cloudnative-pg / **cloudnative-pg**

Code · Issues 184 · **Pull requests 106** · Discussions · Actions · Projects

Fixes Issue 7793: Stuck reconciliation fixes #7794

Edit · Code · ✨ Try the new experience

Open

jmealo wants to merge 1 commit into `cloudnative-pg:main` from `jmealo:feat/fix-issue-7793`

Conversation 14 · Commits 1 · Checks 1 · **Files changed 6**

Changes from **all commits** · File filter · Conversations · Jump to

241 🟩🟩🟩🟩🟩 `internal/controller/cluster_controller.go`

```
        return nil
    }

+ // checkAndClearStuckScalingPhase checks if the cluster is stuck in a scaling phase
+ // but already has the correct number of instances, and clears the phase if needed
+ func (r *ClusterReconciler) checkAndClearStuckScalingPhase(
+     ctx context.Context,
+     cluster *apiv1.Cluster,
+     resources *managedResources,
+ ) error {
+     contextLogger := log.FromContext(ctx)
+
+     // Check if we're in a scaling-related phase
+     scalingPhases := []string{
```

jmealo 10 minutes ago • edited · Contributor · Author

Do these need to come from a const/be-extracted to a helper function `isInScalingPhase`? 🤔

Reply...

Resolve conversation

```
+        "Creating a new replica",
+        "Scaling up",
+        "Scaling down",
+    }
+
+    isInScalingPhase := false
+    for _, phase := range scalingPhases {
+        if cluster.Status.Phase == phase {
+            isInScalingPhase = true
+            break
+        }
+    }
+
+    if !isInScalingPhase {
+        return nil // Not in a scaling phase, nothing to check
+    }
+
+    // Check if we already have the correct number of instances
+    currentInstances := len(resources.instances.Items)
+    desiredInstances := cluster.Spec.Instances
+
+    contextLogger.Debug("Checking scaling phase consistency",
+        "phase", cluster.Status.Phase,
+        "currentInstances", currentInstances,
+        "desiredInstances", desiredInstances,
+        "statusInstances", cluster.Status.Instances)
+
+    // If we have the right number of instances and no running jobs, clear the scaling phase
+    if currentInstances == desiredInstances && len(resources.runningJobNames()) == 0 {
+        contextLogger.Info("Clearing stuck scaling phase - cluster has correct number of instances",
+            "phase", cluster.Status.Phase,
+            "instances", currentInstances)
+
+        r.Recorder.Eventf(cluster, "Normal", "ScalingPhaseCleared",
+            "Cleared stuck scaling phase '%s' - cluster has correct number of instances (%d)",
+            cluster.Status.Phase, currentInstances)
+
+        // Clear the phase to allow normal operation
+        if err := r.RegisterPhase(ctx, cluster, apiv1.PhaseHealthy, ""); err != nil {
+            return fmt.Errorf("failed to clear stuck scaling phase: %w", err)
+         }
+     }
+
+     return nil
+ }
+
+ // clearStuckScalingPhaseAfterJobDeletion clears scaling phases after
  deleting failed jobs
+ func (r *ClusterReconciler) clearStuckScalingPhaseAfterJobDeletion(
+     ctx context.Context,
+     cluster *apiv1.Cluster,
+ ) error {
+     contextLogger := log.FromContext(ctx)
+
+     // Check if we're in a scaling-related phase that might be stuck due
  to failed jobs
```

jmealo 7 minutes ago Contributor Author

TODO: Extract this to `isInScalingPhase` method.

Reply...

Resolve conversation

```
+     scalingPhases := []string{
+         "Creating a new replica",
+         "Scaling up",
+         "Scaling down",
+     }
+
+     isInScalingPhase := false
+     for _, phase := range scalingPhases {
+         if cluster.Status.Phase == phase {
+             isInScalingPhase = true
+             break
+         }
+     }
+
+     if !isInScalingPhase {
+         return nil // Not in a scaling phase
+     }
+
+     contextLogger.Info("Clearing scaling phase after job deletion to
  allow retry",
+         "phase", cluster.Status.Phase)
```

```
+
+	r.Recorder.Eventf(cluster, "Normal", "ScalingPhaseReset",
+		"Reset scaling phase '%s' after deleting failed job to allow retry",
+		cluster.Status.Phase)
+
+	// Clear the phase reason to allow the scaling operation to be retried
+	// We don't set it to healthy here because the scaling operation should be retried
+	cluster.Status.PhaseReason = ""
+
+	return r.Status().Update(ctx, cluster)
+ }
+
  // reconcileResources updates all the objects managed by the controller
  func (r *ClusterReconciler) reconcileResources(
  	ctx context.Context, cluster *apiv1.Cluster,
  	contextLogger := log.FromContext(ctx)
  	runningJobs := resources.runningJobNames()

+	// Check if we're stuck in a scaling phase but already have the correct number of instances
+	if err := r.checkAndClearStuckScalingPhase(ctx, cluster, resources); err != nil {
+		contextLogger.Error(err, "Error while checking stuck scaling phase")
+		return ctrl.Result{}, err
+	}
+
  	// Act on Pods and PVCs only if there is nothing that is currently being created or deleted

  	if len(runningJobs) > 0 {
+		// Let's check for failed or stuck jobs and handle them
+		stuckJobTimeout := 10 * time.Minute // Jobs stuck for more than 10 minutes are considered failed
```

jmealo 7 minutes ago Contributor Author

This feels arbitrary but I'm not sure what the cut-off should be?

Reply...

