**V3-TESTS**

**1-NODE DEPLOYMENT AND CONFIGURATION**

**Test Name:**

New node group deployment on cluster using Crossplane claim file

**Description:**

This test validates the ability to deploy a new node group in the cluster using Crossplane claim files. It ensures that the claim file is processed correctly, resources are provisioned in AWS, and the cluster updates its node group configuration successfully.

**Setup:**

1. Tools and Dependencies:

   - Ensure Crossplane is installed and operational on the regional cluster.

   - Argo CD is configured on the regional cluster to manage Crossplane claim files.

2. Environment:

   - An active regional cluster with Crossplane installed.

   - A pre-configured GitOps repository managed by Argo CD for isv-cluster claims.

3. Resources:

   - A Crossplane claim file defining the node group configuration.

**Procedure:**

1. Pre-Test Validation:

   - Confirm that the isv-cluster is operational and connected:

    -   kubectl get nodes --context=<isv-cluster-context>

   - Check Argo CD application sync status for the isv-cluster claims repository:

    -   Argo CD app get <isv-cluster-claim-app>

2. Create Crossplane Claim File:

   - Add a new claim file 'claim.yaml' in the GitLab repository managed by Argo CD

   - Commit and push the file to the Git repository.

3. Monitor Argo CD for Syncing the Claim:

   - Check the Argo CD application status to ensure synchronization

   - Confirm the claim is applied to the regional cluster

4. Verify Node Group Creation in AWS:

   - Check AWS Management Console or use AWS CLI to confirm node group creation

5. Verify Node Group Status in isv-cluster:

   - Confirm new nodes are added to the isv-cluster

**Results:**

- Expected Outcome:

  - The claim file triggers the creation of a new node group in AWS.

  - The isv-cluster registers the new nodes and scales up as per the specified configuration.

- Pass Criteria:

  - The node group is visible in AWS EKS.

  - New nodes are added to the isv-cluster and are in a `Ready` state.

**Observation:**

**Test Name:**

Node Auto-Scaling on isv-cluster Using Crossplane Claim File

**Description:**

This test validates the auto-scaling functionality of a node group within a isv-cluster managed by Crossplane. It ensures that the scaling configuration specified in the claim file is applied and that the cluster scales up or down based on workload demand.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - Argo CD configured to sync Crossplane claim files for isv-clusters.

2. Environment:

   - An active isv-cluster with node groups created and managed via Crossplane.

   - A workload deployed to the isv-cluster capable of generating CPU or memory demand.

3. Resources:

   - A Crossplane claim file with auto-scaling configuration for the node group.

**Procedure:**

1. Pre-Test Validation:

   - Verify that the isv-cluster and node groups are operational

   - Ensure the auto-scaling configuration is defined in the claim file

2. Deploy the Workload:

   - Deploy a workload that can generate resource demand, if necessary

3. Apply the Claim File:

   - Push the updated claim file to the Git repository managed by Argo CD

   - Monitor the Argo CD sync status

4. Monitor Scaling Behavior:

   - Watch the node group in AWS for changes in desired, minimum, or maximum size

   - Check the number of nodes in the cluster as the workload increases

5. Simulate Reduced Demand:

   - Reduce the workload by scaling down the deployment.

   - Verify that the node group scales down after demand decreases.

**Results:**

- Expected Outcome:

  - The node group scales up when workload demand increases and scales down when demand decreases.

  - AWS reflects the changes in node group size as specified by the scaling configuration.

- Pass Criteria:

  - New nodes are added or removed as expected during the test.

  - Cluster remains stable during scaling operations, with no resource contention or node failures.

**Observation:**

**Test Name:**

Node Configuration Validation on Isv-Cluster Using Crossplane Claim File

**Description:**

This test validates that the node group created on the Isv-Cluster using a Crossplane claim file adheres to the specified configuration. The test ensures that instance types, subnets etc. are correctly applied and functional.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - Argo CD managing Crossplane claim files for Isv-Clusters.

2. Environment:

   - A regional cluster managing Isv-Clusters via Crossplane.

   - A Isv-Cluster with pre-existing configurations and permissions to validate against.

3. Resources:

   - A Crossplane claim file defining the node configuration for the Isv-Cluster.

**Procedure:**

1. Pre-Test Validation:

   - Ensure the Isv-Cluster is active and operational:

   - Validate existing node group configurations in AWS:

2. Deploy the Node Group Claim File:

   - Update or create a Crossplane claim file (`node-cliam.yaml`) specifying the desired node configuration:

   - Commit and push the file to the Git repository managed by Argo CD.

3. Sync Argo CD Application:

   - Monitor the sync status of the claim file:

   - Confirm the claim file is applied to the Isv-Cluster

4. Validate Node Configuration in AWS:

   - Check the node group configuration in AWS:

   - Confirm that the instance types, scaling policies, and subnets etc. match the claim file.

5. Validate Nodes in the Isv-Cluster:

   - Check the node instances added to the cluster:

   - Verify that the instance types match the configuration specified in the claim file.

**Results:**

- Expected Outcome:

  - Node group configurations (instance types, scaling policies, subnets, etc) match the claim file.

  - Nodes are registered with the Isv-Cluster and are functional.

  - Workloads are successfully scheduled and executed on the nodes.

- Pass Criteria:

  - AWS reflects the correct node group configuration as defined in the claim file.

  - Isv-Cluster nodes are healthy and match the instance types and subnets specified.

**Observation:**

**Test Name:**

Network Configuration Validation on Isv-Cluster

**Description:**

This test validates the network configuration of a Isv-Cluster created using Crossplane. It ensures that the subnets, VPC, and associated networking resources (security groups, route tables) are configured correctly and align with the specifications in the Crossplane claim file.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed on the regional cluster.

   - Argo CD managing the Isv-Cluster's Crossplane claim files.

2. Environment:

   - An operational Isv-Cluster managed by Crossplane with a defined network configuration.

3. Resources:

   - Crossplane claim file specifying the Isv-Cluster network configuration, including:

     - VPC ID

     - Subnets

     - Security groups

   - Test workloads to validate network connectivity.

**Procedure:**

1. Pre-Test Validation:

   - Check the Isv-Cluster status

   - Validate the presence of networking CRDs in Crossplane

2. Deploy Networking Configuration Claim File:

   - Update or create a Crossplane claim file.

   - Commit and push the claim file to the Git repository managed by Argo CD.

3. Sync Argo CD Application:

   - Monitor the sync status:

   - Confirm that networking resources are provisioned in AWS:

4. Validate Networking Resources in AWS:

   - Verify the VPC, subnets

   - Check security groups and their rules:

5. Test Connectivity Between Nodes:

   - SSH into one of the Isv-Cluster nodes and test connectivity to other nodes using their private IPs:

**Results:**

- Expected Outcome:

  - VPC, subnets, security groups, and routing configurations match the claim file.

  - Nodes in the Isv-Cluster communicate successfully within the configured subnets.

- Pass Criteria:

  - AWS networking resources are correctly configured and visible.

