In place restart

Author: Giuseppe Tinti Tomio Anu Reddy
Collaborators: Abdullah Gharaibeh, Tim Hockin, Sergey Kanzhelev, Yuan Wang, Andrey Velichkevich, Kevin Hannon
Status: Approved, See KEP and main implementation

Summary

This document proposes the "in place restart" feature for JobSet. The objective is to speed up the restart time of distributed ML model training, which is traditionally done by recreating all Pods. This is especially important for large scales, where frequent failures that take longer to recover can cost millions. The proposal leverages the incoming `RestartPod` action to enable in place restart of Pods for JobSet workloads, which avoids the overhead of recreating Pods and significantly reduces recovery time. The solution involves adding an agent sidecar to each worker Pod to allow Pods to be restarted in place on demand and updating the JobSet controller to orchestrate group restarts by restarting healthy Pods in place. A benchmark prototype demonstrated a significant reduction in restart time from 2m10s to 10s at 5000 Nodes compared to the current JobSet implementation.

Context

Handling failures when training ML models

A common pattern for training ML models in distributed systems is creating one worker process per Node and having them coordinate to run the workload. If any process fails in this setting, the workers lose sync and all of them must be restarted to restore coordination and resume the training from the last checkpoint.

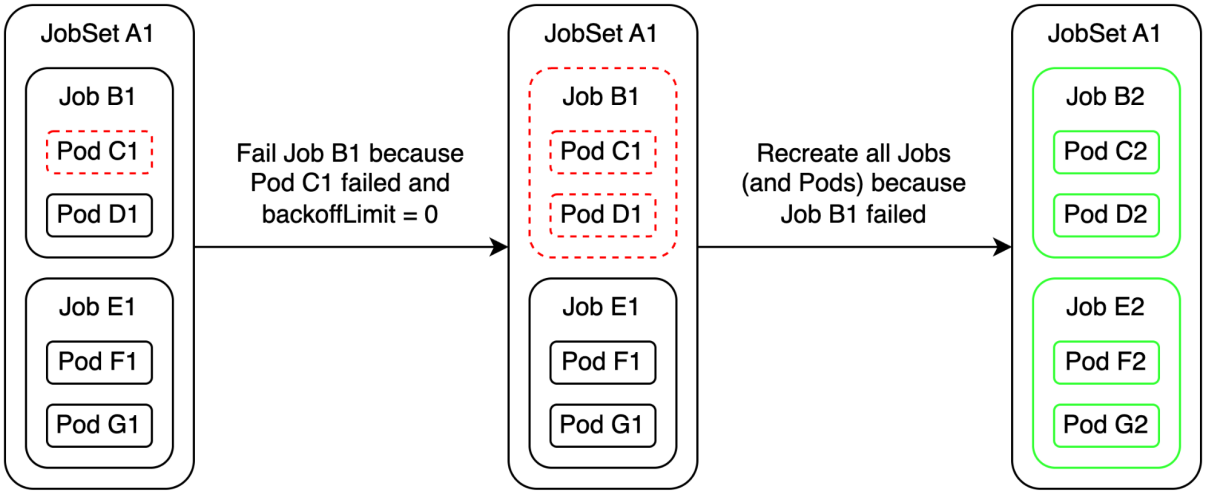
As an example, when using Jax + TPU in GKE, this pattern can be represented in JobSet by:

- Setting `replicas > 1` and `completions = parallelism > 1` to spawn multiple groups of worker Pods
- Setting the annotation `alpha.jobset.sigs.k8s.io/exclusive-topology = cloud.google.com/gke-nodepool` to force 1 Job per Node pool
- Setting the request `google.com/tpu: 4` to force 1 worker Pod per Node by having the worker Pod use all TPU chips in the Node
- Setting `backoffLimit = 0` to trigger a JobSet restart when any failure happens (see more below)

```yaml
apiVersion: jobset.x-k8s.io/v1alpha2
kind: JobSet
metadata:
 name: jobset-example-recreate-all-pods
 annotations:
   # Force 1 Job per Node pool
   alpha.jobset.sigs.k8s.io/exclusive-topology: cloud.google.com/gke-nodepool
spec:
 failurePolicy:
   maxRestarts: ... # Maximum number of full restarts
 replicatedJobs:
 - name: workers
   replicas: ... # Number of Node pools
   template:
     spec:
       completions: ... # Number of Nodes per Node pool
       parallelism: ... # Number of Nodes per Node pool
       backoffLimit: 0 # Must be zero to fail Job if any worker Pod fails
       template:
         spec:
           # Force 1 worker Pod per Node
           resources:
             requests:
               google.com/tpu: 4 # Use all TPU chips in the Node
           containers:
           - name: worker
           ...
```

When a failure happens like a worker container exiting non-zero or a Node failing, the associated Pod will fail. Since `backoffLimit = 0`, the parent Job will fail and trigger a full restart in the parent JobSet. The JobSet controller then recreates all child Jobs, which causes all child Pods to be recreated.

The following diagram shows the process of restarting a JobSet workload by recreating all Pods to recover from a failure.



Financial impact of restarts

Restarting a workload by recreating all Jobs and Pods is good enough for small scales but becomes costly at large scales (>= 1 000 Nodes). This is because of 3 factors:

- The larger the scale the more things exist that can fail, so failures happen more often. It happens every O(hours) at large scales
- The larger the scale the more things exist that must be restarted, so restarts take longer. It takes O(minutes) at large scales
- The larger the scale the more likely the use of expensive hardware like last generation TPUs / GPUs, so the downtime of each Node is more expensive. It costs O(10 dollar) per hour at large scales

Taking everything into account, each extra minute of downtime caused by a longer restart overhead can cost ~ 1 million USD per month as shown below.

