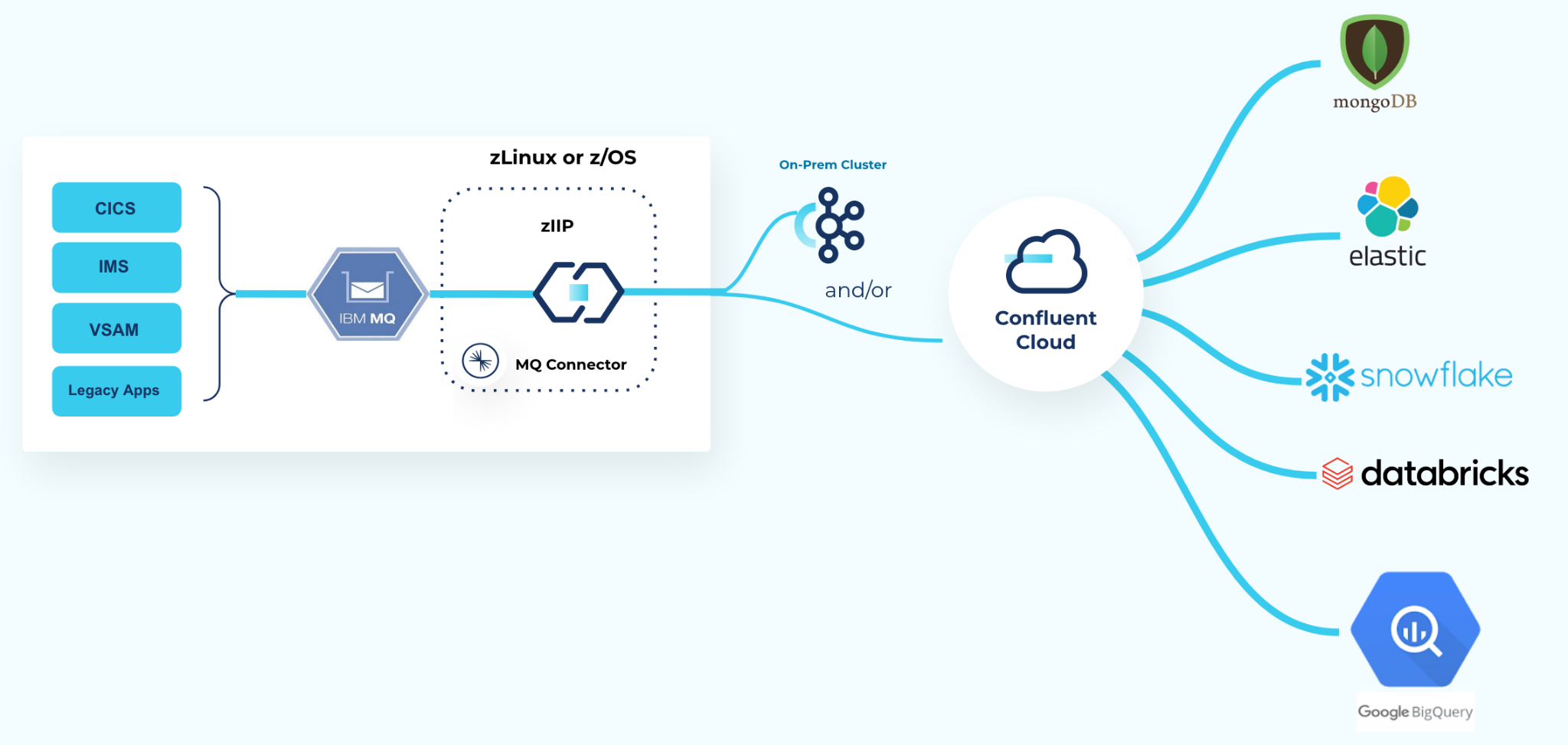
## Jeremy Hogan

IBM Partner Solutions Architect

[jhogan@confluent.io](mailto:jhogan@confluent.io)

**Abstract:**

Enterprises need data streaming on the Mainframe to unlock access to siloed data, improve operational efficiencies, and drive cloud adoption for mission critical business applications. Currently, there is no easy path-to-cloud.



**Solution:**

Confluent addresses this customer need using the connect framework on Z, known as ‘Connect on Z’.

