**IBM’s Power Private Cloud Rack for Db2 Warehouse Solution – Chapter 6: High Availability**

**A computer screen with a logo on it

AI-generated content may be incorrect.**

**By Muhammed Hisham P, Aruna De Silva**

1. **Introduction**

As enterprises increasingly adopt containerized environments for mission-critical workloads, ensuring high availability (HA) of databases like IBM Db2 becomes essential. In this sixth instalment of the Db2 Warehouse on Power Cloud Rack blog series, we explore how the Db2 Warehouse on Power Cloud Rack architecture—deployed on Red Hat OpenShift—handles both soft and hard failover scenarios to maintain service continuity. From Pod-level disruptions to full node failures, this chapter details the mechanisms, tests, and outcomes that validate the robustness of the Db2 Warehouse solution in real-world conditions.

* [Introducing IBM’s Private Cloud Rack for Db2 Warehouse Solution: Architecture, Scalability, and Performance](https://community.ibm.com/community/user/blogs/jana-wong/2025/03/04/introducing-ibms-p10-pcr-for-db2wh)
* [Chapter 2: Db2 12.1 Performance Deep-Dive](https://community.ibm.com/community/user/blogs/jana-wong/2025/05/14/ibms-power10-private-cloud-rack-for-db2-warehouse)
* [Chapter 3: Step-By-Step Guide to Engage IBM Support](https://community.ibm.com/community/user/blogs/jana-wong/2025/05/22/ibms-power-10-private-cloud-rack-for-db2-warehouse)
* [Chapter 4: Enabling Q Replication](https://community.ibm.com/community/user/blogs/jana-wong/2025/05/28/ibms-power10-private-cloud-rack-for-db2-warehouse)
* [Chapter 5: Backup & Restore](https://community.ibm.com/community/user/blogs/jana-wong/2025/08/26/ibms-power-private-cloud-rack-for-db2-warehouse-so)

1. **Understanding HA Failure Scenarios on IBM’s Db2 Warehouse on Power Cloud Rack**

In a production-grade Db2 Warehouse on Power Cloud Rack deployment on OpenShift, ensuring high availability (HA) involves preparing for both soft and hard failover scenarios. These scenarios simulate real-world disruptions and help validate the resilience of the system.

**2.1 Soft Failover Scenarios**

Soft failovers are typically triggered by transient or recoverable issues within the OpenShift environment. These do not involve infrastructure-level failures but still test the robustness of the Db2 HA setup.

**Common Soft Failover Triggers:**

* **Pod deletion** (intentional or accidental)
* **Container crash** due to application-level issues
* OpenShift rolling updates or configuration changes (e.g. Kubelet or Machine Configuration changes)
* **Resource pressure** (e.g., CPU/memory eviction)

**What Happens During a Soft Failover:**

* Kubernetes automatically reschedules the Db2 Pod on the same or a different node.
* Persistent storage (via PVCs) ensures data continuity.
* Liveness probes help detect unhealthy containers and restarts them.
* Db2’s internal recovery mechanisms (e.g., crash recovery) kick in to restore service.

**2.2 Hard Failover Scenarios**

Hard failovers involve infrastructure-level failures that require more complex recovery mechanisms. These are critical to test for true HA readiness.

**Common Hard Failover Triggers:**

* **Worker node shutdown** or crash (e.g kubelet crash or kernel panic)
* **Node reboot** due to OS patching or hardware issues

**What Happens During a Hard Failover:**

* Is determined by the characteristics of the workload:
  + *Stateless workloads*: Kubernetes (OpenShift) scheduler detects node unavailability and reschedules the workload on a healthy node(s).
  + *Stateful workloads*: Stateful application cannot function properly if the Pods are stuck on the shutdown node and not getting rescheduled onto a running node. Starting Kubernetes v1.28 (OpenShift 4.14), an [out-of-service taint](https://kubernetes.io/docs/concepts/cluster-administration/node-shutdown/#non-graceful-node-shutdown) can be applied on the failed node to trigger a force-deletion of Pods on that node and persistent volumes to be detached. This allows new Pods to be created successfully on a different running node. Db2 Pods require a unique/stable network identify and hence fall into the Stateful workload category.
* Persistent storage must be accessible from multiple nodes (e.g., via CSI drivers).