Resolve conversation

```
+
+        for _, job := range resources.jobs.Items {
+            if !utils.IsJobFailedOrStuck(job, stuckJobTimeout) {
+                continue
+            }
+
+            // This job is failed or stuck. We need to record the event, delete the job
+            // and reconcile again
+            if utils.IsJobFailed(job) {
+                r.Recorder.Eventf(cluster, "Warning", "FailingJob",
+                    "Job %v is failing, deleting it to retry", job.Name)
+                contextLogger.Warning("Deleting failed job", "jobName", job.Name)
+            } else {
+                r.Recorder.Eventf(cluster, "Warning", "StuckJob",
+                    "Job %v is stuck in pending state, deleting it to retry", job.Name)
+                contextLogger.Warning("Deleting stuck job", "jobName", job.Name,
+                    "creationTime", job.CreationTimestamp.Time,
+                    "active", job.Status.Active,
+                    "succeeded", job.Status.Succeeded,
+                    "failed", job.Status.Failed)
+            }
+
+            if err := r.Delete(ctx, &job, client.PropagationPolicy(metav1.DeletePropagationBackground)); err != nil {
+                contextLogger.Error(err, "Error while deleting failed/stuck job", "jobName", job.Name)
+                return ctrl.Result{}, err
+            }
+
+            // Clear any stuck scaling phases since we're cleaning up failed jobs
+            if err := r.clearStuckScalingPhaseAfterJobDeletion(ctx, cluster); err != nil {
+                contextLogger.Error(err, "Error clearing stuck scaling phase after job deletion")
+            }
+
+            // Requeue the reconciliation to recreate the job
+            return ctrl.Result{Requeue: true}, nil
+        }
+
+        // Check for equilibrium state - if we have been waiting for jobs for too long
+         // and the cluster state hasn't changed, we might be in a stuck state
+         if err := r.checkForEquilibriumState(ctx, cluster, resources); err != nil {
+             contextLogger.Warning("Detected potential equilibrium state", "error", err)
+             // Continue with normal processing to attempt recovery
+         }
+
          contextLogger.Debug("A job is currently running. Waiting", "runningJobs", runningJobs)
          return ctrl.Result{RequeueAfter: 5 * time.Second}, nil
      }

      return nil
  }
+
+ // checkForEquilibriumState detects when the cluster is stuck in an equilibrium state
+ // where jobs are running but making no progress, potentially due to missing PVCs or other issues
+ func (r *ClusterReconciler) checkForEquilibriumState(
+     ctx context.Context,
+     cluster *apiv1.Cluster,
+     resources *managedResources,
+ ) error {
+     contextLogger := log.FromContext(ctx)
+
+     // Check if we have long-running jobs that might be stuck
+     equilibriumTimeout := 15 * time.Minute // Consider equilibrium after 15 minutes
```

jmealo 5 minutes ago Contributor Author

This is somewhat arbitrary, should this be configurable? What would be a sane value.

Reply...

Resolve conversation

```
+
+     for _, job := range resources.jobs.Items {
+         // Skip completed or failed jobs
+         if utils.IsJobComplete(job) || utils.IsJobFailed(job) {
+             continue
+         }
```

```
+
+		// Check if job has been running for too long without progress
+		if job.CreationTimestamp.Add(equilibriumTimeout).Before(time.Now()) {
+			// Check if job has no active pods (stuck in pending)
+			if job.Status.Active == 0 && job.Status.Succeeded == 0 && job.Status.Failed == 0 {
```

jmealo 4 minutes ago Contributor Author

If the job is failing to schedule for a long time, I'd be more inclined to call it stuck and consider it a failure than if it gets scheduled.

Reply...

Resolve conversation

```
+				contextLogger.Warning("Detected job in equilibrium state - no pods created",
+					"jobName", job.Name,
+					"creationTime", job.CreationTimestamp.Time,
+					"age", time.Since(job.CreationTimestamp.Time))
+
+				// Check if there are missing PVCs that might be causing the issue
+				if err := r.checkForMissingPVCs(ctx, cluster, &job); err != nil {
+					return fmt.Errorf("equilibrium state detected: job %s stuck due to missing PVCs: %w",
```

jmealo 4 minutes ago Contributor Author

It might be useful to surface the different error types in the cluster status somehow? 🤔

Reply...

Resolve conversation

```
+						job.Name, err)
+				}
+
+				return fmt.Errorf("equilibrium state detected: job %s has been pending for %v without creating pods",
+                    job.Name, time.Since(job.CreationTimestamp.Time))
+            }
+
+            // Check if job has active pods but they're not making progress
+            if job.Status.Active > 0 {
+                contextLogger.Warning("Detected job with long-running active pods",
+                    "jobName", job.Name,
+                    "activePods", job.Status.Active,
+                    "age", time.Since(job.CreationTimestamp.Time))
+            }
+        }
+    }
+
+    return nil
+ }
+
+ // checkForMissingPVCs checks if a job is stuck due to missing PVCs
+ func (r *ClusterReconciler) checkForMissingPVCs(
+    ctx context.Context,
+    cluster *apiv1.Cluster,
+    job *batchv1.Job,
+ ) error {
+    contextLogger := log.FromContext(ctx)
+
+    // Extract PVC names from job template
+    var requiredPVCs []string
+    for _, volume := range job.Spec.Template.Spec.Volumes {
+        if volume.PersistentVolumeClaim != nil {
+            requiredPVCs = append(requiredPVCs, volume.PersistentVolumeClaim.ClaimName)
+        }
+    }
+
+    if len(requiredPVCs) == 0 {
+        return nil // No PVCs required
+    }
+
+    // Check if required PVCs exist
+    var missingPVCs []string
+    for _, pvcName := range requiredPVCs {
+        pvc := &corev1.PersistentVolumeClaim{}
+        err := r.Get(ctx, types.NamespacedName{
+            Name:      pvcName,
+            Namespace: cluster.Namespace,
+        }, pvc)
+
+        if apierrs.IsNotFound(err) {
```

```
+             missingPVCs = append(missingPVCs, pvcName)
+         } else if err != nil {
+             contextLogger.Error(err, "Error checking PVC existence",
  "pvcName", pvcName)
+         }
+     }
+
+     if len(missingPVCs) > 0 {
+         return fmt.Errorf("missing PVCs: %v", missingPVCs)
+     }
+
+     return nil
+ }
```