  - No errors in Crossplane reconciliation logs.

  - Successful network connectivity tests between nodes and workloads.

**Observation:**

**Test Name:**

New AMI ID Validation on ISV-Cluster Using Crossplane Claim File

**Description:**

This test ensures that a new AMI ID specified in a Crossplane claim file for a node group is applied correctly to the ISV-Cluster. It validates that the nodes in the ISV-Cluster use the specified AMI ID and confirms the compatibility of the AMI with the cluster setup and workloads.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - Argo CD managing Crossplane claim files for ISV-Clusters.

2. Environment:

   - An operational ISV-Cluster managed by Crossplane.

   - Node group created using a Crossplane claim file.

   - GitOps repository for managing claim files via Argo CD.

3. Resources:

   - A Crossplane claim file specifying the AMI ID for the node group.

**Procedure:**

1. Pre-Test Validation:

   - Verify the ISV-Cluster is active and nodes are operational

   - Ensure the new AMI ID is available and compatible with the cluster

2. Update the Crossplane Claim File:

   - Modify the Crossplane claim file to include the new AMI ID (ng-claim.yaml).

   - Commit and push the claim file to the Git repository managed by Argo CD.

3. Sync the Claim File in Argo CD:

   - Monitor the sync status of the claim file.

   - Confirm that the claim is applied.

4. Validate Node Group Configuration in AWS:

   - Check that the new AMI ID is applied to the node group.

   - Confirm that the launchTemplate reflects the updated AMI ID.

5. Validate Nodes in the ISV-Cluster:

   - Check that new nodes are added using the specified AMI ID.

   - Verify the node image details in AWS EC2 instances.

**Results:**

- Expected Outcome:

  - The new AMI ID is applied to the node group in AWS.

  - Nodes in the ISV-Cluster are provisioned using the specified AMI.

- Pass Criteria:

  - AWS reflects the updated AMI ID in the node group configuration.

  - Nodes are healthy and operational with the new AMI.

**Observation:**

**Test Name:**

Testing High-Availability on ISV-Cluster

**Description:**

This test validates the high-availability (HA) setup of a ISV-Cluster managed through Crossplane. It ensures that the ISV-Cluster can handle node failures, continues to function under stress, and maintains service availability by leveraging multiple availability zones and node groups.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - Argo CD managing Crossplane claim files for ISV-Clusters.

2. Environment:

   - A ISV-Cluster provisioned with nodes distributed across multiple availability zones (AZs).

   - Workloads deployed with appropriate pod anti-affinity rules for HA.

3. Resources:

   - A Crossplane claim file specifying a multi-AZ setup for the node group.

**Procedure:**

1. Pre-Test Validation:

   - Ensure the ISV-Cluster is operational and nodes are distributed across multiple AZs.

   - Verify the node group configuration in AWS.

2. Deploy High-Availability Workload:

   - Deploy a sample workload with at least three replicas and a load balancer.

   - Apply the workload.

3. Simulate Node Failures:

   - Identify nodes hosting the workload pods.

   - Terminate one of the nodes in AWS.

   - Monitor pod rescheduling and node recovery.

4. Simulate Increased Load:

   - Scale the workload to simulate increased traffic.

   - Monitor node autoscaling behavior and ensure load distribution.

**Results:**

- Expected Outcome:

  - The workload remains available despite node failures.

  - Pods are rescheduled automatically to other nodes in different AZs.

  - Autoscaling adds nodes to handle increased traffic.

- Pass Criteria:

  - No service downtime is observed during node failure or recovery.

  - Workload replicas are evenly distributed across AZs.

**Observation:**

**Test Name:**

NodeGroup Rollout on ISV-Cluster

**Description:**

This test validates the rollout of a new or updated NodeGroup in a ISV-Cluster managed using Crossplane. It ensures the correct provisioning of nodes, adherence to the specified configuration, and minimal disruption to workloads during the rollout process.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed on the regional cluster.

   - Argo CD managing Crossplane claim files for ISV-Clusters.

2. Environment:

   - A functioning ISV-Cluster with at least one active NodeGroup.

   - Workloads deployed on the ISV-Cluster to observe impact during the rollout.

3. Resources:

   - A Crossplane claim file defining the new or updated NodeGroup.

**Procedure:**

1. Pre-Test Validation:

   - Verify the ISV-Cluster is active.

   - Confirm the existing NodeGroup configuration in AWS.

2. Prepare the NodeGroup Rollout Claim File:

   - Create or update the Crossplane claim file to define the new NodeGroup or update an existing one.

   - Commit and push the file to the Git repository managed by Argo CD.

3. Sync the Claim File in Argo CD:

   - Monitor the sync process to ensure the NodeGroup claim is applied.

   - Confirm the NodeGroup is being created or updated.

4. Validate NodeGroup Rollout in AWS:

   - Verify the new or updated NodeGroup configuration in AWS:

   - Confirm the desired configuration.

5. Test Node Availability in ISV-Cluster:

   - Check that nodes from the new NodeGroup join the cluster:

   - Confirm that existing workloads are rescheduled to the new nodes if necessary.

6. Deploy Workloads on the Updated NodeGroup:

   - Deploy a test workload to validate functionality.

   - Apply the workload.

   - Confirm the workload is running on the new NodeGroup.

**Results:**

- Expected Outcome:

  - The new NodeGroup is successfully created or updated in AWS and joins the ISV-Cluster.

  - Existing workloads remain unaffected, and new workloads are scheduled on the updated NodeGroup.

  - Configuration matches the specifications in the claim file.

- Pass Criteria:

  - AWS and Kubernetes reflect the updated Node Group configuration.

  - New nodes join the cluster without errors.

  - Workloads are successfully scheduled and operate on the updated NodeGroup.

**Observation:**

**2-Rollback and Recovery**

**Test Name:**

Application Deployment and Rollback Using Helm Claim File

**Description:**

This test validates the deployment and rollback of an application in the ISV-Cluster using a Helm claim file managed by Crossplane and Argo CD. It ensures that the application is deployed as per the Helm chart claim file and that rollback can be performed seamlessly in case of a failed deployment or misconfiguration.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - Argo CD managing Helm claim files for ISV-Clusters.

2. Environment:

   - An operational ISV-Cluster managed by Crossplane.

   - Argo CD repository containing the Helm claim file for the application.

   - A compatible Helm chart for the application stored in a Helm repository.

3. Resources:

   - A Crossplane claim file for the Helm release.

   - Test application Helm chart configuration.

**Procedure:**

1. Pre-Test Validation:

   - Verify ISV-Cluster status:

   - Ensure the Helm repository and chart are accessible:

2. Create the Helm Claim File:

   - Define the Helm claim file for deploying the application ('helm-claim.yaml'):

   - Commit and push the file to the Gitlab repository managed by Argo CD.

3. Sync Argo CD Application:

   - Sync the Helm claim file to trigger the deployment.

   - Verify the application deployment.

4. Simulate Deployment Failure:

   - Update the claim file with an invalid configuration (update worng code in the claim file):

   - Commit and push the changes to simulate a failed deployment.