Data:

- Nodes: 10 000
- Cost per Node: $16.8 / hour
- Failures: 12 / day
- Extra downtime per failure: 1 minute
- Period: 30 days

$$(10\,000\ nodes) \cdot (\$16.8/60\ per\ minute) \cdot (1\ minute\ per\ failure) \cdot (12\ failures\ per\ day) \cdot (30\ days) = \$$$

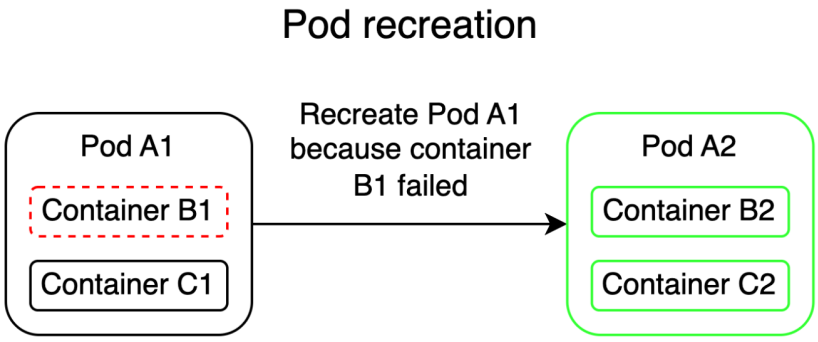
Pod recreation VS In place restart

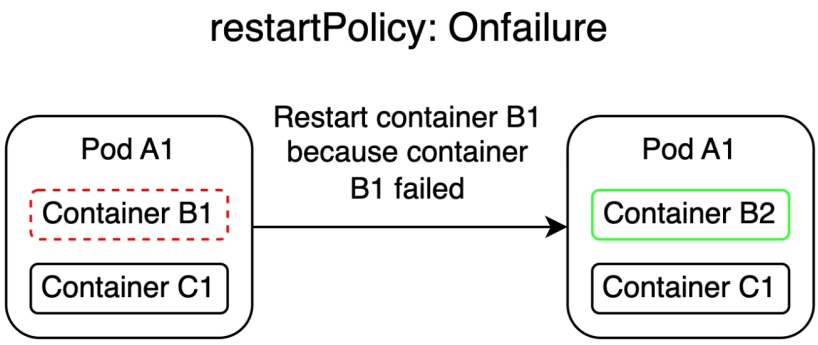
A big contributor to the restart time is the overhead of recreating all child Jobs and Pods. An alternative is to skip the Pod recreation and instead restart only the containers while keeping the

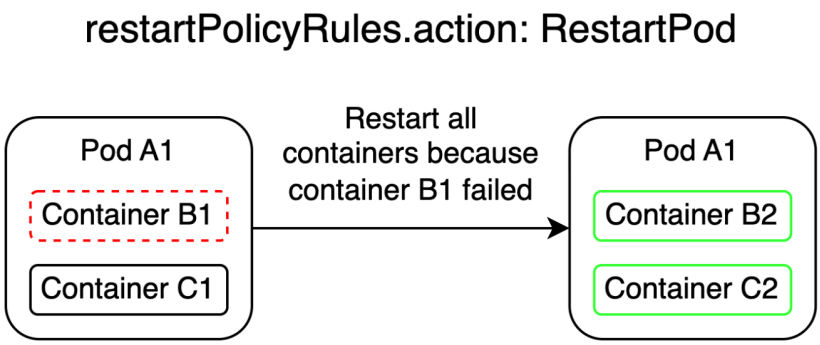
Pods running. We refer to this process as “in place restart”. Currently, this can be achieved by setting `restartPolicy = OnFailure` in the Pod spec, but it is limited to restarting only the failed container and can be triggered by only the failed container.

A more comprehensive API is the RestartPod action which will extend the new restartPolicyRules API (released in version 1.34) and is planned to be released soon (ETA version 1.35). The `RestartPod` action can be triggered by any container exiting a specified exit code and causes all containers to terminate and start in order. In other words, instead of deleting a Pod and creating a new one to replace it, an in place restart can be achieved by forcing one of the containers to exit with a special exit code to trigger `RestartPod` and cause all containers to be restarted while the Pod is still running in the same Node.

The following diagram compares the three options to restart a worker: Pod recreation, restart a single container with `restartPolicy = OnFailure` and restart the whole Pod in place with `restartPolicyRules.action = RestartPod`.







Restarting a Pod in place on demand with a sidecar

To restart a JobSet workload by recreating all Pods, we need a way to force Pods to be recreated. In JobSet, this is achieved by manually deleting and recreating the child Jobs, which causes the child Pods to be deleted and recreated. Similarly for in place restarts, we need a way to force Pods to be restarted in place. Since the `RestartPod` action can be triggered by any container exiting, on demand restarts can be achieved by running an agent sidecar in the worker Pods and forcing the agent to exit with a special exit code.

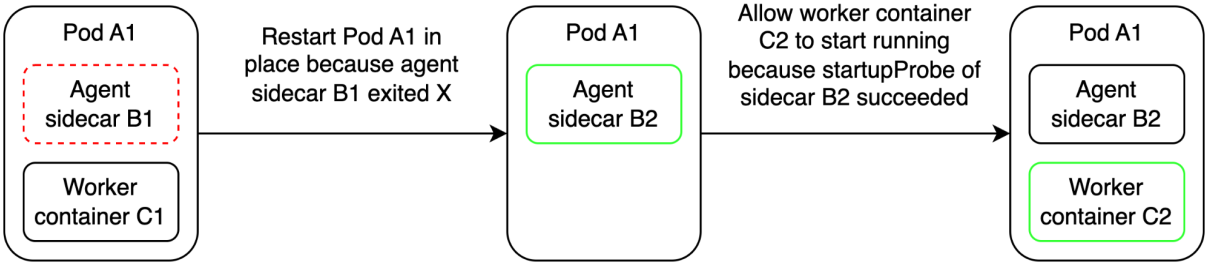
The addition of an agent sidecar also allows the implementation of a barrier at the Pod level. Barriers are important because they make it easier to handle many edge cases that can happen during a workload restart. For instance, without a barrier, a worker container can be restarted twice in the same group restart. The barrier can be achieved by setting up a startup probe for the agent sidecar because startup probes in sidecar containers block the start of the following containers until the probe is ready. In other words, the agent sidecar starts running first because it is also an init container, but the worker container will only start running once the startup probe succeeds.

The following manifest shows an example of a Pod that implements the agent sidecar pattern we described. It allows the Pod to be restarted in place on demand by forcing the agent sidecar to exit `X`. It also allows the Pod to start the worker container only when the agent sidecar creates the endpoint `/barrier-is-lifted`.