1. **Validating Soft Failover Resilience on IBM’s Db2 Warehouse on Power Cloud Rack**

To ensure the robustness of our Db2 Warehouse on Power Cloud Rack deployment on OpenShift, we conducted a series of soft failover tests. These tests simulate common, non-infrastructure-related disruptions and help measure the system’s ability to recover gracefully. The test Cloud Rack system was a Base Large Rack (BRL) with 6 active Pods running on 6 worker nodes and a standby worker node for HA.

**3.1 Test Scenario 1: Single Pod Deletion**

In this test, we randomly deleted a single Db2 Pod to simulate a crash or accidental termination.

**Objective:**

To observe how quickly the system recovers and the Db2 instance becomes operational again.

**Result:**

* The Pod was automatically rescheduled by OpenShift.
* Db2 container reinitialized and passed readiness checks.
* **Average recovery time:** **3 minutes and 10 seconds**

The data below illustrates the outcome of a single-node soft failure test iteration

|  |  |  |
| --- | --- | --- |
| Command ran | Pod Status | Smoke Test Result |
| date; oc delete pod c-db2u-cr-db2u-3; date  Sun Jan 19 23:55:13 EST 2025  pod "c-db2u-cr-db2u-3" deleted  Sun Jan 19 23:55:44 EST 2025 | 23:55:13 - Running  23:55:14 - Terminating  23:55:45- 23:56:07 - Init  23:56:08 - Running Not Ready  23:58:02 - Running Ready  db2 took 3 minutes and 11 seconds to come back up | smoke test successfully completed. |

**3.2 Test Scenario 2: Simultaneous Deletion of Two Pods**

To further stress the system, we **simultaneously deleted two Db2 Pods**, simulating a more severe but still soft failure scenario.

**Objective:**

To evaluate the system’s behaviour under concurrent Pod failures.

**Result:**

* Both Pods were rescheduled and reinitialized in parallel.
* Db2 instances recovered and became ready without manual intervention.
* **Average recovery time:** **3 minutes and 10–11 seconds**

This confirms that the HA setup can handle multiple simultaneous disruptions effectively.

**4. Validating Hard Failure Resilience on IBM’s Db2 Warehouse on Power Cloud Rack**

To complement our soft failover validation, we also tested **hard failure scenarios**—which simulate infrastructure-level disruptions. These tests are critical for evaluating the resilience of the Db2 Warehouse on Power Cloud Rack deployment under more severe conditions.

**4.1 Known Issue: Node Shutdown Not Detected by Kubernetes kubelet’s Shutdown Manager**

Starting Kubernetes 1.21 (OpenShift 4.8), [Graceful Node Shutdown feature](https://kubernetes.io/docs/concepts/cluster-administration/node-shutdown/#graceful-node-shutdown) can be enabled to allow **kubelet’s Node Shutdown Manager** to detect-and-handle planned node shutdowns (e.g. via Linux shutdown or poweroff commands). However, in certain scenarios, a node shutdown may not be properly detected, typically due to one of the following reasons:

* The shutdown command does not trigger the systemd **inhibitor locks mechanism** used by kubelet.
* Misconfiguration of shutdown parameters such as ShutdownGracePeriod and ShutdownGracePeriodCriticalPods.

When this detection fails, Db2u Pods managed by **Db2uEngine custom resource** running on the affected node enter a **stuck terminating state**. Since the kubelet is unavailable to complete the Pod deletion, OpenShift cannot recreate the Pods on another node. If these Pods use persistent volumes, the associated [VolumeAttachment](https://kubernetes.io/docs/reference/kubernetes-api/config-and-storage-resources/volume-attachment-v1/) resources also remain bound to the original node, preventing reattachment elsewhere.