435 🟩🟩🟩🟩🟩 internal/controller/cluster_controller_stuck_reconciliation_test.go

```
@@ -0,0 +1,435 @@
+ /*
+ Copyright © contributors to CloudNativePG, established as
+ CloudNativePG a Series of LF Projects, LLC.
+
+ Licensed 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.
+
+ SPDX-License-Identifier: Apache-2.0
+ */
+
+ package controller
+
+ import (
+     "context"
+     "time"
+
+     batchv1 "k8s.io/api/batch/v1"
+     corev1 "k8s.io/api/core/v1"
+     apierrs "k8s.io/apimachinery/pkg/api/errors"
+     metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
+     "k8s.io/apimachinery/pkg/types"
+     "k8s.io/client-go/tools/record"
+     "sigs.k8s.io/controller-runtime/pkg/client"
+     "sigs.k8s.io/controller-runtime/pkg/client/fake"
+
+     apiv1 "github.com/cloudnative-pg/cloudnative-pg/api/v1"
+     schemeBuilder "github.com/cloudnative-pg/cloudnative-
pg/internal/scheme"
+     "github.com/cloudnative-pg/cloudnative-pg/pkg/postgres"
+     "github.com/cloudnative-pg/cloudnative-pg/pkg/utils"
+
+     . "github.com/onsi/ginkgo/v2"
+     . "github.com/onsi/gomega"
+     fakediscovery "k8s.io/client-go/discovery/fake"
+     k8stesting "k8s.io/client-go/testing"
+ )
+
+ var _ = Describe("Stuck Reconciliation Recovery", func() {
+     var (
+         ctx        context.Context
+         reconciler *ClusterReconciler
+         cluster    *apiv1.Cluster
+         namespace  string
+     )
+
+     BeforeEach(func() {
+         ctx = context.Background()
+         namespace = "test-namespace"
+
+         // Create a fake client with the scheme
+         scheme := schemeBuilder.BuildWithAllKnownScheme()
+
+         fakeClient := fake.NewClientBuilder().
+             WithScheme(scheme).
+             WithStatusSubresource(&apiv1.Cluster{}).
+             WithIndex(&batchv1.Job{}, jobOwnerKey, jobOwnerIndexFunc).
+             WithIndex(&corev1.Pod{}, podOwnerKey, func(rawObj
client.Object) []string {
+                 pod := rawObj.(*corev1.Pod)
+                 if ownerName, ok := IsOwnedByCluster(pod); ok {
+                     return []string{ownerName}
+                 }
+                 return nil
+             }).
+             WithIndex(&corev1.PersistentVolumeClaim{}, pvcOwnerKey,
func(rawObj client.Object) []string {
+                 persistentVolumeClaim := rawObj.
(*corev1.PersistentVolumeClaim)
+                 if ownerName, ok :=
IsOwnedByCluster(persistentVolumeClaim); ok {
+                     return []string{ownerName}
+                 }
+                return nil
+            }).
+            Build()
+
+        // Create fake discovery client
+        fakeDiscoveryClient := &fakediscovery.FakeDiscovery{
+            Fake: &k8stesting.Fake{
+                Resources: []*metav1.APIResourceList{},
+            },
+        }
+
+        // Create a fake event recorder
+        fakeRecorder := record.NewFakeRecorder(100)
+
+        // Create the reconciler
+        reconciler = &ClusterReconciler{
+            Client:          fakeClient,
+            Scheme:          scheme,
+            Recorder:        fakeRecorder,
+            DiscoveryClient: fakeDiscoveryClient,
+        }
+
+        // Create a test cluster
+        cluster = &apiv1.Cluster{
+            ObjectMeta: metav1.ObjectMeta{
+                Name:      "test-cluster",
+                Namespace: namespace,
+                UID:       "test-uid",
+            },
+            Spec: apiv1.ClusterSpec{
+                Instances: 3, // Start with 3 instances
+            },
+            Status: apiv1.ClusterStatus{
+                Instances:      3,
+                ReadyInstances: 3,
+                Phase:          apiv1.PhaseHealthy,
+            },
+        }
+
+        // Create the cluster in the fake client
+        Expect(reconciler.Create(ctx, cluster)).To(Succeed())
+    })
+
+    Describe("End-to-End Stuck Reconciliation Recovery", func() {
+        It("should handle scale up → fail → scale down scenario", func() {
+            By("Starting with a healthy 3-instance cluster")
+            Expect(cluster.Spec.Instances).To(Equal(3))
+            Expect(cluster.Status.Phase).To(Equal(apiv1.PhaseHealthy))
```

```
+
+            By("Scaling up to 4 instances")
+            cluster.Spec.Instances = 4
+            Expect(reconciler.Update(ctx, cluster)).To(Succeed())
+
+            By("Simulating the cluster entering 'Creating a new replica' phase")
+            cluster.Status.Phase = "Creating a new replica"
+            cluster.Status.PhaseReason = "Creating replica test-cluster-4-snapshot-recovery"
+            Expect(reconciler.Status().Update(ctx, cluster)).To(Succeed())
+
+            By("Creating a stuck snapshot-recovery job")
+            stuckJob := &batchv1.Job{
+                ObjectMeta: metav1.ObjectMeta{
+                    Name:      "test-cluster-4-snapshot-recovery",
+                    Namespace: namespace,
+                    OwnerReferences: []metav1.OwnerReference{
+                        {
+                            APIVersion: apiv1.SchemeGroupVersion.String(),
+                            Kind:       apiv1.ClusterKind,
+                            Name:       cluster.Name,
+                            UID:        cluster.UID,
+                            Controller: &[]bool{true}[0],
+                        },
+                    },
+                    CreationTimestamp: metav1.Time{Time: time.Now().Add(-20 * time.Minute)}, // Old job
+                },
+                Spec: batchv1.JobSpec{
+                    Template: corev1.PodTemplateSpec{
+                        Spec: corev1.PodSpec{
+                            Volumes: []corev1.Volume{
+                                {
+                                    Name: "pgdata",
+                                    VolumeSource: corev1.VolumeSource{
+                                        PersistentVolumeClaim: &corev1.PersistentVolumeClaimVolumeSource{