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: pod-example-agent-sidecar-pattern
spec:
 initContainers:
 # Agent sidecar
 - name: agent
   # Restart Pod in place if agent exits with exit code X
   # Otherwise, just restart the agent sidecar
   restartPolicy: Always
   restartPolicyRules:
   - action: RestartPod
     exitCodes:
       operator: In
       values: [X]
   # Barrier
   # Only allow the worker container to start when the agent sidecar creates this endpoint
   startupProbe:
     httpGet:
       path: /barrier-is-lifted
       port: 8080
     failureThreshold: ...
     periodSeconds: 1
```

```yaml
containers:
# Worker container
- name: worker
  # Restart Pod in place if worker exits non-zero
  # Otherwise succeed the Pod
  restartPolicy: Never
  restartPolicyRules:
  - action: RestartPod
    exitCodes:
      operator: NotIn
      values: [0]
```

The following diagram shows the same Pod being restarted in place.



Objectives

In this document, we propose the new feature “in place restart” to JobSet. The goal is to reduce the recovery time of failures for large workloads (>= 1 000 Nodes) that use JobSet. We propose to do so by leveraging the incoming `RestartPod` action to skip the overhead of recreating Jobs and Pods. Importantly, the new feature should support all sources of failure (container failure, Pod failure, Job failure and Node failure) and be compatible with the failure policy feature from JobSet and the Pod failure policy feature from Job.

As a secondary objective, restarting the Pods in place will allow for the same worker to run in the same Node after a group restart instead of being scheduled to a different Node, except for cases such as Node failure. This fact will allow optimizations in other components such as caches which are local to the Nodes and worker-specific.

Changes from the previous proposal

The new proposal heavily incorporates feedback from the previous design. Importantly, the new design does not use Redis nor the CRI API. Instead, it fully embraces Kubernetes API

conventions, adopting a level-based and declarative approach. Moreover, this document provides more context about the problem being solved and the required upstream changes.

Proposal

Proposed changes

We propose to add a new feature called “in place restart” to JobSet. This feature will be optional and will allow JobSet workloads to be restarted much faster at large scales (>= 1 000 Nodes). This will be accomplished by restarting healthy Pods in place instead of recreating them. Importantly, the proposed design can handle Pods being restarted or recreated for any reason (e.g., worker container failure, Node failure, on demand in place restart) and for any number of times during a group restart. The benefit is making sure healthy Pods are quickly restarted in place instead of having to be recreated. Additional benefits are supporting an efficient barrier (similar to `BlockingRecreate`) and keeping Pods in the same Nodes.

To do so, we need 3 things:

- A way to restart Pods in place on demand as part of group restarts. This is achieved by an agent sidecar that will run in each worker Pod
- A way to orchestrate group restarts using in place restarts. This is achieved by changing the JobSet controller
- A way to track which epoch each worker Pod is in, similar to the annotation `jobset.sigs.k8s.io/restart-attempt`. We also need a way to state which epoch the worker Pods should be in. This is achieved by changing the JobSet API

More precisely, we propose to achieve these requirements with the following changes to JobSet.

Change 1: reserve a new Pod annotation. Reserve the following Pod annotation to help orchestrate group restarts.

- `jobset.sigs.k8s.io/epoch`
  - Similar to `jobset.sigs.k8s.io/restart-attempt`
  - Its value is the epoch of the worker Pod and should be treated as `int32`
  - If the epoch of any worker Pod exceeds `jobSet.spec.failurePolicy.maxRestarts`, fail the JobSet
  - The annotation is written by the agent sidecar (see change 4)
  - The annotation is read by the JobSet controller (see change 3)

Change 2: change the JobSet API. Make the following changes to the JobSet API to help orchestrate group restarts.

- `jobSet.spec.failurePolicy.restartStrategy = InPlaceRestart`
    - The field value indicates that JobSet restarts should be achieved by restarting healthy Pods in place instead of recreating them. In case a Pod fails, it will be individually recreated. In case a Job fails, it will be individually recreated (along with its child Pods)
    - The field is set by the user in the JobSet manifest
    - The field is read by the JobSet controller (see change 3)
- `jobSet.status.deprecatedEpoch` (`int32`)
    - The field value is the most recent deprecated epoch of the JobSet workload
    - Pods that have an epoch smaller than or equal to this value should be restarted in place
    - The field is written by the JobSet controller (see change 3)
    - The field is read by the agent sidecars (see change 4)
- `jobSet.status.syncedEpoch` (`int32`)
    - The field value is the most recent synced epoch of the JobSet workload
    - Pods that have an epoch equal to this value should lift their barrier to allow the worker containers to start running
    - The field is written by the JobSet controller (see change 3)
    - The field is read by the agent sidecars (see change 4)

```
type FailurePolicy struct {
   MaxRestarts int32 `json:"maxRestarts,omitempty"`

   // +optional
   // +kubebuilder:default=Recreate
   RestartStrategy JobSetRestartStrategy `json:"restartStrategy,omitempty"`

   Rules []FailurePolicyRule `json:"rules,omitempty"`
}

// +kubebuilder:validation:Enum=Recreate;BlockingRecreate;InPlaceRestart
type JobSetRestartStrategy string

const (
   Recreate JobSetRestartStrategy = "Recreate"
   BlockingRecreate JobSetRestartStrategy = "BlockingRecreate"
   InPlaceRestart JobSetRestartStrategy = "InPlaceRestart"
)
```

```go
type JobSetStatus struct {
    // +optional
    // +listType=map
    // +listMapKey=type
    Conditions []metav1.Condition `json:"conditions,omitempty"`

    // +optional
    Restarts int32 `json:"restarts"`

    // +optional
    RestartsCountTowardsMax int32 `json:"restartsCountTowardsMax,omitempty"`

    // +optional
    TerminalState string `json:"terminalState,omitempty"`

    // +optional
    // +listType=map
    // +listMapKey=name
    ReplicatedJobsStatus []ReplicatedJobStatus `json:"replicatedJobsStatus,omitempty"`

    // +optional
    // +kubebuilder:default=0
    DeprecatedEpoch int32 `json:"deprecatedEpoch,omitempty"`