5. Rollback to Previous Version:

   - Rollback using Argo CD:

   - Validate the application status after rollback:

**Results:**

- Expected Outcome:

  - The application is successfully deployed as per the Helm claim file configuration.

  - In the case of a failure, the rollback restores the application to its previous state without errors.

  - The application remains functional and accessible throughout.

- Pass Criteria:

  - Helm deployment reflects the values and configurations specified in the claim file.

  - Rollback successfully restores the previous state without impacting the ISV-Cluster or other applications.

  - No errors in Crossplane or Argo CD logs during deployment or rollback.

**Observation:**

----

**Test Name:**

Addons Rollback for ISv-Cluster

**Description:**

This test validates the rollback functionality for addons installed in an ISv-Cluster. It ensures that when an addon update or configuration change leads to issues, the ISV-Cluster can revert to the previous working state while maintaining cluster functionality and stability.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - Argo CD managing Crossplane claim files for iSv-Clusters.

2. Environment:

   - An operational EKS Isv-Cluster with addons deployed (vpc-cni,kube-proxy,coredns, etc).

   - Git repository for managing addon configurations via Argo CD.

3. Resources:

   - A Crossplane claim file specifying the addons and their versions.

   - Backup or previous versions of the addon configurations for rollback testing.

**Procedure:**

1. Pre-Test Validation:

   - Verify the IiSv-Cluster is active

   - Confirm the currently installed addons

2. Simulate Addon Update:

   - Update the Crossplane claim file to specify a new addon version (ng-claim.yaml):

   - Commit and push the file to the Git repository managed by Argo CD.

   - Sync the claim file to trigger the addon update.

3. Monitor Addon Update and Test:

   - Verify the addon update status:

   - Test the ISv-Cluster functionality (create resource to validate the updated addons).

4. Simulate Failure:

   - Introduce a misconfiguration or incompatibility in the addon claim file to simulate failure.

   - Sync the claim file and observe failure.

5. Rollback to Previous Version:

   - Revert the claim file to the previous version specifying the working addon configuration.

   - Commit, push, and sync the rollback.

   - Validate the rollback.

6. Post-Rollback Testing:

   - Verify the ISv-Cluster functionality.

   - Confirm workloads and cluster components are functioning correctly.

**Results:**

- Expected Outcome:

  - The updated addon is applied successfully or rolled back seamlessly in case of failure.

  - The IiSv-Cluster and its workloads remain functional and unaffected by the rollback.

  - AWS and Kubernetes reflect the rolled-back addon version.

- Pass Criteria:

  - Addon rollback restores the previous configuration without errors.

  - Cluster nodes and workloads remain operational during and after the rollback.

  - No discrepancies in Crossplane or Argo CD logs related to addon management.

**Observation:**

----

**Test Name:**

Crossplane Resource Recovery

**Description:**

This test validates the ability of Crossplane to recover and reconcile resources in the isv-cluster after an accidental deletion or misconfiguration. It ensures that Crossplane's reconciliation mechanism restores resources to their desired state as defined in the claim files.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - Argo CD managing Crossplane claim files for the isv-cluster.

2. Environment:

   - An operational isv-cluster managed by Crossplane.

   - Resources provisioned in the isv-cluster using Crossplane(workloads).

3. Resources:

   - A Crossplane claim file defining resources to be managed (workloads/resources).

   - Permissions allowing resource creation and management in AWS and Kubernetes.

**Procedure:**

1. Pre-Test Validation:

   - Ensure resources are currently active and provisioned.

   - Validate the health and status of the isv-cluster resources.

2. Simulate Resource Deletion:

   - Identify a managed resource (workloads) and delete it manually.

   - Confirm the resource is deleted.

3. Monitor Crossplane Reconciliation:

   - Verify that the resource is recreated:

   - Confirm the resource status in AWS

4. Simulate Misconfiguration:

   - Update the resource directly in AWS or Kubernetes to create a configuration drift.

   - Verify the drift.

5. Test Recovery from Misconfiguration:

   - Monitor Crossplane reconciliation to detect and revert the changes:

   - Confirm the resource matches the original claim file configuration:

6. Post-Recovery Validation:

   - Verify that the restored resource is functional and meets the desired state.

   - Confirm workloads are unaffected during recovery.

**Results:**

- Expected Outcome:

  - Crossplane successfully recreates deleted resources and reconciles misconfigured resources to the desired state.

  - No disruptions occur to other managed resources or workloads during the recovery process.

- Pass Criteria:

  - Deleted resources are restored as per the claim file.

  - Misconfigured resources are corrected to match the defined configuration.

  - Crossplane logs confirm successful reconciliation without errors.

**Observation:**

----

**Test Name:**

Node Failure Recovery

**Description:**

This test validates the isv-cluster's ability to recover from node failures. It ensures that workloads are rescheduled onto healthy nodes and that the cluster auto-scaler provisions new nodes when needed.

**Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI configured for ISV-cluster access.

   - AWS CLI installed and configured to manage EKS resources.

   - Crossplane and Argo CD managing the ISV-cluster and workloads.

   - Cluster auto-scaler configured for the ISV-cluster.

2. Environment:

   - An ISV-cluster provisioned via Crossplane.

   - Workloads deployed to the ISV-cluster, for testing.

3. Resources:

   - Test workload deployed with at least 2 replicas.

   - IAM roles and policies allowing node termination and autoscaler operation.

**Procedure:**

1. Pre-Test Validation:

   - Confirm the ISV-cluster is operational

   - Verify workloads are running as expected

   - Confirm auto scaler is enabled

2. Simulate Node Failure:

   - Identify a node hosting workload pods

   - Terminate the identified node in AWS

   - Verify that the node is marked as NotRead and eventually removed.

3. Monitor Workload Rescheduling:

   - Check that pods from the terminated node are rescheduled onto healthy nodes.

   - Ensure workloads are functional and accessible during rescheduling.

4. Test Auto scaler Behavior:

   - Observe the cluster auto scaler logs to verify new node provisioning.

   - Confirm the new node is added to the cluster.

5. Post-Recovery Validation:

   - Verify all workloads are running without issues.

   - Confirm that service availability remains uninterrupted.

**Results:**

- Expected Outcome:

  - Terminated node is marked as NotReady and removed from the cluster.

  - Workload pods are rescheduled onto healthy nodes without downtime.

  - Cluster auto scaler provisions new nodes if capacity is insufficient.

- Pass Criteria:

  - Workloads remain functional and accessible throughout the test.

  - New nodes are provisioned when required.

  - No errors in auto scaler or Kubernetes logs related to recovery.

**Observation:**

-----

**Test Name:**

Cluster - Rollback a Cluster-Level Configuration Change

**Description:**

This test validates the ability to rollback a cluster-level configuration change in an EKS cluster managed via Crossplane. It ensures that misconfigured or undesirable changes to the cluster are reverted to a previously stable state, maintaining cluster health and functionality.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - Argo CD managing Crossplane claim files for cluster configurations.

   - Kubernetes CLI and AWS CLI configured for access.