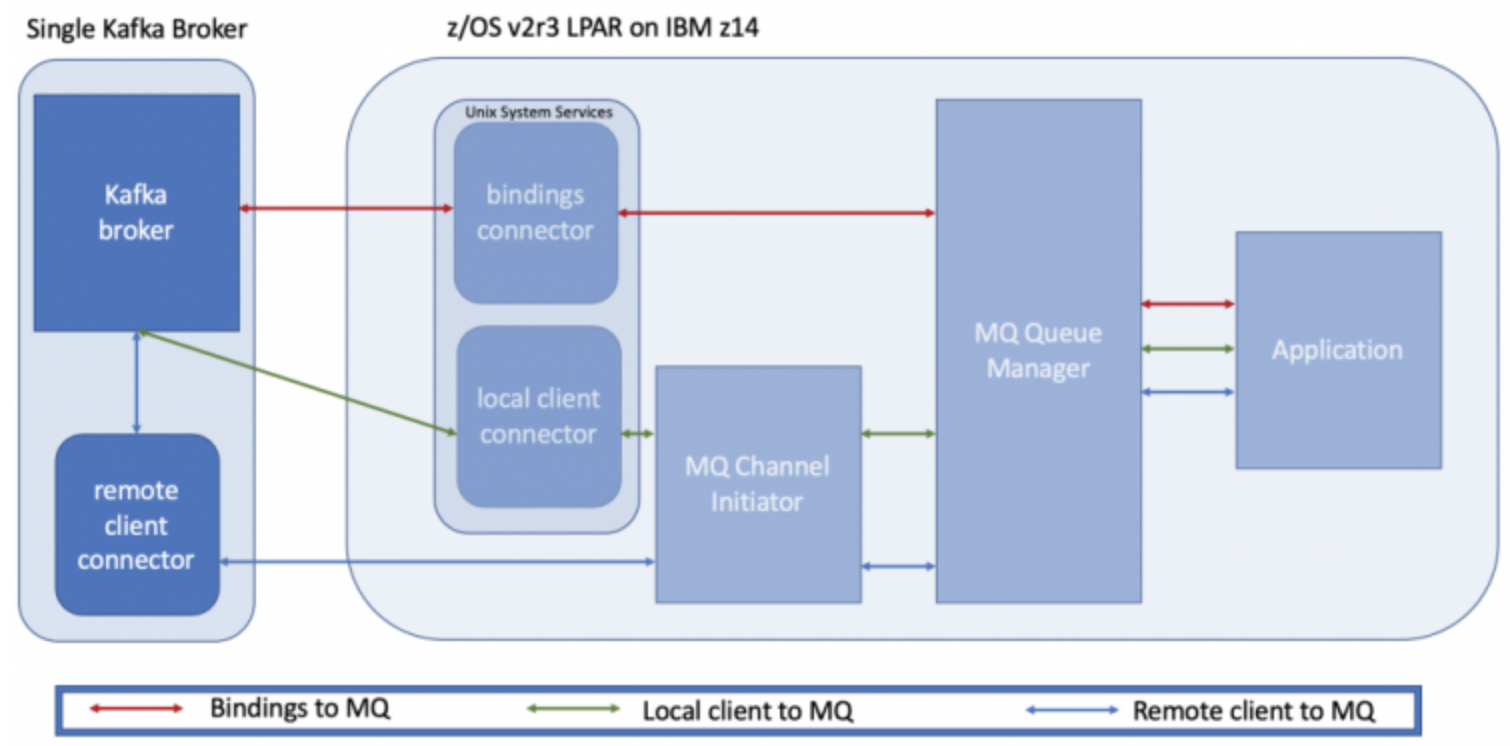
1. Unlock access to siloed critical data using connectors on Z, including MQ on zIIP and many other popular connectors
2. Create a ‘bridge to cloud’ option for mainframe, which otherwise does not exist today
3. Modernize and optimize the mainframe application infrastructure to drive innovation

**Benefits:**

* Sets mainframe data in motion
* Unlocks highest value use cases
* 90% reduction in GP-MIPS consumption per use case
* Can be configured to emit mainframe data (vua ClusterLinked) direct to Cloud

# Deployment Scenario:

An existing mainframe MQ queue contains messages (ex published from a CICS app). The MQ Connector in bindings mode (shared memory) eliminates MQ channel initiator costs (CHINIT) that drives GP MIPS expense. This is the easiest approach to get going and demonstrate value and savings. The following the red line path depicted in the diagram below:



*Note: Connecting to MQ using the blue or green line path, it is not possible to establish a bindings connection to MQ in this manner without going through the CHINIT. In terms of i/o consumption costs, these are significantly higher. This is the target savings area.*