As a result, the application becomes non-functional until the original node is restored. If the node does not come back online, the Pods remain indefinitely stuck in a terminating state.

To address the limitations of Kubelet’s Node Shutdown Manager in detecting certain shutdown scenarios, we developed a **custom cronjob** that proactively handles node-level failures. This solution polls [Node Status Conditions](https://kubernetes.io/docs/reference/node/node-status/#condition) for types NotReady and Unknown to detect worker node failures and trigger recovery. That ensures Db2u Pods are rescheduled successfully and associated Persistent Volumes are re-attached, even when the default shutdown mechanisms fail.

**4.2 Custom Node-HA CronJob: Automated Recovery from Undetected Node Shutdowns**

In the following section, we outline the steps to set up and configure this cronjob, which plays a critical role in maintaining high availability in our Db2 Warehouse on Power Cloud Rack deployment.

To begin setting up the Node-HA CronJob, you must first create a dedicated ServiceAccount (SA), along with the necessary ClusterRole and ClusterRoleBinding to apply the minimal [RBAC](https://kubernetes.io/docs/reference/access-authn-authz/rbac/) required to execute the job. Use the node-ha-rbac.yaml file provided for this purpose, ensuring that you **update the** ServiceAccount **namespace** in the ClusterRoleBinding section to match the project where the Db2uInstance Custom Resource (CR) is deployed.

**4.2.1 Kubernetes RBAC Configuration for Node-HA CronJob:**

|  |
| --- |
| apiVersion: v1  # If required specify pull secret  #imagePullSecrets:  #- name: <pull secret>  kind: ServiceAccount  metadata:  name: db2u-node-admin  ---  apiVersion: rbac.authorization.k8s.io/v1  kind: ClusterRole  metadata:  name: db2u-node-admin  rules:  - apiGroups:  - ""  resources:  - nodes  verbs:  - get  - list  - patch  - update  - apiGroups:  - ""  resources:  - pods  - pods/finalizers  verbs:  - delete  - get  - list  - watch  ---  apiVersion: rbac.authorization.k8s.io/v1  kind: ClusterRoleBinding  metadata:  name: db2u-node-admin  roleRef:  apiGroup: rbac.authorization.k8s.io  kind: ClusterRole  name: db2u-node-admin  subjects:  - kind: ServiceAccount  name: db2u-node-admin  namespace: db2 |

**4.2.2 Node-HA CronJob YAML Configuration**

Create the CronJob resource along with an associated ConfigMap to encapsulate the Node HA script logic. By wrapping HA logic in a ConfigMap allows using an existing OpenShift CLI client image instead of building a custom image to package the script. To deploy this configuration, use the file provided below