    // +optional
    // +kubebuilder:default=0
    SyncedEpoch int32 `json:"syncedEpoch,omitempty"`
}
```

Change 3: change the JobSet controller. Change the JobSet controller to orchestrate group restarts using in place restarts. To do that, the JobSet controller should be changed to:

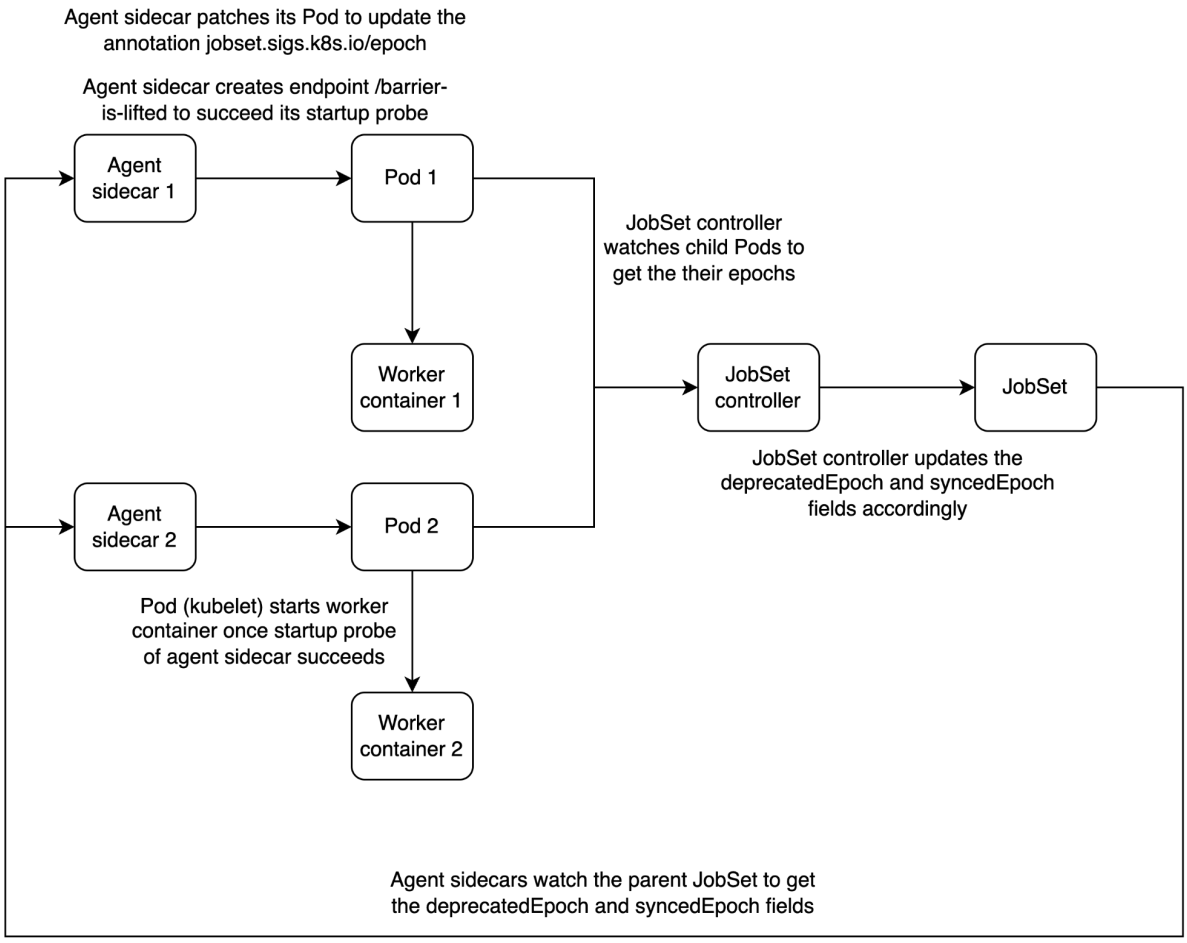
- Watch child Pods
- Index child Pods to make it efficient to list them
- Update the new status fields `deprecatedEpoch` and `syncedEpoch` accordingly (see more in section Implementation)

Change 4: add a new container image for the agent sidecar. Add a new container image that will be buildable from new code in the JobSet repo. The container should be added by the user to the JobSet manifest as the “agent sidecar”. The agent sidecar is responsible for handling in place restart operations at the worker Pod level for the JobSet controller. This new container does 4 things:

- Watch the parent JobSet for changes to the new status fields `deprecatedEpoch` and `syncedEpoch` to act accordingly (see more in section Implementation)
- Similarly to subsection Restarting a Pod in place on demand with a sidecar, the agent sidecar can force its Pod to be restarted in place on demand by exiting with a special exit code
- Similarly to subsection Restarting a Pod in place on demand with a sidecar, the agent sidecar can delay its worker container from starting on demand by succeeding a startup probe
- The agent sidecar calculates its Pod epoch and exposes it by setting and updating the value of Pod annotation `jobset.sigs.k8s.io/epoch`

Change 5: change the JobSet validation webhook. Change the JobSet validation webhook to make sure the manifest spec is compatible when in place restart is enabled. See more at subsection Manifest requirements.

The following diagram summarizes how the components are connected.



Manifest requirements

When in place restart is enabled (i.e., the field `jobSet.spec.failurePolicy.restartStrategy` is set to `InPlaceRestart`), the following is required:

- `jobSet.spec.replicatedJobs[].template.spec.backOffLimit` should be set to `MaxInt32` (i.e., `2147483647`) to avoid unnecessarily failing Jobs if a child Pod fails individually or a Pod is restarted in place
- `jobSet.spec.replicatedJobs[].template.spec.podReplacementPolicy` should be set to `Failed` to make sure the replacement Pod is created only after the original Pod has fully failed
- `jobSet.spec.replicatedJobs[].template.spec.template.spec.initContainers` should contain the agent sidecar to handle in place restart operations at the worker Pod level for the JobSet controller

- Ideally but not necessarily, `jobSet.spec.replicatedJobs[].template.spec.template.spec.containers` should contain a worker container that triggers the `RestartPod` action for recoverable failures

We propose to add documentation to the JobSet website to explain these requirements. Also, we propose to change the JobSet webhook to validate these requirements.

The following manifest shows how the JobSet from subsection Handling failures when training ML models can be modified to use in place restart.