# Prerequisites before installation:

* IBM MQ Running on Mainframe, know how MQ Manager is started:
  + MQ Connector needs to run in the same LPAR as MQ, set local, and in bindings mode
  + MQ Connector user model need to be setup equal to the MQ Manager
  + MQ Connector needs to be set to “system user”, not “end user”
  + Know which userid MQ Manager operates under
  + Know the version of MQ Manager on the mainframe (v9 is recommended)
  + If same MQ System is not used, then MQ Permissions are within scope (reading queues, committing messages from queue)
  + Run the MQ Manager as a stand alone gateway to isolate it’s resources to the MQ Connector only
* SSH (scp and sftp)
  + access to a Kafka SME to work in partnership w/Mainframe resource
  + Permissions read/write required for files and directories
  + USS access
  + Bash Shell (Anaconda recommended for Dev for simplicity)

# Java (know the version number)

* Run as a “Started Task”, not a “User Task”

**Testing workloads after installation:**

| **PRO TIP:** Use a real workload (data) based on your initial use case for testing and don’t set an expiry time on the messages.. When executing your JCL first use a standard sample program like **CSQUTIL** or **CSQ4BVK1**, not OEMPUT, or MQPUTVAR, and avoid using custom programs for the initial tests.. |
| --- |

When planning to start, set a message length to 2k or 3k. As long as Kafka consumes the job, you can put as many messages as possible. Run the job for 1 min to ensure 0 or single digit Qdepths and if good, run the job for 15 mins. Later, run longer durations & message lengths.

// SET M=MD9T QManager Name

// SET Q=MQM.QALIAS.KAFKA.POC QName

// SET N=1 Put number of messages at a time

// SET P=3K Put message length

// SET G=3K Get message length

// SET T=1 Time to run in minutes (1 minute)

// SET W=10 Wait in hundreths of a second 10=0.1

Use the mainframe SDSF console where this is reported in real-time. Run two repeatable workloads in separate LPAR’s. One with traffic running in **bindings mode** and another in **client mode**. Test after start up to get it to a steady state run back-to-back or 30 mins apart.. Run the workloads for 30 mins each.. MQ and Java (your MQ worker) will cut SMF records periodically as the work runs (hourly to a rolling 4 hour average). From these SMF records it is possible to report out by workload accurately how much time was spent and where for each type of workload.

# Installation Process

# Running connectors on IBM z/OS

You can use the IBM MQ connectors to connect into IBM MQ for z/OS, and you can run the connectors on z/OS as well, connecting into the queue manager using bindings mode. These instructions explain how to run Kafka Connect in standalone mode.

Before you can run IBM MQ connectors on IBM z/OS, you must prepare your Confluent files and your system as follows.

Setting up Kafka to run on IBM z/OS

You can run Confluent Platform Connect workers on IBM z/OS Unix System Services. To do so, you must ensure that the Confluent Platform Connect shell scripts and the Confluent Platform Connect configuration files are converted to EBCDIC encoding.

| **PRO TIP:** If you use scp (ascii mode) and sftp (binary) all the conversions happen automatically, and all the scripts and jars are ready to configure and test:  scp user@host:/path/to/cp/ sftp user@host:/path/to/cp/jars (let this overwrite the prior versions)  Or transfer the tar over and use dd like: dd conv=ebcdic if=ASCII\_file.txt of=EBCDIC\_file.txt  For example:  dd conv=ebcdic if=connect-standalone.properties of=connect-standalone2.properties |
| --- |

Downloads

Confluent Platform

Download Confluent Platform to a non-z/OS system to retrieve the .tar file that includes the Connect shell scripts and JAR files.

To download Confluent Platform and make it available to your z/OS system:

1. Log in to a system that is not running IBM z/OS, for example, a Linux system.
2. [Download](https://www.confluent.io/get-started/?product=software) latest stable release of Confluent Platform
3. Extract the downloaded .gz file, for example:  
   gunzip -k confluent-<ver>.tar.gz

**Confluent MQ Connector(s)**

From the Confluent site download the latest Connector version.

1. Download the [source connector JAR](https://www.confluent.io/hub/confluentinc/kafka-connect-ibmmq)  ([IBM MQ Source Connector Configuration Properties | Confluent Documentation](https://docs.confluent.io/kafka-connect-ibmmq-source/current/source_connector_config.html#ibmmq-source-connector-config))
2. (optional) Download the [sink connector JAR](https://www.confluent.io/hub/confluentinc/kafka-connect-ibmmq-sink) and [sink configuration file](https://docs.confluent.io/kafka-connect-ibmmq-sink/current/overview.html)
3. Extract Connector zip file

Copy to IBM z/OS

1. FTP the Confluent Platform .tar file to a directory on the z/OS Unix System Services
2. FTP the extracted Confluent IBM Source Connector root-folder (confluentinc-kafka-connect-ibmmq-<ver>) to another directory on the mainframe.