|  |
| --- |
| apiVersion: batch/v1  kind: CronJob  metadata:  name: db2u-node-ha  spec:  schedule: "\*/5 \* \* \* \*"  # [Tech preview] Optional timezone - can help with correlating job logs  #timeZone: Etc/UTC  concurrencyPolicy: Forbid  #startingDeadlineSeconds: 200  successfulJobsHistoryLimit: 1  # Defaults to 3  failedJobsHistoryLimit: 3  jobTemplate:  spec:  template:  metadata:  labels:  parent: "cronjob-db2ha"  spec:  serviceAccountName: db2u-node-admin  volumes:  - name: db2u-node-ha-script-volume  configMap:  name: db2u-node-ha-script  # Read-Execute for all: mode-bits (0555)  defaultMode: 365  containers:  - name: db2u-ha  image: registry.redhat.io/openshift4/ose-cli:v4.12.0-202405222205.p0.gd691257.assembly.stream.el8  volumeMounts:  - mountPath: /db2u  name: db2u-node-ha-script-volume  command: ["/db2u/db2u-node-ha.sh"]  restartPolicy: OnFailure  apiVersion: v1  kind: ConfigMap  metadata:  name: db2u-node-ha-script  data:  db2u-node-ha.sh: |  #!/bin/bash  echo "Checking if any node is down ..."  nodls=""  nodlsNotReady=$(oc get nodes --selector '!node-role.kubernetes.io/master' --output jsonpath="{range .items[?(@.status.conditions[-1].type=='NotReady')]}{.metadata.name} {.status.conditions[-1].type}{'\n'}{end}" | cut -d" " -f1)  echo -e "NotReady nodes: \n${nodlsNotReady}"  nodlsUnknown=$(oc get nodes --selector '!node-role.kubernetes.io/master' --output jsonpath="{range .items[?(@.status.conditions[-1].status=='Unknown')]}{.metadata.name} {.status.conditions[-1].type}{'\n'}{end}" | cut -d" " -f1)  echo -e "Unknown nodes: \n${nodlsUnknown}"  nodlsReady=$(oc get nodes --selector '!node-role.kubernetes.io/master' --output jsonpath="{range .items[?(@.status.conditions[-1].type=='Ready')]}{.metadata.name} {.status.conditions[-1].type}{'\n'}{end}" | cut -d" " -f1)  echo -e "Ready nodes: \n${nodlsReady}"  nodls="${nodlsNotReady} ${nodlsUnknown}"  echo "${nodls}" | grep -E "^[ \t]+$"  if [[ $? -eq 0 ]] && nodls=""; then  echo -e "\nAll nodes are running fine."  else  for node in $(echo ${nodls}); do  echo "Node ${node} not ready, wait 10 min before taking action"  date  sleep 600  status\_condtype=$(oc get node ${node} -o jsonpath="{.status.conditions[-1].type}")  status\_condstatus=$(oc get node ${node} -o jsonpath="{.status.conditions[-1].status}")  if [[ "${status\_condtype}" == "NotReady" || "${status\_condstatus}" == "Unknown" ]]; then  echo "Node ${node} is still Not Ready after 10 min. Taking action."  echo "Tainting node ${node} as out-of-service"  oc adm taint nodes ${node} node.kubernetes.io/out-of-service=nodeshutdown:NoExecute --overwrite  else  echo "Node ${node} recovered after 10 minutes, no action taken."  fi  date  done  fi  nodlsReady=$(oc get nodes --selector '!node-role.kubernetes.io/master' --output jsonpath="{range .items[?(@.status.conditions[-1].type=='Ready')]}{.metadata.name} {.status.conditions[-1].type}{'\n'}{end}" | cut -d" " -f1)  echo -e "Ready nodes: \n${nodlsReady}"  for node in ${nodlsReady}; do  echo "Check Node ${node} is still ready, before removing taint"  status\_condtype=$(oc get node ${node} -o jsonpath="{.status.conditions[-1].type}")  status\_condstatus=$(oc get node ${node} -o jsonpath="{.status.conditions[-1].status}")  if [[ "${status\_condtype}" != "NotReady" && "${status\_condstatus}" != "Unknown" ]]; then  oc get node ${node} -o jsonpath='{range .items[\*]}{.spec.taints[\*].key}' | grep -qE "^node.\*out-of-service$"  if [[ $? -eq 0 ]] ; then  echo "Removing out-of-service Taint from node ${node}"  oc adm taint nodes ${node} node.kubernetes.io/out-of-service- --overwrite  else  echo "No out-of-service taint found on the node ${node}. Checking the next node"  continue  fi  fi  done  echo "Script execution completed" |

***Note****: Refer to OpenShift CLI client repository history for list of client images available.*

# **4.2.3 Verifying CronJob Execution and Reviewing Logs**

Once the Node-HA CronJob is configured and actively running, you can verify its execution status and inspect the logs using the following steps:

1. **Check CronJob Status:**

|  |
| --- |
| # oc get po --selector parent=cronjob-db2ha  NAME READY STATUS RESTARTS AGE  db2u-node-ha-28657725-bgx6d 0/1 Completed 0 71s    # oc logs db2u-node-ha-28657725-bgx6d  Checking if any node is down ...  NotReady nodes:    Unknown nodes:    Ready nodes:  worker0.adesilva.cp.fyre.ibm.com  worker1.adesilva.cp.fyre.ibm.com  worker2.adesilva.cp.fyre.ibm.com    All nodes are running fine. |

# **[Optional] Enabling Cascading Deletion for the CronJob ConfigMap**

By default, when a CronJob resource is deleted, its associated ConfigMap is **not automatically removed**. To streamline cleanup and ensure consistency, you can enable **cascading deletion** by configuring an ownerReference on the ConfigMap. This ensures that the ConfigMap is automatically deleted when the CronJob is removed.