2. Environment:

   - An operational EKS cluster with configurations managed by Crossplane.

   - Backup of the previous cluster configuration stored in the GitOps repository.

3. Resources:

   - Crossplane claim file defining the cluster-level configuration.

   - Permissions allowing cluster-level operations in AWS and Kubernetes.

**Procedure:**

1. Pre-Test Validation:

   - Verify the isv-cluster is operational and configuration matches the claim file.

   - Confirm cluster-level configuration.

2. Apply a Configuration Change:

   - Modify the cluster-level configuration in the Crossplane claim file:

   - Commit and push the updated claim file to the Git repository managed by Argo CD.

   - Monitor the sync process.

   - Confirm the changes have been applied.

3. Simulate Misconfiguration or Failure:

   - Will simulate a failure.

   - Push the changes and observe cluster behavior.

4. Rollback the Configuration:

   - Revert the claim file to the previously stable configuration using git checkout to previous version.

   - Sync the reverted configuration in Argo CD.

   - Monitor Crossplane logs for reconciliation.

5. Post-Rollback Validation:

   - Verify the cluster configuration matches the rolled-back claim file.

   - Ensure workloads are functional and unaffected.

**Results:**

- Expected Outcome:

  - The misconfigured cluster-level changes are rolled back to the previous stable state.

  - Workloads remain functional and unaffected throughout the rollback process.

  - The cluster configuration aligns with the claim file after rollback.

- Pass Criteria:

  - Cluster-level configurations are restored successfully without errors.

  - No disruptions to workloads or cluster functionality during the rollback.

  - Crossplane and Argo CD logs confirm successful reconciliation.

**Observation:**

------

**Test Name:**

PV or PVC - Restore After Accidental Deletion or Corruption

**Description:**

This test validates the ability to recover Persistent Volumes (PV) or Persistent Volume Claims (PVC) in the isv-cluster after accidental deletion or corruption. It ensures that the storage configurations defined via Crossplane are restored to their desired state without data loss.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - Argo CD managing PVC claim files for the isv-cluster.

   - Kubernetes CLI configured for cluster access.

   - Backup solution (Velero/Snapshots) enabled for persistent volumes.

2. Environment:

   - An operational EKS cluster with workloads using PVC-backed storage.

   - PVCs configured with appropriate storage classes.

3. Resources:

   - Crossplane claim file defining the PVC configuration.

   - Test application utilizing a PVC.

   - Backup snapshot of the PV if available.

**Procedure:**

1. Pre-Test Validation:

   - Verify the cluster is operational and PVCs are bound.

   - Check the associated PV and its status.

   - Confirm the application using the PVC is functioning correctly.

2. Simulate PVC or PV Deletion:

   - Identify the PVC or PV associated with a workload.

   - Delete the PVC to simulate accidental deletion.

   - Observe the impact on the associated workload.

3. Restore PVC or PV:

   - Option 1: If backed by a snapshot or backup tool.

     - Restore the PVC using the backup.

     - Verify the restored PVC.

     - Validate the restored PVC.

4. Verify Workload Recovery:

   - Restart the workload or redeploy the application to use the restored PVC.

   - Confirm the application is functional and data is intact.

**Results:**

- Expected Outcome:

  - The PVC or PV is restored to its original state.

  - Workloads utilizing the PVC are functional and their data is present.

  - The cluster remains stable and operational throughout the process.

- Pass Criteria:

  - Restored PVC is in Bound status and associated with the correct workload.

  - Application functionality and data consistency are verified after recovery.

  - No errors in logs related to PVC or PV restoration.

**Observation:**

------

**Test Name:**

Application State Recovery

**Description:**

This test validates the ability to recover an application’s state in the EKS cluster after unexpected disruptions, such as pod crashes, data loss, or configuration corruption.

**Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI configured for cluster access.

   - Crossplane and Argo CD managing workloads and persistent resources.

2. Environment:

   - An operational EKS cluster with workloads deployed using Helm claim files.

3. Resources:

   - Application workload with persistent storage and service endpoint.

**Procedure:**

1. Pre-Test Validation:

   - Verify the application is running and functional.

   - Confirm the application’s persistent storage.

2. Simulate Application Disruption:

   - Delete or crash the application pods to simulate failure.

   - Check the application logs for error details.

3. Recover Application State:

   - If backup is available:

     - Restore the application’s state from a backup or snapshot.

     - Restart the application after data recovery.

   - If no backup is available:

     - Re-deploy the application using its Helm chart or Kubernetes manifest.

5. Validate Application Recovery:

   - Verify the application’s pods are running and functional.

   - Test the application endpoint to ensure state recovery.

**Results:**

- Expected Outcome:

  - The application is successfully restored to a functional state with its data intact.

  - Kubernetes self-healing mechanisms restart the application if a pod crash occurs.

- Pass Criteria:

  - Application pods are running and functional after recovery.

  - Data consistency is verified, and no loss is observed.

  - Application services remain accessible and operational post-recovery.

**Observation:**

---------

**Test Name:**

Recover the Cluster from Complete Failure

**Description:**

This test validates the ability to recover an EKS cluster from a complete failure, such as when all nodes or control plane components become inaccessible or corrupted. The recovery process ensures that the cluster is restored to a functional state with workloads and configurations intact, leveraging Crossplane and GitOps practices.

**Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - Argo CD managing cluster configuration and workloads.

   - AWS CLI and Kubernetes CLI configured for access.

2. Environment:

   - A fully operational EKS cluster provisioned via Crossplane.

   - Workloads deployed using Helm charts or Kubernetes manifests.

   - Backup or GitOps repository containing the cluster and application configuration.

3. Resources:

   - Backup of cluster state.

   - Crossplane claim file defining the cluster infrastructure.

**Procedure:**

1. Pre-Test Validation:

   - Confirm the cluster is operational and workloads are running.

   - Ensure backups exist for critical data.

2. Simulate Complete Cluster Failure:

   - Simulate a critical failure by terminating all cluster nodes.

   - Verify that the cluster becomes non-functional.

   - Attempt to access the cluster control plane to confirm inaccessibility.

3. Initiate Cluster Recovery:

   - Step 1: Restore the Control Plane:

       - Sync the claim file in Argo CD.

   - Step 2: Restore Worker Nodes:

       - Monitor Node Group creation.

   - Step 3: Restore Persistent Volumes and Workloads:

     - Use Velero or snapshots to restore PVCs.

     - Resync workloads and applications via Argo CD.

4. Validate Cluster and Workloads:

   - Verify nodes are restored and Ready.

   - Confirm workloads are running:

   - Test the functionality of services and endpoints.

**Results:**

- Expected Outcome:

  - The EKS cluster is fully restored to its previous state.

  - Workloads and services are operational and accessible.

  - Persistent data is intact if a backup exists.

- Pass Criteria:

  - Restored cluster matches the configuration in the Crossplane claim file.

  - Nodes and workloads are healthy and functional after recovery.

  - No errors or inconsistencies in Crossplane and Argo CD logs.