Create a folder on mainframe zFS file system that will contain both the Confluent Platform tar files and the MQ Source Connector files  
  
Example: /slocal/zwas/confluent/confluent-7.0.1 &   
 /slocal/zwas/confluent/ibmmq-11.0.11

Extract Confluent Platform:

1. Log in to the IBM z/OS system and access the Unix System Services.
2. Change to an empty directory that you want to use for Confluent Platform, and copy the .tar file to the new directory.
3. Extract the .tar file, for example:  
   tar -xvf confluent-<ver>.tar
4. Change to the resulting confluent-<ver> directory.

Convert the shell scripts:

Convert the following shell scripts and configuration files from ISO8859-1 to EBCDIC encoding   
  
In the Confluent Platform Directory:

* bin/kafka-run-class
* bin/connect-standalone
* etc/kafka/connect-standalone.properties

In the Connector Directory:

* etc/IbmMQSourceConnector.properties

bin:

connect-standalone  
kafka-run-class

1. Copy the connect-standalone shell script (or connect-distributed for a distributed setup) into the current directory (CP root directory), for example:  
   cp bin/connect-standalone ./connect-standalone.orig
2. Determine the codeset on the IBM z/OS system by running:  
   locale -k codeset
3. Convert the script to EBCDIC encoding and replace the original, for example for codeset IBM-1047:  
   iconv -f ISO8859-1 -t IBM-1047 ./connect-standalone.orig > bin/connect-standalone
4. Ensure the file permissions are set so that the script is executable, for example:  
   chmod +x bin/connect-standalone
5. Copy the kafka-run-class shell script into the current directory, for example:  
   cp bin/kafka-run-class ./kafka-run-class.orig
6. Convert the script to EBCDIC encoding and replace the original, for example for codeset IBM-1047:  
   iconv -f ISO8859-1 -t IBM-1047 ./kafka-run-class.orig > bin/kafka-run-class
7. Ensure the file permissions are set so that the script is executable, for example:  
   chmod +x bin/kafka-run-class

Convert the following files from the Confluent folders:

Convert the properties files:

etc/kafka:

connect-standalone.properties

Convert the configuration files:

1. Copy the connect-standalone.properties file into the current directory, for example:  
   cp config/connect-standalone.properties ./connect-standalone.properties.orig
2. Determine the codeset on the IBM z/OS system by running:  
   locale -k codeset
3. Convert the script to EBCDIC encoding and replace the original, for example for codeset IBM-1047:  
   iconv -f ISO8859-1 -t IBM-1047 ./connect-standalone.properties.orig > config/connect-standalone.properties

In standalone mode:

1. Copy etc/IbmMQSourceConnector.properties file into the current directory, for example:  
   cp etc/IbmMQSourceConnector.properties ./IbmMQSourceConnector.properties.orig
2. Determine the codeset on the IBM z/OS system by running:  
   locale -k codeset
3. Convert the script to EBCDIC encoding and replace the original, for example for codeset IBM-1047:  
   iconv -f ISO8859-1 -t IBM-1047 ./IbmMQSourceConnector.properties.orig > etc/IbmMQSourceConnector.properties

Connect workers can be run in two different modes.. For development or environments that lend themselves to single agents (e.g. sending logs from webservers to Kafka), **standalone** **mode** is a good start..   
  
When ready to test heavier data volumes (e.g. sending data from Kafka to cCloud), **distributed mode** is highly scalable and offers the added advantage of a high availability service to minimize downtime. We recommend distributed mode for production deployments for ease of management and scalability.  
  
Assuming a start in Dev, running a standalone Kafka Connect worker and as a result will need a connect-standalone.properties file.

Update the Connect configuration

The connect-standalone.properties file must include the correct bootstrap.servers and SASL/SSL configuration for your Confluent Platform install.

Generate credentials that can produce, consume and create topics and update the connect-standalone.properties file.

—------------------------- Sample connect-standalone.properties file —--------------------------------  
  
# Licensed to the Apache Software Foundation (ASF) under one or more  
# contributor license agreements. See the NOTICE file distributed with  
# this work for additional information regarding copyright ownership.  
# The ASF licenses this file to You under the Apache License, Version 2.0  
# (the "License"); you may not use this file except in compliance with  
# the License. You may obtain a copy of the License at  
#  
# <http://www.apache.org/licenses/LICENSE-2.0>  
#  
# Unless required by applicable law or agreed to in writing, software  
# distributed under the License is distributed on an "AS IS" BASIS,  
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.  
  