+                                            ClaimName: "missing-pvc", // This PVC doesn't exist
+                                        },
+                                    },
+                                },
+                            },
+                            Containers: []corev1.Container{
+                                {
+                                    Name:  "postgres",
+                                    Image: "postgres:15",
+                                },
+                            },
+                        },
+                    },
+                },
+                Status: batchv1.JobStatus{
+                    Active:    0, // No active pods due to missing PVC
+                    Succeeded: 0,
+                    Failed:    0,
+                },
+            }
+            Expect(reconciler.Create(ctx, stuckJob)).To(Succeed())
+
+            By("Creating managed resources with the stuck job")
+            resources := &managedResources{
+                nodes: make(map[string]corev1.Node),
+                jobs: batchv1.JobList{
+                    Items: []batchv1.Job{*stuckJob},
+                },
+                instances: corev1.PodList{
+                    Items: []corev1.Pod{
+                        // Simulate 3 existing healthy instances
+                        createTestPod("test-cluster-1", namespace, cluster),
+                        createTestPod("test-cluster-2", namespace, cluster),
+                        createTestPod("test-cluster-3", namespace, cluster),
+                    },
+                },
+                pvcs: corev1.PersistentVolumeClaimList{
+                    Items: []corev1.PersistentVolumeClaim{
+                        // Only PVCs for existing instances, missing the one for instance 4
+                        createTestPVC("test-cluster-1-pgdata", namespace, cluster),
+                        createTestPVC("test-cluster-2-pgdata", namespace, cluster),
+                        createTestPVC("test-cluster-3-pgdata", namespace, cluster),
+                    },
+                },
+            }
+
+            By("Testing stuck job handling through reconcileResources")
+            // The reconcileResources method checks for stuck jobs and deletes them
+           // Pass an empty PostgresqlStatusList as it's not needed for
stuck job detection
+           var instancesStatus postgres.PostgresqlStatusList
+           result, err := reconciler.reconcileResources(ctx, cluster,
resources, instancesStatus)
+           Expect(err).ToNot(HaveOccurred())
+           Expect(result.Requeue).To(BeTrue(), "Should requeue after
deleting stuck job")
+
+           By("Verifying the stuck job was deleted")
+           deletedJob := &batchv1.Job{}
+           err = reconciler.Get(ctx, types.NamespacedName{
+               Name:      stuckJob.Name,
+               Namespace: stuckJob.Namespace,
+           }, deletedJob)
+           Expect(apierrs.IsNotFound(err)).To(BeTrue(), "Stuck job
should be deleted")
+
+           By("Simulating user decision to scale down instead of
retrying")
+           // Refresh cluster state
+           Expect(reconciler.Get(ctx, types.NamespacedName{
+               Name:      cluster.Name,
+               Namespace: cluster.Namespace,
+           }, cluster)).To(Succeed())
+
+           // User scales down to 3 instances
+           cluster.Spec.Instances = 3
+           Expect(reconciler.Update(ctx, cluster)).To(Succeed())
+
+           By("Running checkAndClearStuckScalingPhase with correct
instance count")
+           // Update resources to reflect no running jobs and correct
instance count
+           resources.jobs.Items = []batchv1.Job{} // No more jobs
+
+           err = reconciler.checkAndClearStuckScalingPhase(ctx, cluster,
resources)
+           Expect(err).ToNot(HaveOccurred())
+
+           By("Verifying the scaling phase was cleared")
+           // Refresh cluster state
+           Expect(reconciler.Get(ctx, types.NamespacedName{
+               Name:      cluster.Name,
+               Namespace: cluster.Namespace,
+           }, cluster)).To(Succeed())
+
+           Expect(cluster.Status.Phase).To(Equal(apiv1.PhaseHealthy),
+                "Cluster phase should be cleared to healthy when instance count matches")
+        })
+
+        It("should detect and handle missing PVCs", func() {
+            By("Creating a job that requires a missing PVC")
+            jobWithMissingPVC := &batchv1.Job{
+                ObjectMeta: metav1.ObjectMeta{
+                    Name:              "test-job-missing-pvc",
+                    Namespace:         namespace,
+                    CreationTimestamp: metav1.Time{Time: time.Now().Add(-20 * time.Minute)},
+                },
+                Spec: batchv1.JobSpec{
+                    Template: corev1.PodTemplateSpec{
+                        Spec: corev1.PodSpec{
+                            Volumes: []corev1.Volume{
+                                {
+                                    Name: "data",
+                                    VolumeSource: corev1.VolumeSource{
+                                        PersistentVolumeClaim: &corev1.PersistentVolumeClaimVolumeSource{
+                                            ClaimName: "missing-pvc-name",
+                                        },
+                                    },
+                                },
+                            },
+                            Containers: []corev1.Container{
+                                {Name: "test", Image: "test"},
+                            },
+                        },
+                    },
+                },
+                Status: batchv1.JobStatus{
+                    Active:    0,
+                    Succeeded: 0,
+                    Failed:    0,