**Observation:**

-------------

**3 - UPGRADE Validation**

**### Test Name:**

EKS Cluster Version Upgrade

**### Description:**

This test validates the process of upgrading the Kubernetes version of an EKS cluster managed via Crossplane. It ensures that the upgrade process is seamless, workloads remain unaffected, and the cluster and its resources operate as expected with the new version.

**### Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - ArgoCD managing the EKS cluster configuration.

   - Kubernetes CLI (`kubectl`) and AWS CLI configured for cluster access.

2. Environment:

   - An operational EKS cluster provisioned via Crossplane with workloads deployed.

   - The desired Kubernetes version supported by AWS.

3. Resources:

   - Crossplane claim file defining the EKS cluster configuration.

   - Workloads deployed to validate functionality before and after the upgrade.

**### Procedure:**

1. Pre-Test Validation:

   - Verify the cluster is operational and workloads are running.

   - Confirm the current EKS version.

   - Backup critical cluster resources.

2. Prepare for Upgrade:

   - Identify the desired Kubernetes version.

   - Update the Crossplane claim file to specify the new version.

   - Commit and push the updated claim file to the Git repository managed by ArgoCD.

3. Initiate the Upgrade:

   - Sync the updated claim file to trigger the upgrade.

4. Validate Upgrade Completion:

   - Confirm the upgraded version in AWS(using consoe ort cli).

   - Check the status of the cluster and workloads:

5. Upgrade Node Groups:

   - Update the node group configuration in the Crossplane claim file to match the new version:

     - Sync the node group configuration.

   - Verify new nodes are added and the old ones are drained.

**### Results:**

- Expected Outcome:

  - The EKS cluster and node groups are successfully upgraded to the new Kubernetes version.

  - All workloads and cluster components remain functional after the upgrade.

  - No errors or disruptions are observed during the process.

- Pass Criteria:

  - The cluster version matches the specified upgrade version.

  - Node groups are updated, and workloads are rescheduled successfully.

  - Monitoring tools show no critical errors during the upgrade.

**### Observation:**

--------

**### Test Name:**

Addons Version Upgrade

**### Description:**

This test validates the process of upgrading the versions of critical EKS addons (`vpc-cni`, `kube-proxy`, `coredns`) in the cluster. The test ensures that the addon upgrade is seamless, workloads remain unaffected, and the cluster continues to function optimally with the upgraded addons.

**### Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - ArgoCD managing the configuration of EKS addons.

2. Environment:

   - An operational EKS cluster with default or older versions of addons deployed.

3. Resources:

   - Crossplane claim files defining the addon configurations.

   - The desired versions of EKS addons as supported by AWS.

**### Procedure:**

1. Pre-Test Validation:

   - Verify the current versions of the addons in the EKS cluster.

   - Check the cluster status and running workloads.

2. Update Addon Configuration:

   - Modify the Crossplane claim file to specify the new version for an addons.

   - Commit and push the changes to the Git repository managed by ArgoCD.

3. Initiate Addon Upgrade:

   - Sync the updated claim file in ArgoCD.

   - Monitor the upgrade progress in Crossplane logs.

4. Validate Addon Upgrade:

   - Confirm the new version of the addon is applied.

   - Check the status of the upgraded addon in the cluster.

5. Repeat for Other Addons:

   - Update the Crossplane claim file for each addon with the desired version.

   - Sync and validate upgrades for each addon using the steps above.

**### Results:**

- Expected Outcome:

  - The specified addons are successfully upgraded to the desired versions.

  - Workloads and cluster functionality remain unaffected during and after the upgrade.

  - The cluster operates optimally with the upgraded addons.

- Pass Criteria:

  - The versions of all upgraded addons match the specified versions in the claim files.

  - Workloads and services continue to function without errors.

  - No critical issues are observed in the logs or events.

**### Observation:**

--------------

**### Test Name:**

Crossplane Version Upgrade

**### Description:**

This test validates the process of upgrading Crossplane to a newer version in the regional cluster. It ensures that the upgrade is seamless, existing resources and claims remain intact, and Crossplane continues to manage and reconcile the isv-cluster and its associated resources without errors.

**### Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI configured for regional cluster access.

   - Helm installed for managing the Crossplane installation.

   - Git repository managed by ArgoCD for Crossplane configurations.

2. Environment:

   - A regional cluster with Crossplane installed and operational.

   - Resources provisioned in clusters and managed by Crossplane.

   - Helm repository with the desired Crossplane version.

3. Resources:

   - Backup of the current Crossplane installation and resources (if applicable).

**### Procedure:**

1. Pre-Test Validation:

   - Verify the current version of Crossplane installed:

   - Check the health of Crossplane resources:

   - Backup Crossplane CRDs and resource configurations:

2. Prepare for the Upgrade:

   - Identify the target Crossplane version from the Helm repository:

   - Confirm compatibility of the target version with your Kubernetes cluster.

3. Upgrade Crossplane:

   - Update the Crossplane Helm release with the new version:

   - Monitor the Helm upgrade process:

4. Validate Post-Upgrade Functionality:

   - Verify that Crossplane pods are running and healthy:

   - Confirm the new version of Crossplane is installed:

   - Check the status of Crossplane CRDs and managed resources.

   - Test reconciliation of an existing resource:

5. Test a New Resource Claim:

   - Deploy a new resource claim to validate compatibility.

   - Monitor the resource creation:

**### Results:**

- Expected Outcome:

  - Crossplane is successfully upgraded to the target version.

  - Existing resources remain intact and are reconciled without errors.

  - New resources can be created and managed using the upgraded Crossplane.

- Pass Criteria:

  - The Crossplane version matches the target version.

  - All Crossplane-managed resources are functional post-upgrade.

  - No critical errors are observed in logs or resource reconciliation.

**### Observation:**

---------------

**### Test Name:**

Storage Version Upgrade Validation

**### Description:**

This test validates the upgrade of the storage version used by the EKS isv-cluster. It ensures that existing Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) remain functional, workloads are unaffected, and new storage resources utilize the upgraded version.

**### Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI (`kubectl`) configured for isv-cluster access.

   - Crossplane managing the storage configurations.

2. Environment:

   - An operational EKS isv-cluster with workloads using PVCs for persistent storage.

   - Storage class configured for dynamic provisioning(gp2,gp3`).

3. Resources:

   - PVCs and PVs using the existing storage version.

   - Updated storage version to be validated.

**### Procedure:**

1. Pre-Test Validation:

   - Verify existing PVCs and PVs:

   - Confirm workloads are functioning correctly with current storage:

   - Check the storage class details:

2. Update the Storage Class:

   - Modify the storage class configuration (gp2,gp3):

   - Apply the updated storage class:

3. Update Existing PVCs to Use New Storage Version:

   - Identify PVCs to migrate to the new storage class:

   - Migrate a PVC to the new storage class:

     - Backup the PVC data if necessary:

     - Create a new PVC with the updated storage class:

     - Attach the new PVC to the workload.

4. Test New PVC Functionality:

   - Verify the new PVC is bound and functional.