# These are defaults. This file just demonstrates how to override some settings.  
# A list of host/port pairs to use for establishing the initial connection to the Kafka cluster.  
bootstrap.servers=<remote-host1-url>:9092,<remote-host2-url>:9092,<remote-host3-url>:9092

# The converters specify the format of data in Kafka and how to translate it into Connect data. Every Connect user will  
# need to configure these based on the format they want their data in when loaded from or stored into Kafka  
key.converter=org.apache.kafka.connect.json.JsonConverter  
value.converter=org.apache.kafka.connect.json.JsonConverter

# Converter-specific settings can be passed in by prefixing the Converter's setting with the converter we want to apply  
# it to

key.converter.schemas.enable=true  
value.converter.schemas.enable=true

# Sample String Converter  
# key.converter=org.apache.kafka.connect.storage.StringConverter  
# value.converter=org.apache.kafka.connect.storage.StringConverter  
  
#  
#  
# SSL Configs  
#  
# Protocol to use to communicate with brokers. Valid values: PLAINTEXT, SSL, SASL\_PLAINTEXT, SASL\_SSL

# Workder Settings  
security.protocol=SASL\_SSL  
ssl.protocol=TLSv1.2  
# file format of the trust store file  
ssl.truststore.type=JKS  
ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks  
ssl.truststore.password=test1234  
# Worker authentication  
ssl.keystore.location=/var/ssl/private/kafka.worker.keystore.jks  
ssl.keystore.password=worker1234  
ssl.key.password=workder1234  
sasl.mechanism=PLAIN  
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required   
 username="client" password="client-secret";

# Producer (Source Connector) Settings  
producer.security.protocol=SASL\_SSL  
producer.ssl.protocol=TLSv1.2  
producer.ssl.truststore.type=JKS  
producer.ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks  
producer.ssl.truststore.password=test1234  
producer.ssl.keystore.location=/var/ssl/private/kafka.connector.keystore.jks  
producer.ssl.keystore.password=connector1234  
producer.ssl.key.password=connector1234  
producer.sasl.mechanism=PLAIN  
producer.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required  
username="client" password="client-secret";

# Consumer (Sink Connector) Settings  
consumer.security.protocol=SASL\_SSL  
consumer.ssl.truststore.type=JKS  
consumer.ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks  
consumer.ssl.truststore.password=test1234  
consumer.ssl.keystore.location=/var/ssl/private/kafka.connector.keystore.jks  
consumer.ssl.keystore.password=connector1234  
consumer.ssl.key.password=connector1234  
consumer.sasl.mechanism=PLAIN  
consumer.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="client" password="client-secret";

# This property only applies to workers of standalone source connectors  
# The file to store connector offsets in. By storing offsets on disk, a standalone process can be stopped and started on a single node and resume where it previously left off.  
offset.storage.file.filename=/tmp/connect.offsets

# Flush much faster than normal, which is useful for testing/debugging  
offset.flush.interval.ms=10000

# Set to a list of filesystem paths separated by commas (,) to enable class loading isolation for plugins  
# (connectors, converters, transformations). The list should consist of top level directories that include

# any combination of:   
# a) directories immediately containing jars with plugins and their dependencies  
# b) uber-jars with plugins and their dependencies  
# c) directories immediately containing the package directory structure of classes of plugins  
and their dependencies  
# Note: symlinks will be followed to discover dependencies or plugins.  
# Examples:   
plugin.path=/usr/local/share/java,/usr/local/share/kafka/plugins,/opt/connectors

—------------------------------------- End of Sample —-----------------------------------------------------

—------------------------- Sample IbmMQSourceConnector.properties file —--------------------------

name=connector1

tasks.max=1

connector.class=io.confluent.connect.ibm.mq.IbmMQSourceConnector

*# The following values must be configured and should match your environment.*

mq.transport.type=bindings

mq.queue.manager=QM1  
  
*#Connector should run on the same LPAR as the MQ Manager so mq.hostname not needed*

mq.hostname=” “  
  
*#no need for mq.channel since we’re running in bindings mode*

mq.channel=” “

*# MQ Session params*

jms.destination.name=SOURCE.MQ.QUEUE.NAME

*# valid values [queue,topic]*

jms.destination.type=queue

*# maximum number of records that a connector task may read from MQ before writing to Kafka, default is 1024*

batch.size=1024

*# maximum amount of time in milliseconds (default is 5) that a JMS receive call will be blocked. Helps to reduce the number of empty receive calls*

jms.receive.block.duration=5

*# The acknowledge mode for the JMS Session. valid values [client, auto, dups\_ok] default is client*

jms.session.acknowledge.mode=client

*# Kafka*

kafka.topic=MyKafkaTopicName

*# The following define the information used to validate the license stored in Kafka*