+                },
+            }
+
+            By("Checking for missing PVCs")
+            err := reconciler.checkForMissingPVCs(ctx, cluster, jobWithMissingPVC)
+            Expect(err).To(HaveOccurred())
+            Expect(err.Error()).To(ContainSubstring("missing PVCs: [missing-pvc-name]"))
+        })
+
+        It("should detect equilibrium state", func() {
+            By("Creating a long-running job with no progress")
+            oldJob := &batchv1.Job{
+                ObjectMeta: metav1.ObjectMeta{
+                    Name:              "old-stuck-job",
+                    Namespace:         namespace,
+                    CreationTimestamp: metav1.Time{Time: time.Now().Add(-20 * time.Minute)},
+                },
+                Status: batchv1.JobStatus{
+                    Active:    0,
+                    Succeeded: 0,
+                    Failed:    0,
+                },
+            }
+
+            resources := &managedResources{
+                jobs: batchv1.JobList{
+                    Items: []batchv1.Job{*oldJob},
+                },
+            }
+
+            By("Checking for equilibrium state")
+            err := reconciler.checkForEquilibriumState(ctx, cluster, resources)
+            Expect(err).To(HaveOccurred())
+            Expect(err.Error()).To(ContainSubstring("equilibrium state detected"))
+        })
+
+        It("should clear scaling phase after job deletion", func() {
+            By("Setting cluster to a scaling phase")
+            cluster.Status.Phase = "Creating a new replica"
+            cluster.Status.PhaseReason = "Creating replica test-cluster-4"
+            Expect(reconciler.Status().Update(ctx, cluster)).To(Succeed())
+
+            By("Clearing scaling phase after job deletion")
+            err := reconciler.clearStuckScalingPhaseAfterJobDeletion(ctx, cluster)
+            Expect(err).ToNot(HaveOccurred())
+
+            By("Verifying phase reason was cleared")
+            // Refresh cluster state
+            Expect(reconciler.Get(ctx, types.NamespacedName{
+                Name:      cluster.Name,
+                Namespace: cluster.Namespace,
+            }, cluster)).To(Succeed())
```

```
+
+            Expect(cluster.Status.PhaseReason).To(BeEmpty(),
+                "Phase reason should be cleared to allow retry")
+        })
+    })
+
+    Describe("Job Utility Functions Integration", func() {
+        It("should correctly identify stuck jobs", func() {
+            By("Creating a stuck job")
+            stuckJob := batchv1.Job{
+                ObjectMeta: metav1.ObjectMeta{
+                    CreationTimestamp: metav1.Time{Time: time.Now().Add(-15 * time.Minute)},
+                },
+                Status: batchv1.JobStatus{
+                    Active:    0,
+                    Succeeded: 0,
+                    Failed:    0,
+                },
+            }
+
+            By("Verifying job is detected as stuck")
+            isStuck := utils.IsJobStuck(stuckJob, 10*time.Minute)
+            Expect(isStuck).To(BeTrue())
+
+            isFailedOrStuck := utils.IsJobFailedOrStuck(stuckJob, 10*time.Minute)
+            Expect(isFailedOrStuck).To(BeTrue())
+        })
+
+        It("should correctly identify failed jobs", func() {
+            By("Creating a failed job")
+            failedJob := batchv1.Job{
+                Status: batchv1.JobStatus{
+                    Conditions: []batchv1.JobCondition{
+                        {
+                            Type:   batchv1.JobFailed,
+                            Status: corev1.ConditionTrue,
+                        },
+                    },
+                },
+            }
+
+            By("Verifying job is detected as failed")
+            isFailed := utils.IsJobFailed(failedJob)
+            Expect(isFailed).To(BeTrue())
+
+            isFailedOrStuck := utils.IsJobFailedOrStuck(failedJob, 10*time.Minute)
+             Expect(isFailedOrStuck).To(BeTrue())
+         })
+     })
+ })
+
+ // Helper functions for creating test objects
+
+ func createTestPod(name, namespace string, cluster *apiv1.Cluster)
  corev1.Pod {
+     return corev1.Pod{
+         ObjectMeta: metav1.ObjectMeta{
+             Name:      name,
+             Namespace: namespace,
+             OwnerReferences: []metav1.OwnerReference{
+                 {
+                     APIVersion: apiv1.SchemeGroupVersion.String(),
+                     Kind:       apiv1.ClusterKind,
+                     Name:       cluster.Name,
+                     UID:        cluster.UID,
+                     Controller: &[]bool{true}[0],
+                 },
+             },
+         },
+         Status: corev1.PodStatus{
+             Phase: corev1.PodRunning,
+             Conditions: []corev1.PodCondition{
+                 {
+                     Type:   corev1.PodReady,
+                     Status: corev1.ConditionTrue,
+                 },
+             },
+         },
+     }
+ }
+
+ func createTestPVC(name, namespace string, cluster *apiv1.Cluster)
  corev1.PersistentVolumeClaim {
+     return corev1.PersistentVolumeClaim{
+         ObjectMeta: metav1.ObjectMeta{
+             Name:      name,
+             Namespace: namespace,
+             OwnerReferences: []metav1.OwnerReference{
+                 {
+                     APIVersion: apiv1.SchemeGroupVersion.String(),
+                     Kind:       apiv1.ClusterKind,
+                     Name:       cluster.Name,
+                     UID:        cluster.UID,
+                     Controller: &[]bool{true}[0],
+                 },
```

```diff
+               },
+               Annotations: map[string]string{
+                   utils.PVCStatusAnnotationName: "ready",
+               },
+           },
+           Status: corev1.PersistentVolumeClaimStatus{
+               Phase: corev1.ClaimBound,
+           },
+       }
+ }
```