   - Confirm workload functionality with the new PVC.

5. Validate Existing PVCs:

   - Ensure workloads using existing PVCs continue to function without issues.

6. Post-Upgrade Validation:

   - Validate storage class details and ensure the new version is used for new PVCs.

   - Check PV health and status.

7. Clean Up:

   - Remove any test-specific resources created during the upgrade.

**### Results:**

- Expected Outcome:

  - The storage class is successfully upgraded to the new version.

  - Existing PVCs and PVs remain functional and workloads are unaffected.

  - New PVCs utilize the updated storage class with the new version.

- Pass Criteria:

  - The updated storage class is applied and functional.

  - No disruptions to workloads using existing PVCs.

  - New PVCs are provisioned using the upgraded storage class version.

**### Observation:**

--------------

**### Test Name:**

NetworkPolicy Validation After Upgradation

**### Description:**

This test validates the functionality of NetworkPolicies in the EKS cluster after a cluster or network component upgrade. It ensures that the applied NetworkPolicies enforce the expected traffic restrictions and that no unintentional connectivity issues occur after the upgrade.

**### Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI (`kubectl`) configured for cluster access.

   - Crossplane managing the cluster and related resources.

   - A monitoring tool to observe network traffic if required.

2. Environment:

   - An operational EKS cluster with workloads deployed.

   - NetworkPolicies applied to control traffic between pods, namespaces, or external endpoints.

3. Resources:

   - Workloads in multiple namespaces with different NetworkPolicies.

   - Monitoring pods for testing connectivity.

**### Procedure:**

1. Pre-Test Validation:

   - Verify the cluster is operational and workloads are running:

   - Confirm the current NetworkPolicy configurations:

   - Test connectivity between pods using the current NetworkPolicies:

2. Perform the Upgrade:

   - Upgrade the relevant component (e.g., Kubernetes version, CNI plugin):

     - For Kubernetes version:

       - Update the Crossplane claim file with the new version:

     - For CNI plugin:

       - Update the Crossplane addon configuration with the new version:

       - Sync the updated addon configuration:

3. Post-Upgrade Validation:

   - Verify the cluster and workloads are running.

   - Check the status of the upgraded component.

   - Confirm NetworkPolicies are applied and functional.

4. Validate NetworkPolicy Enforcement:

   - Test connectivity between pods and services based on NetworkPolicy rules.

**### Results:**

- Expected Outcome:

  - NetworkPolicies enforce the expected traffic restrictions after the upgrade.

  - No unexpected connectivity issues or policy violations are observed.

  - Updated NetworkPolicies are reconciled and applied correctly.

- Pass Criteria:

  - Traffic flows are consistent with the defined NetworkPolicies.

  - No errors or inconsistencies in policy application.

  - Logs and monitoring tools confirm correct network behavior.

**### Observation:**

-----------------

**### Test Name:**

API Server and CRD Compatibility

**### Description:**

This test validates the compatibility of custom resource definitions (CRDs) with the Kubernetes API server in the EKS sub-cluster, especially after upgrades to the Kubernetes version, CRDs, or other related components. It ensures that CRDs remain functional, their schemas are compatible with the API server, and no errors occur in managing or reconciling custom resources.

**### Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI configured for sub-cluster access.

   - Crossplane managing custom resources and CRDs.

2. Environment:

   - An operational EKS sub-cluster with CRDs deployed.

   - Backup of existing CRDs to restore in case of failure.

3. Resources:

   - CRDs currently installed in the sub-cluster.

   - A backup file containing all CRDs.

**### Procedure:**

1. Pre-Test Validation:

   - Verify the list of installed CRDs:

   - Check the status and functionality of existing custom resources:

   - Ensure no errors or warnings in the Kubernetes API server logs:

2. Perform Kubernetes or CRD Upgrade:

   - Upgrade the Kubernetes cluster version or CRD definitions.

     - For Kubernetes Upgrade:

       Update the cluster version in the Crossplane claim file.

       Sync the updated claim file via ArgoCD.

     - For CRD Update:

       Apply the updated CRD definition.

3. Validate Post-Upgrade Compatibility:

   - Verify that all CRDs are available and functional.

   - Test the creation, update, and deletion of custom resources for the upgraded CRD.

   - Check for schema validation issues or errors in API server logs.

4. Simulate Compatibility Edge Cases:

   - Create a custom resource with invalid or edge-case data to test schema enforcement.

     Apply the resource and observe errors.

5. Reconcile Existing Custom Resources:

   - Test reconciliation of existing custom resources to ensure compatibility.

**### Results:**

- Expected Outcome:

  - CRDs are fully functional and compatible with the upgraded API server.

  - Existing custom resources remain intact and are reconciled without errors.

  - Schema validation issues are handled appropriately, and meaningful errors are logged.

- Pass Criteria:

  - All CRDs are visible and operational after the upgrade.

  - No errors or warnings in API server logs related to CRDs or custom resources.

  - Custom resources can be created, updated, and deleted without issues.

**### Observation:**

---------------

**### Test Name:**

Application Compatibility After Upgrade (Crossplane, EKS, NodeGroup, or Addons Upgrade)

**### Description:**

This test validates the compatibility and functionality of deployed applications in the EKS sub-cluster after upgrading critical components, including Crossplane, EKS, NodeGroups (NG), or addons . It ensures that the applications continue to run without errors, maintain their state, and interact as expected with upgraded components.

**### Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI (`kubectl`) configured for sub-cluster access.

   - ArgoCD managing application configurations.

   - Crossplane managing cluster-level resources and configurations.

2. Environment:

   - A fully operational EKS sub-cluster with applications deployed.

   - Persistent storage and service endpoints configured for the applications.

3. Resources:

   - Applications using various Kubernetes features (deployments, stateful sets, services).

**### Procedure:**

1. Pre-Test Validation:

   - Verify the health and status of the applications:

   - Confirm the functionality of the application endpoints:

2. Perform the Upgrade:

   - Upgrade the targeted component:

     - Crossplane Upgrade:

       Update Crossplane via Helm file

     - EKS Cluster Upgrade.

       Update the cluster version in the Crossplane claim file and sync via ArgoCD.

     - NodeGroup Upgrade.

       Modify the NodeGroup version in the claim file.

     - Addon Upgrade:

       Update the addon version in the Crossplane claim file.

3. Validate Application Compatibility:

   - Verify that all application pods are running.

   - Check the logs of application pods for errors.

   - Test the application endpoints to confirm functionality.

   - Validate persistent storage access and data consistency.

4. Test Application Interaction:

   - Test inter-application communication

5. Simulate Load:

   - Apply load to the application to test its performance post-upgrade.

**### Results:**

- Expected Outcome:

  - Applications continue to function without errors after the upgrade.

  - Application endpoints are accessible, and inter-service communication remains intact.

  - Persistent data is consistent, and no performance degradation is observed.

- Pass Criteria:

  - All application pods are in the `Running` state after the upgrade.

  - Application logs and endpoints confirm normal operation.

  - No errors in Kubernetes logs or events related to the applications.