**4.2.4 [Optional] Enabling Cascading Deletion for the CronJob ConfigMap**

1. **Retrieve the UID of the CronJob:**

|  |
| --- |
| oc get cronjob db2u-node-ha -o jsonpath='{.metadata.uid} {"\n"}' |

Example output:

|  |
| --- |
| c997615e-bec5-4943-9826-8fd40898a570 |

1. **Edit the ConfigMap to add ownerReferences:**

|  |
| --- |
| oc edit cm db2u-node-ha-script |

In the metadata section, add the following block (replace the uid with the one retrieved in step 1):

|  |
| --- |
| ownerReferences:  - apiVersion: batch/v1  blockOwnerDeletion: true  controller: true  kind: CronJob  name: db2u-node-ha  uid: c997615e-bec5-4943-9826-8fd40898a570 |

1. **Verify Cascading Deletion: After applying the changes, delete the CronJob:**

|  |
| --- |
| oc delete cronjob db2u-node-ha |

The associated db2u-node-ha-script ConfigMap should now be automatically deleted as part of the cascading cleanup process.

**4.3 Test Scenario 1: Shutting Down a Worker Node**

In this test, we **randomly shut down one of the worker nodes** in the OpenShift cluster. As a result, the Db2 Pod scheduled on that node became unavailable.