86 internal/controller/cluster_controller_test.go

```diff
  import (
      "time"

-     cnpgTypes "github.com/cloudnative-pg/machinery/pkg/types"
      batchv1 "k8s.io/api/batch/v1"
      corev1 "k8s.io/api/core/v1"
+     metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
      "k8s.io/apimachinery/pkg/types"
+     "sigs.k8s.io/controller-runtime/pkg/client"
      "sigs.k8s.io/controller-runtime/pkg/reconcile"

      apiv1 "github.com/cloudnative-pg/cloudnative-pg/api/v1"
      "github.com/cloudnative-pg/cloudnative-pg/internal/configuration"
      "github.com/cloudnative-pg/cloudnative-pg/pkg/postgres"
      "github.com/cloudnative-pg/cloudnative-
  pg/pkg/reconciler/persistentvolumeclaim"
      "github.com/cloudnative-pg/cloudnative-pg/pkg/specs"
+     "github.com/cloudnative-pg/cloudnative-pg/pkg/utils"
+     cnpgTypes "github.com/cloudnative-pg/machinery/pkg/types"

      . "github.com/onsi/ginkgo/v2"
      . "github.com/onsi/gomega"
+     apierrs "k8s.io/apimachinery/pkg/api/errors"
  )

- var _ = Describe("Filtering cluster", func() {
-     metrics := make(map[string]string, 1)
-     metrics["a-secret"] = "test-version"
+ var _ = Describe("reconcileResources", func() {
+     var env *testingEnvironment

-     cluster := apiv1.Cluster{
-         Spec: apiv1.ClusterSpec{
-             ImageName: "postgres:13.0",
-         },
-         Status: apiv1.ClusterStatus{
```

```diff
-			SecretsResourceVersion: apiv1.SecretsResourceVersion{Metrics: metrics},
-			ConfigMapResourceVersion: apiv1.ConfigMapResourceVersion{Metrics: metrics},
-		},
-	}
+	BeforeEach(func() {
+		env = buildTestEnvironment()
+	})

-	items := []apiv1.Cluster{cluster}
-	clusterList := apiv1.ClusterList{Items: items}
+	It("should delete a failed job and requeue", func(ctx SpecContext) {
+		namespace := newFakeNamespace(env.client)
+		cluster := newFakeCNPGCluster(env.client, namespace)

-	It("using a secret", func() {
-		secret := corev1.Secret{}
-		secret.Name = "a-secret"
-		req := filterClustersUsingSecret(clusterList, &secret)
-		Expect(req).ToNot(BeNil())
-	})
+		// Create the CA secrets that the cluster expects
+		generateFakeCASecret(env.client, cluster.GetServerCASecretName(), namespace, "cluster-test")
+		generateFakeCASecret(env.client, cluster.GetClientCASecretName(), namespace, "cluster-test")
+
+		instanceName := cluster.Name + "-1"
+		failedJob := &batchv1.Job{
+			ObjectMeta: metav1.ObjectMeta{
+				Name:      instanceName + "-snapshot-recovery",
+				Namespace: namespace,
+				Labels: map[string]string{
+					utils.ClusterLabelName:      cluster.Name,
+					utils.InstanceNameLabelName: instanceName,
+					utils.JobRoleLabelName:      "snapshot-recovery",
+				},
+			},
+			Status: batchv1.JobStatus{
+				Conditions: []batchv1.JobCondition{
+					{
+						Type:   batchv1.JobFailed,
+						Status: corev1.ConditionTrue,
+					},
+				},
+			},
+		}
+
+        // Create the failed job
+        Expect(env.client.Create(ctx, failedJob)).To(Succeed())
+
+        // Create minimal managed resources for the test
+        managedResources := &managedResources{
+            nodes:     make(map[string]corev1.Node),
+            instances: corev1.PodList{Items: []corev1.Pod{}},
+            pvcs:      corev1.PersistentVolumeClaimList{Items: []corev1.PersistentVolumeClaim{}},
+            jobs:      batchv1.JobList{Items: []batchv1.Job{*failedJob}},
+        }
+
+        // Test the reconcileResources method directly to avoid architecture validation
+        var instancesStatus postgres.PostgresqlStatusList
+        result, err := env.clusterReconciler.reconcileResources(ctx, cluster, managedResources, instancesStatus)
+
+        // Check the result
+        Expect(err).ToNot(HaveOccurred())
+        Expect(result.Requeue).To(BeTrue())

-    It("using a config map", func() {
-        configMap := corev1.ConfigMap{}
-        configMap.Name = "a-secret"
-        req := filterClustersUsingConfigMap(clusterList, &configMap)
-        Expect(req).ToNot(BeNil())
+        // Check if the job was deleted
+        err = env.client.Get(ctx, client.ObjectKeyFromObject(failedJob), failedJob)
+        Expect(err).To(HaveOccurred())
+        Expect(apierrs.IsNotFound(err)).To(BeTrue())
     })
 })