**### Observation:**

-------------

4 -  Instance type change validation

**### Test Name:**

Node Group Migration to New Instance Type

**### Description:**

This test validates the migration of a Node Group in the EKS cluster to a new instance type. It ensures that the transition is seamless, workloads are rescheduled onto the new nodes without errors, and cluster stability is maintained during the process.

**### Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI (`kubectl`) configured for cluster access.

   - Crossplane managing NodeGroup configurations.

   - ArgoCD managing the GitOps repository for the NodeGroup claim file.

2. Environment:

   - An operational EKS cluster with workloads running on an existing NodeGroup.

3. Resources:

   - Existing NodeGroup with workloads scheduled.

   - Updated instance type.

**### Procedure:**

1. Pre-Test Validation:

   - Verify the current NodeGroup configuration.

   - Check the health and status of workloads.

   - Confirm no pending or unscheduled pods.

2. Update NodeGroup Configuration:

   - Modify the NodeGroup claim file to specify the new instance type.

   - Commit and push the updated claim file to the Git repository managed by ArgoCD.

3. Initiate Migration:

   - Sync the updated NodeGroup configuration in ArgoCD.

   - Monitor the NodeGroup creation and scaling.

4. Drain and Remove Old Nodes:

   - Identify nodes in the old NodeGroup.

   - Drain workloads from old nodes.

   - Delete the old NodeGroup using Crossplane or AWS CLI.

5. Validate Workload Rescheduling:

   - Ensure workloads are rescheduled onto the new nodes.

   - Check pod logs for errors during migration.

**### Results:**

- Expected Outcome:

  - The NodeGroup is successfully migrated to the new instance type.

  - Workloads are rescheduled onto the new nodes without downtime or errors.

  - Old nodes are drained and removed cleanly.

- Pass Criteria:

  - New nodes with the updated instance type join the cluster and become `Ready`.

  - All workloads are rescheduled onto the new nodes.

  - No errors or warnings in pod logs or Kubernetes events.

**### Observation:**

---------

**### Test Name:**

Resource Compatibility Testing

**### Description:**

This test validates the compatibility of Kubernetes resources (e.g., Deployments, StatefulSets, ConfigMaps, Secrets, CRDs) with various components of the EKS cluster, such as Crossplane, upgraded Kubernetes versions, NodeGroups, and addons. It ensures that resources function correctly, adhere to defined specifications, and maintain compatibility after upgrades or changes to the cluster.

**### Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI (`kubectl`) configured for cluster access.

   - Crossplane managing the cluster and associated resources.

   - ArgoCD managing Kubernetes resource definitions.

2. Environment:

   - An operational EKS cluster with multiple Kubernetes resources deployed.

   - Updated specifications for testing new resource compatibility or modifications.

3. Resources:

   - Workloads and services utilizing Kubernetes resources.

**### Procedure:**

1. Pre-Test Validation:

   - Verify the current status of Kubernetes resources.

   - Check the functionality of applications and services.

2. Deploy New or Modified Resources:

   - Create or update Kubernetes resources to test compatibility.

   - Apply the resource configuration.

3. Validate Resource Creation:

   - Verify that the resources are created and operational.

   - Check pod statuses and logs for errors.

4. Simulate Cluster Changes:

   - Upgrade the Kubernetes version or NodeGroup instance types.

   - Modify addons or CRDs used by the resources.

   - Test the impact of these changes on the deployed resources.

5. Validate Post-Change Compatibility:

   - Confirm that resources remain functional and compliant.

   - Test application functionality.

6. Test Edge Cases:

   - Modify resource configurations with invalid or edge-case data to test schema enforcement.

     Apply the ConfigMap and observe behavior.

**### Results:**

- Expected Outcome:

  - Kubernetes resources are created, updated, and managed without errors.

  - Resources maintain compatibility with cluster upgrades and modifications.

  - No schema validation issues or resource management failures are observed.

- Pass Criteria:

  - All resources are in the expected state and functional.

  - Applications and services remain operational throughout the test.

  - No critical errors in Kubernetes logs or events.

**### Observation:**

---------------

**### Test Name:**

Networking Validation Testing

**### Description:**

This test validates the networking configurations and connectivity within the EKS cluster, including pod-to-pod communication, pod-to-service communication, external connectivity, and adherence to NetworkPolicies. It ensures that the cluster networking setup is functioning correctly and complies with the defined rules, even after upgrades or modifications to the cluster.

**### Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI configured for cluster access.

   - Test utilities for validating connectivity.

2. Environment:

   - An operational EKS cluster with multiple pods and services deployed.

   - NetworkPolicies applied to control traffic between pods and services.

3. Resources:

   - Test workloads and services for verifying connectivity.

   - NetworkPolicy configurations for enforcing traffic rules.

**### Procedure:**

1. Pre-Test Validation:

   - Verify the health of the cluster.

   - Confirm the existing NetworkPolicies.

   - Test connectivity between pods.

2. Validate Pod-to-Pod Connectivity:

   - Test communication between pods.

3. Validate Pod-to-Service Connectivity:

   - Deploy a test service and corresponding pod.

   - Verify service functionality.

4. Validate External Connectivity:

   - Test connectivity from a pod to an external endpoint.

   - Validate DNS resolution from pods.

5. Validate NetworkPolicies:

   - Apply a restrictive NetworkPolicy.

   - Verify that communication is blocked as per the policy.

   - Modify the NetworkPolicy to allow specific traffic and re-test.

**### Results:**

- Expected Outcome:

  - Pods communicate as per the configured NetworkPolicies.

  - Pod-to-service and external connectivity function as expected.

  - DNS resolution and traffic flows align with the network configurations.

- Pass Criteria:

  - Successful connectivity tests for allowed traffic.

  - Blocked traffic adheres to the NetworkPolicy rules.

  - No critical errors in logs or Kubernetes events.

**### Observation:**

-------------------

**### Test Name:**

Crossplane Resource Drift Detection

**### Description:**

This test validates Crossplane’s ability to detect and reconcile resource drift, where actual resource configurations deviate from the desired state defined in Crossplane claims. The test ensures that Crossplane identifies and corrects such drifts, maintaining consistency between the infrastructure and the desired state.

**### Setup:**

1. Tools and Dependencies:

   - Crossplane installed and operational on the regional cluster.

   - ArgoCD managing Crossplane claim files for infrastructure and application resources.

   - AWS CLI configured to manually modify resources provisioned via Crossplane.

2. Environment:

   - An operational EKS cluster with resources provisioned using Crossplane.

   - A Git repository containing the Crossplane claim files.

3. Resources:

   - Example resources to test drift detection (e.g., AWS NodeGroups, VPCs).

   - The Crossplane claim file defining the desired resource state.

**### Procedure:**

1. Pre-Test Validation:

   - Verify the current state of managed resources:

   - Confirm the resource configuration matches the claim file:

   - Backup the original resource definitions, if necessary.

2. Simulate Resource Drift:

   - Manually modify a managed resource using AWS CLI to introduce drift.

   - Confirm the drift using AWS CLI or the AWS Management Console.