*# or leave blank for a 30 day trial*

confluent.license=” “

confluent.topic.bootstrap.servers=<remote-host1-url>:9092,<remote-host2-url>:9092,<remote-host3-url>:9092

[MQ Source Connector Properties Documentation](https://docs.confluent.io/kafka-connect-ibmmq-source/current/source_connector_config.html)

bootstrap.servers=<bootstrapServers>

security.protocol=SASL\_SSL

ssl.protocol=TLSv1.2

ssl.truststore.location=/opt/kafka/es-cert.p12

ssl.truststore.password=<truststorePassword>

ssl.truststore.type=PKCS12

sasl.mechanism=SCRAM-SHA-512

sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required username="<userName>" password="<password>";

producer.security.protocol=SASL\_SSL

producer.ssl.protocol=TLSv1.2

producer.ssl.truststore.location=/opt/kafka/es-cert.p12

producer.ssl.truststore.password=<truststorePassword>

producer.ssl.truststore.type=PKCS12

producer.sasl.mechanism=SCRAM-SHA-512

producer.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required username="userName" password="<password>";

consumer.security.protocol=SASL\_SSL

consumer.ssl.protocol=TLSv1.2

consumer.ssl.truststore.location=/opt/kafka/es-cert.p12

consumer.ssl.truststore.password=<truststorePassword>

consumer.ssl.truststore.type=PKCS12

consumer.sasl.mechanism=SCRAM-SHA-512

consumer.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required username="userName" password="<password>";

plugin.path=/opt/connectors

—------------------------------------- End of Sample —-----------------------------------------------------

# 

# Starting Kafka Connect on z/OS

Kafka Connect is started using a bash script. If you do not already have bash installed on your z/OS system install it now.

To install bash version 4.2.53 or later:

1. Bash binaries can be downloaded from Anaconda @ <https://anaconda.org/zoss-appdev/bash/files>
2. Extract the archive file to get the .tar file:  
   gzip -d bash.tar.gz
3. FTP the .tar file to your z/OS USS directory such as /bin
4. Extract the .tar file to install bash:  
   tar -xf bash.tar

If bash on your z/OS system is not in /bin, you need to update the connect-standalone & kafka-run-class file. For example, if bash is located in /usr/local/bin update the first line of the connect-standalone & kafka-run-class files to have #!/usr/local/bin/bash

Starting Kafka Connect in standalone mode

Navigate to your Confluent Platform bin directory and run the connect-standalone script, passing in your connect-standalone.properties and IbmMQSourceConnector.properties For example:

cd confluent-<ver>/bin

./connect-standalone ../etc/kafka/connect-standalone.properties <connector-dir>/etc/IbmMQSourceConnector.properties

**Running the Connector Within a PROC**

To run the Connect Workers as a Started Task running locally to the MQ qmgr in bindings mode.

Add the following to the mainframe PROCLIB. Use BPXBATCH to drive a script that exports all the runtime properties prior to calling Confluent’s Standalone Connector

Example PROC to put in SYSx.PROCLIB

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Top of Data \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//NameOfSTC PROC

//\*

// SET INSTDIR='**path to confluent bin dir example /slocal/zwas/confluent/bin**'

// SET QMGR='**your-qmgr SSID**'

//\*

//CONNECT EXEC PGM=BPXBATCH,REGION=0M,TIME=NOLIMIT,MEMLIMIT=12G,

// PARM='**SH &INSTDIR./&QMGR.-mq-connect-standalone**'

//STEPLIB DD DISP=SHR,DSN=HLQ.&QMGR..SCSQANLE

// DD DISP=SHR,DSN=HLQ.&QMGR..SCSQAUTH

//STDOUT DD SYSOUT=\*

//STDERR DD SYSOUT=\*

//STDIN DD DUMMY

//STDENV DD \*

TMPDIR=</path/to/tmp/dir>

JAVA\_HOME=/usr/lpp/java/J8.0\_64

You can write an intermediary z/OS shell script to your own in-house defined directory to provide a level of indirection and to set the environment as desired.