```
apiVersion: jobset.x-k8s.io/v1alpha2
kind: JobSet
metadata:
 name: jobset-example-in-place-restart
 annotations:
   # Force 1 Job per Node pool
   alpha.jobset.sigs.k8s.io/exclusive-topology: cloud.google.com/gke-nodepool
spec:
 failurePolicy:
   maxRestarts: ... # Maximum number of full restarts
   restartStrategy: InPlaceRestart # Enable in place restart
 replicatedJobs:
 - name: workers
   replicas: ... # Number of Node pools
   template:
     spec:
       completions: ... # Number of Nodes per Node pool
       parallelism: ... # Number of Nodes per Node pool
       backoffLimit: 2147483647 # MaxInt32. Avoid full recreation by not failing Jobs due to failed Pods and containers
       podReplacementPolicy: Failed # Make sure the replacement Pod is created only after the original has fully failed
       template:
         spec:
           initContainers:
           # Agent sidecar
           - name: agent
             image: ... # Should be buildable from new code in the JobSet repo
```

```yaml
            # Restart Pod in place if agent exits with exit code X
            # Otherwise, fail Pod
            restartPolicy: Always # Necessary for sidecar
            restartPolicyRules:
            - action: RestartPod
              exitCodes:
                operator: In
                values: [X]
            - action: Terminate
              exitCodes:
                operator: NoIn
                values: [X]
            # Barrier
            # Allow worker container to start only when the agent sidecar creates this endpoint
            startupProbe:
              httpGet:
                path: /barrier-is-lifted
                port: 8080
              failureThreshold: ...
              periodSeconds: 1
            env:
              # Required env variables for agent sidecar
              - name: NAMESPACE
                valueFrom:
                  fieldRef:
                    fieldPath: metadata.namespace
              - name: POD_NAME
                valueFrom:
                  fieldRef:
                    fieldPath: metadata.name
              - name: JOBSET_NAME
                valueFrom:
                  fieldRef:
                    fieldPath: metadata.annotations['jobset.sigs.k8s.io/jobset-name']
              # Optional env variables for agent sidecar
```

```
            # Customize the exit code of agent sidecar for triggering RestartPod
            - name: RESTART_POD_IN_PLACE_EXIT_CODE
              value: "X"
        containers:
        # Worker container
        - name: worker
          # Restart Pod in place if worker exits non-zero exit code
          # Otherwise succeed the Pod
          restartPolicy: Never
          restartPolicyRules:
          - action: RestartPod
            exitCodes:
              operator: NotIn
              values: [0]
        # Force 1 worker Pod per Node
        resources:
          requests:
            google.com/tpu: 4 # Use all TPU chips in the Node
```

Integration with failure policy

In place restart fully supports failure policies and directly depends on the `maxRestarts` and `restartStrategy` fields to be configured. The field `rules` is also supported, but it is not expected to be used except for failing the JobSet in case an unrecoverable failure is detected.

The following manifest shows an example of how in place restart can be integrated with a failure policy containing `action: FailJobSet`. In this example:

- If agent sidecar exits X: restart Pod in place (on demand in place restart)
- If agent sidecar exits anything but X: fail the Pod without failing the parent Job (failure in agent, recreate the Pod individually)
- If worker container exits 0: succeed the Pod to complete the workload (workload finished successfully)
- If worker container exits Y: fail the JobSet no matter the number of restarts so far (unrecoverable failure)
- If worker container exits Z: fail the Pod without failing the parent Job (worker failure that is recoverable with Pod recreation but not Pod in place restart)

```yaml
apiVersion: jobset.x-k8s.io/v1alpha2
kind: JobSet
metadata:
 name: jobset-example-in-place-restart-failure-policy
 annotations:
   # Force 1 Job per Node pool
   alpha.jobset.sigs.k8s.io/exclusive-topology: cloud.google.com/gke-nodepool
Spec:
 failurePolicy:
   maxRestarts: ... # Maximum number of full restarts
   restartStrategy: InPlaceRestart # Enable in place restart
   # Fail JobSet if Job fails with reason PodFailurePolicy
   # Equivalently, fail JobSet if a worker container exits with exit code Y
   rules:
   - action: FailJobSet
     onJobFailureReasons:
     - PodFailurePolicy
 replicatedJobs:
 - name: workers
   replicas: ... # Number of Node pools
   template:
     spec:
       completions: ... # Number of Nodes per Node pool
       parallelism: ... # Number of Nodes per Node pool
       backoffLimit: 2147483647 # MaxInt32. Required to not fail Job due to failed Pods and intentional container restarts
       podReplacementPolicy: Failed # Make sure the replacement Pod is created only after the original has fully failed
       # Fail Job with reason PodFailurePolicy if a worker container exits with exit code Y
       podFailurePolicy:
         rules:
         - action: FailJob
           onExitCodes:
             containerName: worker
             operator: In
             values: [Y]
       template:
```

```yaml
    spec:
      initContainers:
      # Agent sidecar
      - name: agent
        image: ... # Should be buildable from new code in the JobSet repo
        # Restart Pod in place if agent exits with exit code X
        # Otherwise, fail Pod
        restartPolicy: Always # Necessary for sidecar
        restartPolicyRules:
        - action: RestartPod
          exitCodes:
            operator: In
            values: [X]
        - action: Terminate
          exitCodes:
            operator: NoIn
            values: [X]
        # Allow worker container to start only when the agent container creates this endpoint
        startupProbe:
          httpGet:
            path: /barrier-is-lifted
            port: 8080
          failureThreshold: ...
          periodSeconds: 1
        env:
          # Required env variables for agent sidecar
          - name: NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
          - name: POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: JOBSET_NAME
```