```

14 🟩🟩🟩🟩🟩 internal/controller/suite_test.go

```diff
         WithStatusSubresource(&apiv1.Cluster{}, &apiv1.Backup{}, &apiv1.Pooler{}, &corev1.Service{},
             &corev1.ConfigMap{}, &corev1.Secret{}).
         WithIndex(&batchv1.Job{}, jobOwnerKey, jobOwnerIndexFunc).
+        WithIndex(&corev1.Pod{}, podOwnerKey, func(rawObj client.Object) []string {
+            pod := rawObj.(*corev1.Pod)
+            if ownerName, ok := IsOwnedByCluster(pod); ok {
+                return []string{ownerName}
+            }
+            return nil
+    }).
+    WithIndex(&corev1.PersistentVolumeClaim{}, pvcOwnerKey, func(rawObj client.Object) []string {
+      persistentVolumeClaim := rawObj.(*corev1.PersistentVolumeClaim)
+      if ownerName, ok := IsOwnedByCluster(persistentVolumeClaim); ok {
+        return []string{ownerName}
+      }
+      return nil
+    }).
     Build()
   Expect(err).ToNot(HaveOccurred())

```

71 ▆▆▆▆▆ pkg/utils/jobs.go

```
@@ -0,0 +1,71 @@
+ /*
+ Copyright © contributors to CloudNativePG, established as
+ CloudNativePG a Series of LF Projects, LLC.
+
+ Licensed 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.
+
+ SPDX-License-Identifier: Apache-2.0
+ */
+
+ package utils
+
+ import (
+   "time"
+
+   batchv1 "k8s.io/api/batch/v1"
+   corev1 "k8s.io/api/core/v1"
+ )
+
+ // IsJobFailed check if a job has failed
+ func IsJobFailed(job batchv1.Job) bool {
+   for _, condition := range job.Status.Conditions {
```

```diff
+         if condition.Type == batchv1.JobFailed && condition.Status == corev1.ConditionTrue {
+             return true
+         }
+     }
+     return false
+ }
+
+ // IsJobStuck checks if a job is stuck in pending state for too long
+ func IsJobStuck(job batchv1.Job, timeout time.Duration) bool {
+     // If the job is already marked as failed or complete, it's not stuck
+     if IsJobFailed(job) || IsJobComplete(job) {
+         return false
+     }
+
+     // Check if job has been pending for too long
+     if job.CreationTimestamp.Add(timeout).Before(time.Now()) {
+         // Check if any pods are unschedulable
+         if job.Status.Active == 0 && job.Status.Succeeded == 0 && job.Status.Failed == 0 {
+             // No pods have been created or they're all unschedulable
+             return true
+         }
+     }
+
+     return false
+ }
+
+ // IsJobComplete checks if a job has completed successfully
+ func IsJobComplete(job batchv1.Job) bool {
+     for _, condition := range job.Status.Conditions {
+         if condition.Type == batchv1.JobComplete && condition.Status == corev1.ConditionTrue {
+             return true
+         }
+     }
+     return false
+ }
+
+ // IsJobFailedOrStuck checks if a job has failed or is stuck
+ func IsJobFailedOrStuck(job batchv1.Job, stuckTimeout time.Duration) bool {
+     return IsJobFailed(job) || IsJobStuck(job, stuckTimeout)
+ }
```