Here’s an example of such a file:  
  
**qmgr-mq-connect-standalone.sh** file referenced on the PARM line in the PROC above::

#!/bin/sh  
qmgr='qmgr'  
jmxport='1234'  
mq\_maint='v910/cycle6'  
srvAcct='user'  
  
export qmgrlog="$qmgr-Dist"

bin\_dir='/slocal/zwas/confluexec/bin'  
cfg\_dir='/slocal/zwas/confluexec/etc/kafka'  
  
CLASSPATH0='/slocal/zwas/conflucoex/lib'  
  
# S.M. - Don't think we need this since the Confluent MQ Source Connector has a kafka-connect-jms-source-base-ver.jar"  
  
#CLASSPATH1='/slocal/zwas/confluexec/$mq\_maint/java/lib/com.ibm.mqjms.jar'

export CLASSPATH="$CLASSPATH0"

# MQ Environment - Library Path to MQ Libs and STEPLIB pathexport LIBPATH=/path/to/mq/$mq\_maint/java/lib  
export STEPLIB=HLQ.$qmgr.SCSQAUTH:HLQ.$qmgr.SCSQANLE:$STEPLIB

#Don’t default this log writing out to Starter Task SYSOUT because it’ll consume GP time  
# To write out unde zIIP specify the parameters below

#Customize Garbage Collection (GC)  
#  
kafka\_logs\_dir='/apps/websphere/kafka/logs/'$qmgr' -Dist-GC'  
TIMESTAMP=`date+'%Y%m%d%H%M'`  
# GC log location/name prior to .n addition by log rotation  
GC\_LOG\_NAME="$kafka\_logs\_dir/gc.log-$TIMESTAMP"  
GC\_LOG\_ENABLE\_OPTS="-verbose:gc -Xloggc:$GC\_LOG\_NAME"  
GC\_LOG\_ROTATION\_OPTS="-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M"  
GC\_LOG\_FORMAT\_OPTS="-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps"

export KAFKA\_GC\_LOG\_OPTS="$GC\_LOG\_ENABLE\_OPTS $GC\_LOG\_ROTATION\_OPTS $GC\_LOG\_FORMAT\_OPTS"  
  
#  
# externalize override of default heap  
#  
export KAFKA\_HEAP\_OPTS="Xms256M -Xmx6G"  
#  
# DUMP Options  
#  
export IBM\_JAVA\_ZOS\_TDUMP=ALL  
export IBM\_JAVA\_ZOS\_TDUMP\_COUNT=2  
export IBM\_JAVA\_ZOS\_TDUMP\_PATTERN="HLQ.<MLQ>.&JOBNAME..T&HHMMSSS."  
export JAVA\_DUMP\_TDUMP\_PATTERN="HLQ.<MLQ>.&JOBNAME..T&HHMMSS."  
export \_CEE\_DMPTARG=/apps/websphere/kafka/dumps (location of MQ filesystem)  
#  
# add JMX Port  
#  
export JMX\_PORT=$jmxport  
#  
# Awareness for JAAS.CONF for KRB5 License check  
#  
export KAKFA\_OPTS="-Djava.security.auth.login.config=/path/to/keytab/$srvAcct-JAAS.conf"  
#  
# override LOG4J  
#  
export KAFKA\_LOG4J\_OPTS="-Dlog4j.configuration=file:/slocal/zwas/confluexec/etc/kafka/$qmgr-Connect-log4j.properties" – should be unique per worker config file.  
#  
#  
# java.security  
#  
export KAFKA\_OPTS="-Djava.security.properties=/path/to/java.security"  
#  
# Start the Connect Worker:  
  
exec $bin\_dir/connect-standalone $cfg\_dir/connect-standalone.properties $cfg\_dir/IbmMQSourceConnector.properties