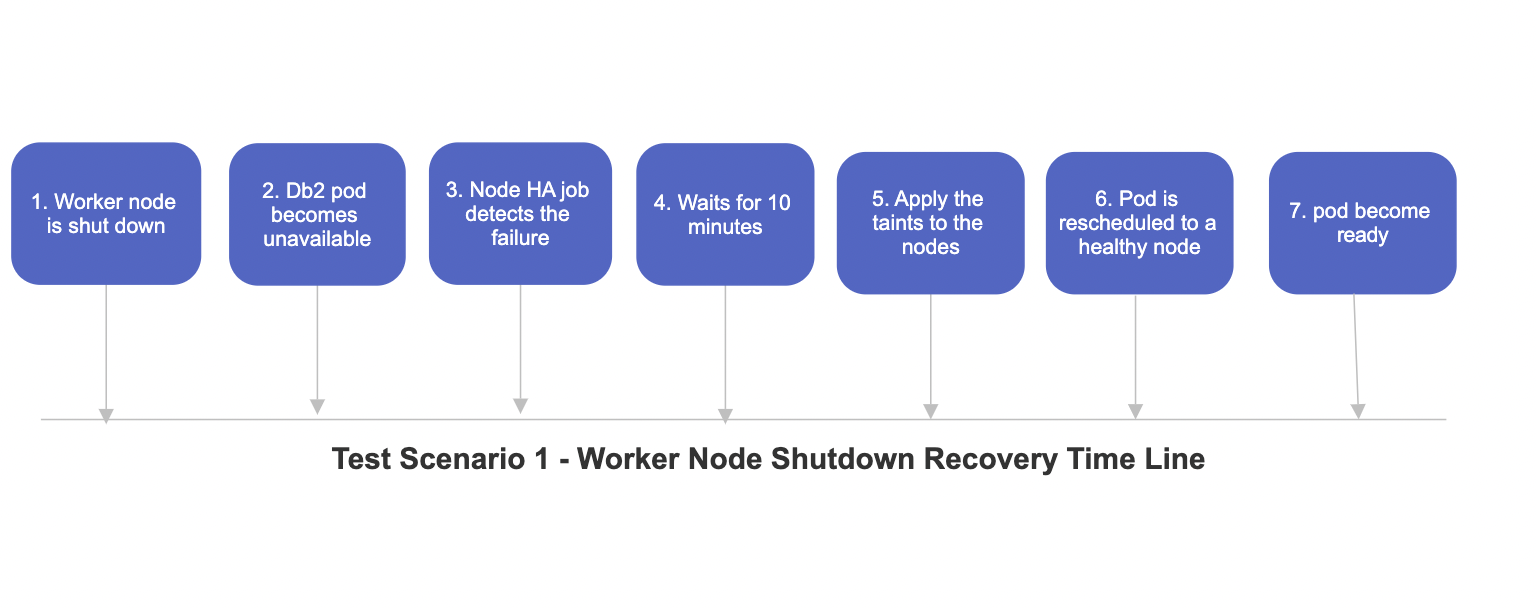
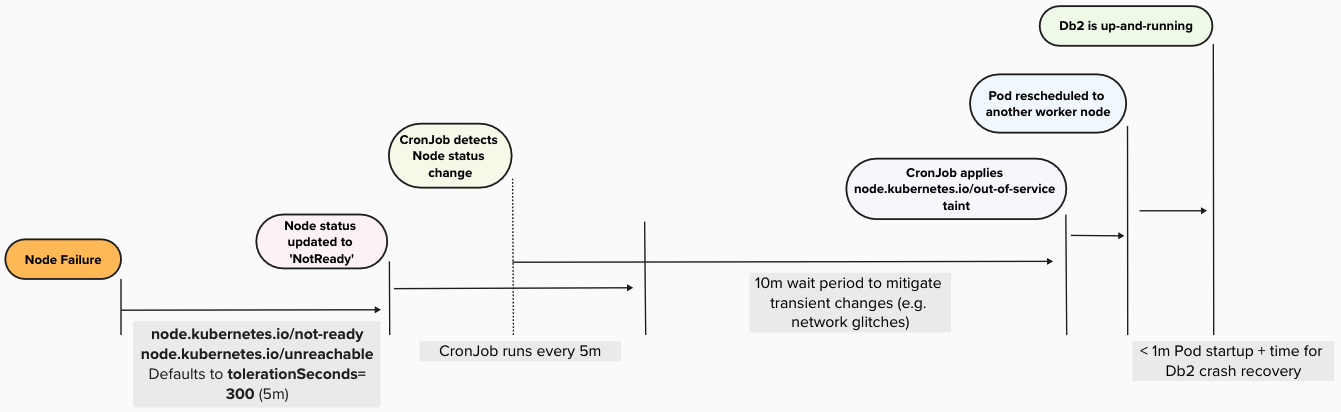
Thanks to the **Node-HA CronJob**, the failure was detected and handled automatically. After a 10-minute grace period (as defined in the script), the cronjob identified the node as still unresponsive and applied a taint to mark it as out-of-service. This action allowed the Pod to be rescheduled and successfully started on a **spare, healthy node**.

**Result:**

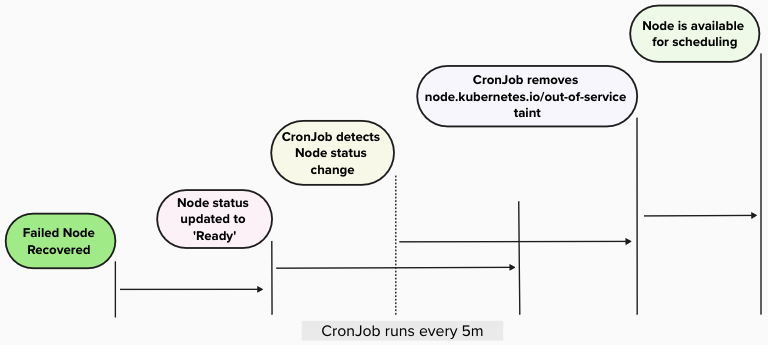
* The Pod was automatically recovered on a different node.
* No manual intervention was required.
* Application availability was maintained with minimal disruption.