279 🟩🟩🟩🟩🟩 pkg/utils/jobs_test.go

```diff
@@ -0,0 +1,279 @@
+ /*
+ Copyright © contributors to CloudNativePG, established as
+ CloudNativePG a Series of LF Projects, LLC.
+
+ Licensed 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.
+
+ SPDX-License-Identifier: Apache-2.0
+ */
+
+ package utils
+
+ import (
+     "testing"
+     "time"
+
+     batchv1 "k8s.io/api/batch/v1"
+     corev1 "k8s.io/api/core/v1"
+     metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
+ )
+
+ func TestIsJobFailed(t *testing.T) {
+     tests := []struct {
+         name     string
+         job      batchv1.Job
+         expected bool
+     }{
+         {
+             name: "job with failed condition",
+             job: batchv1.Job{
+                 Status: batchv1.JobStatus{
+                     Conditions: []batchv1.JobCondition{
+                         {
+                             Type:   batchv1.JobFailed,
+                             Status: corev1.ConditionTrue,
+                         },
+                     },
+                 },
+             },
+             expected: true,
+         },
+         {
+             name: "job without failed condition",
+             job: batchv1.Job{
+                 Status: batchv1.JobStatus{
+                     Conditions: []batchv1.JobCondition{
+                         {
+                             Type:   batchv1.JobComplete,
+                             Status: corev1.ConditionTrue,
+                         },
+                     },
+                 },
+             },
+             expected: false,
+         },
+         {
+             name: "job with no conditions",
+             job: batchv1.Job{
+                 Status: batchv1.JobStatus{},
+             },
+             expected: false,
+         },
+     }
+
+     for _, tt := range tests {
+         t.Run(tt.name, func(t *testing.T) {
+             result := IsJobFailed(tt.job)
+             if result != tt.expected {
+                 t.Errorf("IsJobFailed() = %v, expected %v", result, tt.expected)
+             }
+         })
+     }
+ }
+
+ func TestIsJobComplete(t *testing.T) {
+     tests := []struct {
+         name     string
+         job      batchv1.Job
+         expected bool
+     }{
+         {
+             name: "job with complete condition",
+             job: batchv1.Job{
+                 Status: batchv1.JobStatus{
+                     Conditions: []batchv1.JobCondition{
+                         {
+                             Type:   batchv1.JobComplete,
+                             Status: corev1.ConditionTrue,
+                         },
+                    },
+                },
+            },
+            expected: true,
+        },
+        {
+            name: "job without complete condition",
+            job: batchv1.Job{
+                Status: batchv1.JobStatus{
+                    Conditions: []batchv1.JobCondition{
+                        {
+                            Type:   batchv1.JobFailed,
+                            Status: corev1.ConditionTrue,
+                        },
+                    },
+                },
+            },
+            expected: false,
+        },
+    }
+
+    for _, tt := range tests {
+        t.Run(tt.name, func(t *testing.T) {
+            result := IsJobComplete(tt.job)
+            if result != tt.expected {
+                t.Errorf("IsJobComplete() = %v, expected %v", result, tt.expected)
+            }
+        })
+    }
+ }
+
+ func TestIsJobStuck(t *testing.T) {
+    now := time.Now()
+    timeout := 10 * time.Minute
+
+    tests := []struct {
+        name     string
+        job      batchv1.Job
+        timeout  time.Duration
+        expected bool
+    }{
+        {
+            name: "stuck job - old with no active pods",
+            job: batchv1.Job{
+                ObjectMeta: metav1.ObjectMeta{
+                    CreationTimestamp: metav1.Time{Time: now.Add(-15 * time.Minute)},
+                },
+				Status: batchv1.JobStatus{
+					Active:    0,
+					Succeeded: 0,
+					Failed:    0,
+				},
+			},
+			timeout:  timeout,
+			expected: true,
+		},
+		{
+			name: "not stuck - recent job",
+			job: batchv1.Job{
+				ObjectMeta: metav1.ObjectMeta{
+					CreationTimestamp: metav1.Time{Time: now.Add(-5 * time.Minute)},
+				},
+				Status: batchv1.JobStatus{
+					Active:    0,
+					Succeeded: 0,
+					Failed:    0,
+				},
+			},
+			timeout:  timeout,
+			expected: false,
+		},
+		{
+			name: "not stuck - has active pods",
+			job: batchv1.Job{
+				ObjectMeta: metav1.ObjectMeta{
+					CreationTimestamp: metav1.Time{Time: now.Add(-15 * time.Minute)},
+				},
+				Status: batchv1.JobStatus{
+					Active:    1,
+					Succeeded: 0,
+					Failed:    0,
+				},
+			},
+			timeout:  timeout,
+			expected: false,
+		},
+		{
+			name: "not stuck - already completed",
+			job: batchv1.Job{
+				ObjectMeta: metav1.ObjectMeta{
+					CreationTimestamp: metav1.Time{Time: now.Add(-15 * time.Minute)},
+				},
+				Status: batchv1.JobStatus{
```

```
+                     Active:    0,
+                     Succeeded: 1,
+                     Failed:    0,
+                     Conditions: []batchv1.JobCondition{
+                         {
+                             Type:   batchv1.JobComplete,
+                             Status: corev1.ConditionTrue,
+                         },
+                     },
+                 },
+             },
+             timeout:  timeout,
+             expected: false,
+         },
+     }
+
+     for _, tt := range tests {
+         t.Run(tt.name, func(t *testing.T) {
+             result := IsJobStuck(tt.job, tt.timeout)
+             if result != tt.expected {
+                 t.Errorf("IsJobStuck() = %v, expected %v", result, tt.expected)
+             }
+         })
+     }
+ }
+
+ func TestIsJobFailedOrStuck(t *testing.T) {
+     now := time.Now()
+     timeout := 10 * time.Minute
+
+     tests := []struct {
+         name     string
+         job      batchv1.Job
+         expected bool
+     }{
+         {
+             name: "failed job",
+             job: batchv1.Job{
+                 Status: batchv1.JobStatus{
+                     Conditions: []batchv1.JobCondition{
+                         {
+                             Type:   batchv1.JobFailed,
+                             Status: corev1.ConditionTrue,
+                         },
+                     },
+                 },
+             },
+             expected: true,
+         },
+         {
+             name: "stuck job",
+             job: batchv1.Job{
+                 ObjectMeta: metav1.ObjectMeta{
+                     CreationTimestamp: metav1.Time{Time: now.Add(-15 * time.Minute)},
+                 },
+                 Status: batchv1.JobStatus{
+                     Active:    0,
+                     Succeeded: 0,
+                     Failed:    0,
+                 },
+             },
+             expected: true,
+         },
+         {
+             name: "healthy job",
+             job: batchv1.Job{
+                 ObjectMeta: metav1.ObjectMeta{
+                     CreationTimestamp: metav1.Time{Time: now.Add(-5 * time.Minute)},
+                 },
+                 Status: batchv1.JobStatus{
+                     Active:    1,
+                     Succeeded: 0,
+                     Failed:    0,
+                 },
+             },
+             expected: false,
+         },
+     }
+
+     for _, tt := range tests {
+         t.Run(tt.name, func(t *testing.T) {
+             result := IsJobFailedOrStuck(tt.job, timeout)
+             if result != tt.expected {
+                 t.Errorf("IsJobFailedOrStuck() = %v, expected %v", result, tt.expected)
+             }
+         })
+     }
+ }
```