3. Monitor Crossplane Drift Detection:

   - Observe Crossplane logs for drift detection and reconciliation actions.

   - Check the status of the managed resource.

4. Verify Reconciliation:

   - Confirm that Crossplane has restored the resource to the desired state.

   - Check the resource status in Kubernetes.

5. Test Drift with Multiple Resources:

   - Repeat the process with other managed resources.

   - Introduce intentional misconfigurations and validate reconciliation.

**### Results:**

- Expected Outcome:

  - Crossplane detects resource drift and logs the discrepancies.

  - Drifted resources are reconciled automatically to match the desired state defined in the claim files.

  - No errors or warnings are logged during the reconciliation process.

- Pass Criteria:

  - Resource configurations in AWS match the desired state after reconciliation.

  - Crossplane logs confirm successful detection and reconciliation of drifted resources.

  - No unmanaged or uncorrected discrepancies remain in the infrastructure.

**### Observation:**

-------------------------

**### Test Name:**

Zero-Downtime Upgrade Validation

**### Description:**

This test validates the ability of the EKS cluster and Crossplane-managed resources to perform upgrades (e.g., Kubernetes version, NodeGroups, or addons) without causing downtime or disruption to deployed applications. The focus is on ensuring the availability of services and the integrity of data during the upgrade process.

**### Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI (`kubectl`) configured for cluster access.

   - Crossplane installed and managing infrastructure resources.

2. Environment:

   - An operational EKS cluster with multiple workloads deployed, including services with external endpoints and stateful workloads using Persistent Volumes (PVs).

   - ArgoCD managing cluster and application configurations.

3. Resources:

   - Workloads that can validate service availability during the upgrade.

**### Procedure:**

1. Pre-Test Validation:

   - Verify the cluster health:

   - Confirm application availability:

   - Backup critical application data if applicable:

2. Perform Upgrade:

   - Step 1: Upgrade Kubernetes Version:

     - Update the Kubernetes version in the Crossplane claim file.

       - Sync the updated claim file via ArgoCD.

     - Monitor Crossplane logs for upgrade progress.

   - Step 2: Upgrade NodeGroups:

     - Update the NodeGroup version or instance type in the claim file.

       - Sync the updated NodeGroup configuration via ArgoCD.

     - Monitor node replacement and workload rescheduling.

   - Step 3: Upgrade Addons:

     - Update the addon versions (e.g., CoreDNS, kube-proxy).

       - Sync the addon configuration.

     - Validate the status of upgraded addons.

3. Validate During Upgrade:

   - Validate that workloads remain unaffected.

**### Results:**

- Expected Outcome:

  - Upgrades to the cluster, NodeGroups, and addons are completed without application downtime.

  - Workloads remain functional and accessible throughout the process.

  - The cluster operates with upgraded components in a stable and consistent state.

- Pass Criteria:

  - Application endpoints are accessible without interruptions.

  - No errors or warnings in Kubernetes logs or events related to the upgrade.

  - All upgraded resources reflect the desired versions or configurations.

**### Observation:**

-------------

**### Test Name:**

Resource Deletion Testing

**### Description:**

This test validates the deletion of resources managed by Crossplane in the EKS cluster to ensure that resources are properly removed without leaving orphaned or dangling resources. It confirms that workloads, dependencies, and associated configurations are gracefully cleaned up and the cluster remains stable after resource deletion.

**### Setup:**

1. Tools and Dependencies:

   - Kubernetes CLI configured for cluster access.

   - Crossplane managing the cluster and its resources.

   - AWS CLI configured for monitoring and verifying resource states.

2. Environment:

   - An operational EKS cluster with Crossplane managing resources such as NodeGroups, VPCs, and applications.

3. Resources:

   - A test NodeGroup, VPC, or workload resource to delete during the test.

   - Existing workloads or services associated with the resources.

**### Procedure:**

1. Pre-Test Validation:

   - Verify the current state of the resource to be deleted.

   - Confirm that workloads are functional and stable.

   - Backup critical data or configurations if applicable.

2. Initiate Resource Deletion:

   - Delete the Crossplane claim file for the resource.

   - Monitor Crossplane logs for deletion events.

3. Validate Resource Deletion:

   - Confirm the resource is removed from Kubernetes.

   - Verify the resource is deleted in AWS.

4. Test Dependency Cleanup:

   - Check if associated resources are also cleaned up:

5. Simulate Failure Scenarios:

   - Introduce a failure during deletion and observe Crossplane’s behavior.

   - Monitor Crossplane’s attempt to reconcile or handle the failure.

**### Results:**

- Expected Outcome:

  - Resources managed by Crossplane are deleted successfully.

  - All associated and dependent resources are removed without leaving orphaned resources.

  - The cluster remains stable, and unaffected workloads continue to operate normally.

- Pass Criteria:

  - No remaining traces of the deleted resource in Kubernetes or AWS.

  - No errors or warnings in Crossplane logs related to the deletion process.

  - Workloads unrelated to the deleted resource remain functional.

**### Observation:**

----------------------

**### Test Name:**

Logging and Metrics Validation (Fluentd)

**### Description:**

This test validates the logging and metrics pipeline in the EKS cluster using Fluentd. It ensures that application logs, Kubernetes events, and metrics are correctly collected, processed, and forwarded to the designated log aggregation or monitoring platform (e.g., CloudWatch, Elasticsearch, or Prometheus).

**### Setup:**

1. Tools and Dependencies:

   - Fluentd installed and configured in the EKS cluster.

   - Log aggregation or monitoring platform.

   - Kubernetes CLI configured for cluster access.

2. Environment:

   - An operational EKS cluster with Fluentd DaemonSet deployed for log forwarding.

   - Applications deployed with varying log levels and workloads generating logs.

3. Resources:

   - Fluentd configuration files.

   - Workloads to generate logs.

**### Procedure:**

1. Pre-Test Validation:

   - Verify Fluentd is deployed and running in the cluster.

   - Confirm the target log aggregation platform is reachable.

2. Deploy Applications to Generate Logs:

   - Deploy an application to generate logs for Fluentd.

   - Apply the deployment.

3. Validate Log Collection by Fluentd:

   - Check Fluentd logs for processing activities:

   - Confirm logs are being sent to the aggregation platform.

4. Test Application-Specific Logs:

   - Access the application and generate logs.

   - Verify the logs appear in the aggregation platform.

5. Test Kubernetes Event Logs:

   - Generate Kubernetes events.

   - Confirm these events are captured by Fluentd and appear in the aggregation platform.

**### Results:**

- Expected Outcome:

  - Logs are collected, processed, and forwarded to the aggregation platform without loss.

  - Fluentd handles high log volumes and recovers gracefully from transient failures.

  - Kubernetes events and application logs are visible and searchable in the aggregation platform.

- Pass Criteria:

  - All logs generated by applications and Kubernetes components appear in the aggregation platform.

  - Fluentd logs indicate no errors or data loss during processing.

  - Recovery from failure scenarios occurs without loss of log data.

**### Observation:**