```
              valueFrom:
                fieldRef:
                  fieldPath: metadata.annotations['jobset.sigs.k8s.io/jobset-name']
            # Optional env variables for agent sidecar
            # Customize the exit code of agent sidecar for triggering RestartPod
            - name: RESTART_POD_IN_PLACE_EXIT_CODE
              value: "X"
          containers:
          # Worker container
          - name: worker
            # Complete Pod if worker exits 0
            # Fail Pod if worker exits with exit code Y or Z
            # Otherwise, restart Pod in place
            restartPolicy: Never
            restartPolicyRules:
            - action: Terminate
              exitCodes:
                operator: In
                values: [Y, Z]
            - action: RestartPod
              exitCodes:
                operator: NotIn
                values: [0, Y, Z]
            ...
          # Force 1 worker Pod per Node
          resources:
            requests:
              google.com/tpu: 4 # Use all TPU chips in the Node
```

Implementation

High level

At a high level, the implementation of in place restart works as follows:

- **Worker epoch:** each worker Pod has an epoch which is exposed in the annotation `jobset.sigs.k8s.io/epoch`
- **Reconciliation objective:** the objective of the JobSet controller is to make all workers be at the same epoch and only then allow the worker containers to start running
- **Lift barrier:** when all workers are at the same epoch `e`, the JobSet controller sets `syncedEpoch` to this common value `e` (which is equivalent to `syncedEpoch += 1`). This will make the agent sidecars lift their barriers and let the worker containers start running
- **Group restart:** if all worker Pods are in sync but one of them restarts or is recreated, its agent sidecar will set its Pod epoch to `syncedEpoch + 1` at start up to inform the JobSet controller that there is a new epoch and a group restart is required. Consequently, the JobSet controller will notice that the worker Pods lost sync and set `deprecatedEpoch` to `max(epochs) - 1` (equivalent to `syncedEpoch`). This will make the agent sidecars that are not at the most recent epoch `syncedEpoch + 1` restart in place to reach the new epoch

The best way to understand this algorithm is with an example, so we strongly suggest checking this presentation [PUBLIC] In place restart - Step by step example which shows step by step how a JobSet workload is created and recovers from a failure.

Implementation details - Agent

The agent sidecar is responsible for handling in place restart operations at the worker Pod level for the JobSet controller. Basically, it runs the following program.

```python
# Initialize
parentJobSet = getJobSet(env.namespace, env.jobSetName)
mostRecentSyncedEpoch = jobset.status.syncedEpoch
epoch = mostRecentSyncedEpoch + 1
patch = {"metadata" : {"annotations" : {"jobset.sigs.k8s.io/epoch" : epoch}}}
patchPod(env.namespace, env.podName, patch)

# Watch
for event in watchJobSet(env.namespace, env.jobSetName):
    mostRecentDeprecatedEpoch = event.jobSet.status.deprecatedEpoch
    mostRecentSyncedEpoch = event.jobSet.status.syncedEpoch

    # Check if Pod must be restarted in place because its epoch has been
deprecated
    # If so, exit special exit code to trigger RestartPod action
    if epoch <= mostRecentDeprecatedEpoch:
        exit(env.restartInPlaceExitCode)
```

```
    # Check if Pod barrier must be lifted because its epoch has been marked as
synced
    # If so, create endpoint "/barrier-is-lifted" to succeed startup probe
    if epoch == mostRecentSyncedEpoch:
        createEndpoint("/barrier-is-lifted") # Idempotent
```

The highlights are:

- Calculate the Pod epoch at start up as `jobSet.status.syncedEpoch + 1`. This makes sure the worker container will start running only when the JobSet controller updates `jobSet.status.syncedEpoch`. This is done only when all worker Pods are at the same epoch
- Restart the Pod in place if its epoch has been deprecated by checking `epoch <= jobSet.status.deprecatedEpoch`. This is done only when a group restart is necessary
- Lift the Pod barrier if its epoch has been marked as synced by checking `epoch == mostRecentSyncedEpoch`. Again, this is done only when all worker Pods are at the same epoch

The real program behind the agent sidecar will be different in a few aspects. For instance:

- Instead of first getting the parent JobSet during initialization and then setting up a watch over it, the program will only create a watch over the parent JobSet and run all logic inside the same watch loop. This is done to mitigate thundering herd problems (see next point)
- Use backoff jitter (wait a random delay before retrying) to create the JobSet watch. During a group restart, all agent sidecars are expected to start roughly at the same time, which creates a thundering herd problem when all of them try to acquire a watch at the same time

Implementation details - Agent sidecar permissions

To allow the agent sidecar to only update the epoch annotation and no other field, the following combination of RBAC role and VAP role can be used.

```
# Service account
apiVersion: v1
kind: ServiceAccount
```

```yaml
metadata:
 name: in-place-restart-sa
---
# Pod patcher role
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
 name: pod-patcher-role
 namespace: default
rules:
- apiGroups: [""]
 resources: ["pods"]
 verbs: ["patch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
 name: bind-in-place-restart-sa-to-pod-patcher-role
 namespace: default
subjects:
- kind: ServiceAccount
 name: in-place-restart-sa
 namespace: default
roleRef:
 kind: Role
 name: pod-patcher-role
 apiGroup: rbac.authorization.k8s.io
---
# If service account is "in-place-restart-sa", only allow the pod annotation "jobset.sigs.k8s.io/epoch" to be updated
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicy
metadata:
 name: update-only-epoch-pod-annotation
spec:
 failurePolicy: Fail
```

```
 matchConstraints:
   resourceRules:
   - apiGroups:   [""]
     apiVersions: ["v1"]
     operations:  ["UPDATE"]
     resources:   ["pods"]
 validations:
   - expression: >
       request.userInfo.username != 'system:serviceaccount:default:in-place-restart-sa' ||
      (
        oldObject.spec == object.spec &&
        oldObject.metadata.labels == object.metadata.labels &&
        oldObject.metadata.annotations.all(key, key == 'jobset.sigs.k8s.io/epoch' || (key in object.metadata.annotations && oldObject.metadata.annotations[key] == object.metadata.annotations[key])) &&
        object.metadata.annotations.all(key, key == 'jobset.sigs.k8s.io/epoch' || (key in oldObject.metadata.annotations && oldObject.metadata.annotations[key] == object.metadata.annotations[key]))
      )
     message: "ServiceAccount 'in-place-restart-sa' can only update the Pod annotation 'jobset.sigs.k8s.io/epoch'."
---
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicyBinding
metadata:
 name: update-only-epoch-pod-annotation-binding
spec:
 policyName: update-only-epoch-pod-annotation
 validationActions: [Deny]
```

Implementation details - Controller

The changes to the JobSet controller are responsible for orchestrating the group restarts of JobSet workloads when using in place restart. Basically, the JobSet controller will run the following reconciliation function.