**Appendix**

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

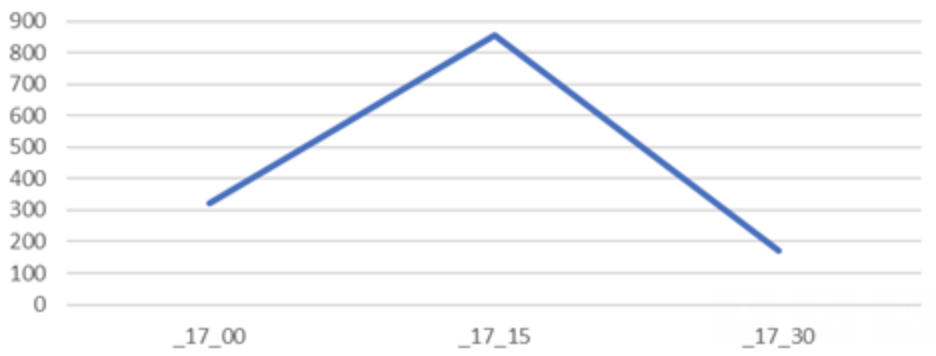
# 

# Typical TCO analysis from Testing

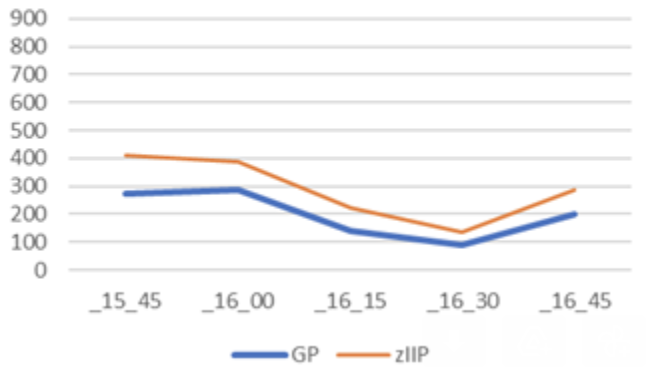
All Opens and GETs will traverse the CHINT when running in Client mode. This is the first chart below and all these are GP MIPs.

Results from testing will reveal lower MIPs cost. All Opens and GETs will traverse the CHINIT running in Client mode which consumes (CHINIT) that drives GP MIPS expense. Actual production workload will utilize more zIIP processing than test LPARs (see notes below on Kafka testing).

* SRVCONN testing consumed upwards of 850 MIPS (almost a single engine) in a single 15 minute interval:



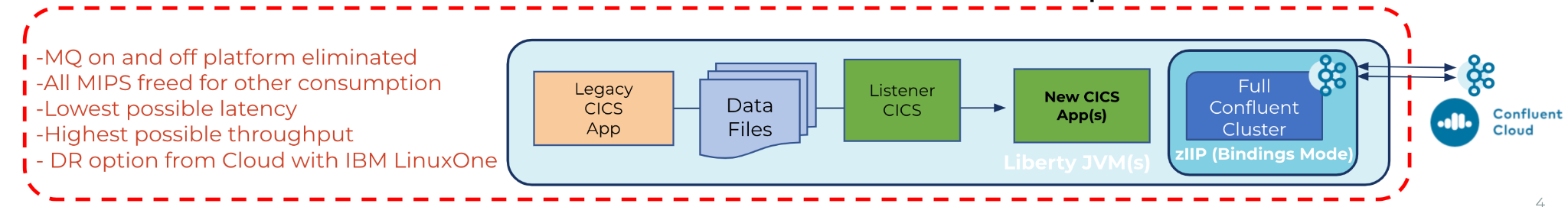
* At peak, combined zIIP and GP MIPs reached slightly over 400 MIPs, then tailed off afterwards. Important to note, there is only one zIIP engine available on this Test CPU, and while there was eligible time spent on GP, roughly ~40% offload was achieved in this test:



# 

# Experimental Deployment Scenario:

Write a CICS application in Java that uses Kafka Java APIs to emit/consume events to/from Kafka. This will require some Java development skills, and setting up a CICS Liberty JVM server to run the application. This is likely to be the most flexible and reflects the Target State. This will also prove that distributed MQ depths can be modernized to eliminate saving another layer of middleware expense.



For legacy CICS/IMS/WAS, rather than using connect, you can also produce/consume directly to/from Kafka brokers themselves. The easiest way to do this is to enable the Liberty JVM, have CICS applications (or IMS or WAS) emit to Liberty and our off-the-shelf [consumer](https://docs.confluent.io/platform/current/clients/consumer.html) or [producer](https://docs.confluent.io/platform/current/clients/producer.html) will drop in place to take care of the rest.   
  
CICS and Kafka integration:  
<https://community.ibm.com/community/user/ibmz-and-linuxone/blogs/mark-cocker1/2020/08/07/cics-and-kafka-integration>  
  
Configuring a JVM server for web applications:  
<https://www.ibm.com/docs/en/cics-ts/5.1?topic=server-configuring-jvm-web-applications>  
  
Deploying a web application as a WAR or EBA file to a Liberty JVM server:  
<https://www.ibm.com/docs/api/v1/content/SSGMCP_5.1.0/com.ibm.cics.ts.java.doc/JVMserver/create_liberty_bundle.html>  
  
  
Java info from our own test env:

java version “1.8.0\_331”

Java(TM) SE Runtime Environment (build 8.0.7.10 - pmz6480sr7fp10-20220505\_01(SR7 FP10))

IBM J9 VM (build 2.9, JRE 1.8.0 z/OS s390x-64-Bit Compressed References 20220427\_27745 (JIT enabled, AOT enabled)

OpenJ9 - b15041a

OMR - 3671a9f

IBM - 1b0232b)

JCL - 20220504\_01 based on Oracle jdk8u331-b09