Here is a visual timeline illustrating the sequence of events in **Test Scenario 1:**

**Worker Node Shutdown**:

**Figure 1 – Worker Node Shutdown Recovery Timeline (Node Failure)**



**Figure 2 – Worker Node Shutdown Recovery Timeline (Node Recovery)**

**Notes**:

Starting Kubernetes v1.18 (OpenShift 4.8), [Taint based Evictions](https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/#taint-based-evictions) are applied automatically by the Node Controller when certain Node conditions are true. However, any Pod that is getting scheduled will have tolerationSeconds: 300 tolerations applied to ensure that Pods remain bound to Nodes for 5 minutes after any problem is detected.

**4.4 Test Scenario 2: Rebooting a Worker Node**

In this scenario, we **randomly selected a worker node** that was actively hosting a Db2 Pod and performed a **system reboot**. This test simulates a planned maintenance event or an unexpected system restart.

### **Objective:**

To validate whether the Db2 Pod remains stable and is able to recover on the **same node** after the reboot, without triggering a failover or rescheduling.

### **Behaviour Observed:**

* The node temporarily entered a **NotReady** state during the reboot.
* The Db2 Pod became briefly unavailable but was **not deleted or rescheduled**.
* Once the node came back online, the Pod **automatically resumed** and transitioned back to a **Ready** state.

### **Outcome:**

* The Pod successfully recovered on the **same node** post-reboot.
* No taints were applied, and no rescheduling occurred.
* Application continuity was preserved with minimal disruption.

**5. Conclusion**

The high availability (HA) validation of IBM’s Cloud Rack for Db2 Warehouse Solution demonstrates a robust and resilient architecture capable of handling both soft and hard failover scenarios. Soft failover tests, including single and multiple Pod deletions, confirmed that OpenShift's orchestration and Db2's recovery mechanisms can restore services with minimal downtime. Hard failover scenarios, such as node shutdowns and reboots, further validated the system's resilience, especially with the integration of the custom Node-HA CronJob.  
  
The Node-HA CronJob played a pivotal role in detecting and recovering from node-level failures that were not handled by default Kubernetes scheduling mechanisms. Its proactive approach ensured that Db2uEngine custom resource Pods and their Persistent Volumes were detached from failed node(s) and attached to other running node(s) facilitating graceful recovery, and maintaining application availability without manual intervention.  
  
Overall, IBM’s Db2 Warehouse on Power Cloud Rack provides a highly available and production-ready environment for Db2 Warehouse deployments, ensuring business continuity and operational efficiency even under adverse conditions.

**About the Authors**

**Muhammed Hisham P** is a seasoned QA/Test Engineer with expertise in functional validation and System Verification Testing (SVT) for IBM’s Db2 Warehouse on Power Cloud Rack. In his current role, he ensures the reliability and resilience of the solution by designing and executing comprehensive QA scenarios that simulate and validate potential failure conditions. Hisham also leads initiatives to advance test automation—covering unit, integration, and functional layers—to enhance the efficiency and consistency of Db2 Warehouse on Power Cloud Rack deployment validations. He can be contacted at [Muhammed.Hisham.P@ibm.com](mailto:Muhammed.Hisham.P@ibm.com).

**Aruna De Silva** is the architect for Db2/Db2 Warehouse containerized offerings on IBM Cloud Pack for Data, OpenShift and Kubernetes. He has nearly 20 years of database technology experience and is based off IBM Toronto software laboratory. Since 2015, he has been actively involved with modernizing Db2 bringing *Db2 Warehouse – Common Container*, the first containerized Db2 solution out into production in 2016. Since 2019, he has been primarily focused on bringing the success of Db2 Warehouse into cloud native platforms such as OpenShift and Kubernetes while embracing micro service architecture and deployment patterns. He can be contacted at [adesilva@ca.ibm.com](mailto:adesilva@ca.ibm.com)