```python
# Reconcile JobSet
def reconcile(jobSet):
    # Current code
    # ...
```

```
    # New code
    if isInPlaceRestartEnabled(jobSet):
        reconcileEpochs(jobSet)

# Check if in place restart is enabled
def isInPlaceRestartEnabled(jobSet):
    return jobSet.spec.failurePolicy.restartStrategy == "InPlaceRestart"

# Reconcile only in place restart fields
def reconcileEpochs(jobSet):
    childPods = listChildPods(jobSet.metadata.namespace, jobSet.metadata.name)
    epochs = extractEpochs(childPods)
    expectedEpochLength = countExpectedChildPods(jobSet)

    # Check if all worker Pods are at the same epoch
    # If so, make sure syncedEpoch is equal to this common value
    # (represented here by `generations[0]`)
    # This makes sure the Pod barriers are lifted
    if len(epochs) == expectedEpochsLength and allEqual(epochs):
        jobSet.status.syncedEpoch = epochs[0] # Idempotent

    # Otherwise, it means that the worker Pods are not in sync
    # If so, make sure deprecatedEpoch is equal to max(epochs) - 1
    # This makes sure all Pods that are not at the last epoch will be restarted in
place
    else:
        jobSet.status.deprecatedEpoch = max(epochs) - 1 # Idempotent

# Extract values of jobset.sigs.k8s.io/epoch annotations
def extractEpochs(pods):
    epochs = []
    for pod in pods:
        rawEpoch = pod.metadata.annotations["jobset.sigs.k8s.io/epoch"]
        epoch = int(rawEpoch)
        epochs.append(epoch)

    return epochs

# Count expected number of child Pods
def countExpectedChildPods(jobSet):
    count = 0
    for replicatedJob in jobSet.spec.replicatedJobs:
        jobTemplate = replicatedJob.template
        count += replicatedJob.replicas * jobTemplate.spec.parallelism
```

```
    return count
```

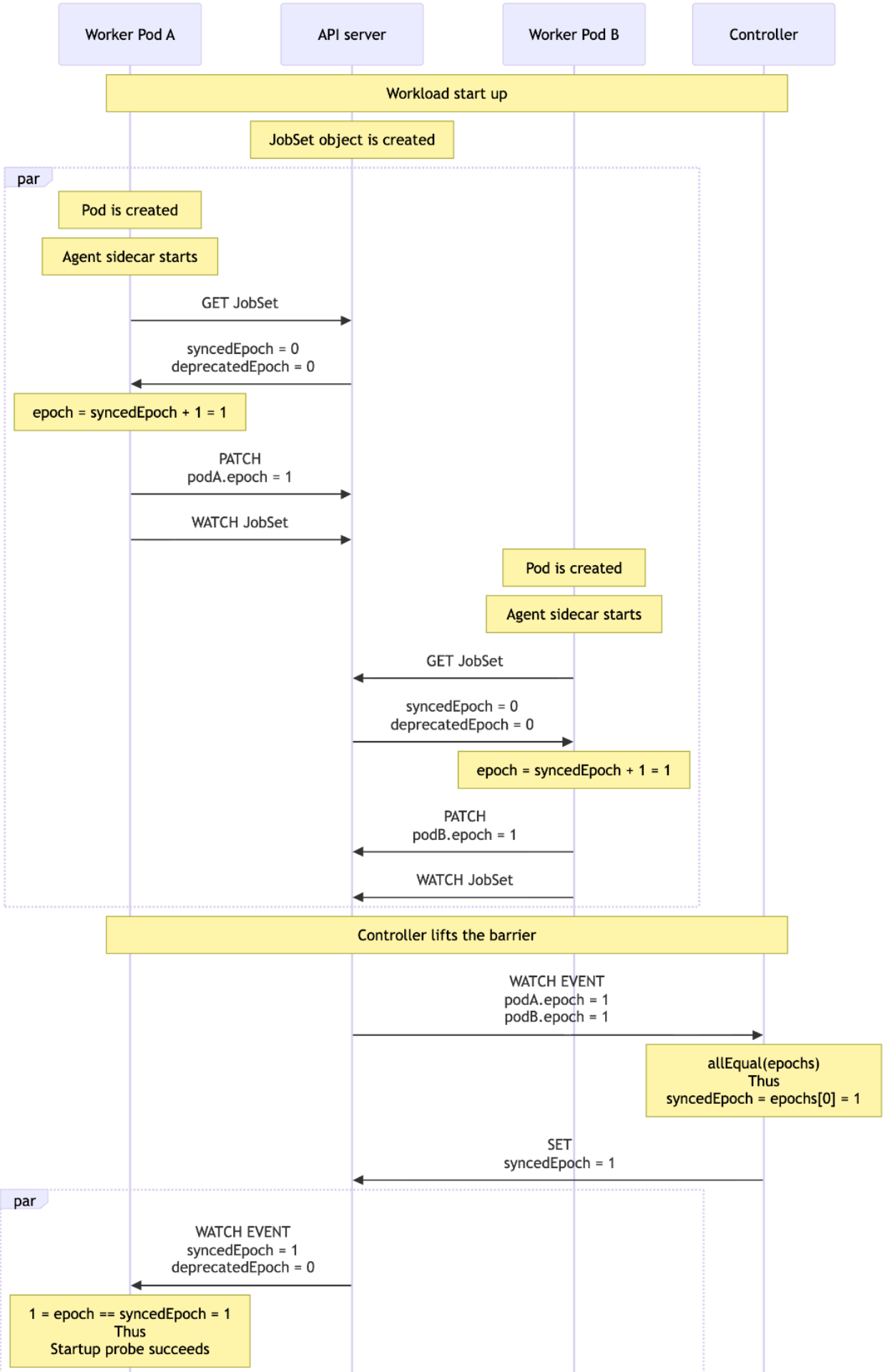
The highlights are:

- Only run in place restart logic for JobSet objects that have in place restart enabled (i.e., the field `jobSet.spec.failurePolicy.restartStrategy` is set to `InPlaceRestart`)
- If all child Pods exist and have the same epoch, it means they are in sync and should have their barriers lifted, so set `jobSet.status.syncedEpoch = epochs[0]` (equivalent to `jobSet.status.syncedEpoch += 1`). The agent sidecars will get this new synced epoch value and lift their barriers
- If the Pods are still not in sync (there is a mismatch in their epochs), make sure to deprecate all epochs that are not the most recent with `jobSet.status.deprecatedEpoch = max(epochs) - 1` (equivalent to `syncedEpoch`). This makes sure all agent sidecars that are not at the most recent epoch will restart in place to reach the new epoch

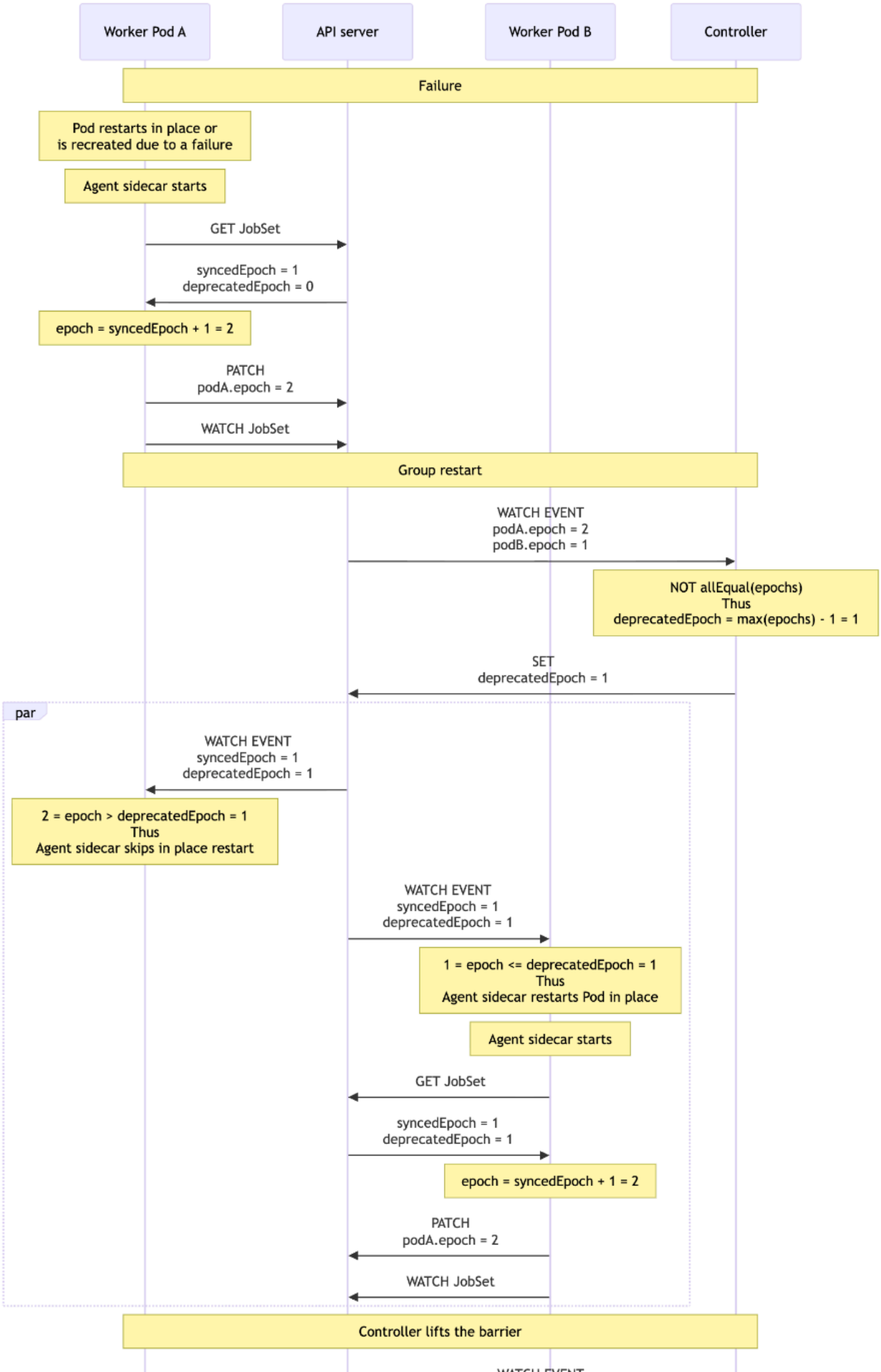
Besides the mentioned changes to the reconciliation loop, we also require to:

- Change the JobSet controller to watch child Pods for reconciliation
- Change the JobSet controller to index child Pods for efficient listing
- If the worker epochs ever exceed `jobset.spec.failurePolicy.maxRestarts`, fail the JobSet

Sequence diagram - Workload creation


Worker Pod A
API server
Worker Pod B
Controller
Workload start up
JobSet object is created
par
Pod is created
Agent sidecar starts
GET JobSet
syncedEpoch = 0
deprecatedEpoch = 0
epoch = syncedEpoch + 1 = 1
PATCH
podA.epoch = 1
WATCH JobSet
Pod is created
Agent sidecar starts
GET JobSet
syncedEpoch = 0
deprecatedEpoch = 0
epoch = syncedEpoch + 1 = 1
PATCH
podB.epoch = 1
WATCH JobSet
Controller lifts the barrier
WATCH EVENT
podA.epoch = 1
podB.epoch = 1
allEqual(epochs)
Thus
syncedEpoch = epochs[0] = 1
SET
syncedEpoch = 1
par
WATCH EVENT
syncedEpoch = 1
deprecatedEpoch = 0
1 = epoch == syncedEpoch = 1
Thus
Startup probe succeeds

Sequence diagram - Failure recovery


Worker Pod A
API server
Worker Pod B
Controller
Failure
Pod restarts in place or is recreated due to a failure
Agent sidecar starts
GET JobSet
syncedEpoch = 1
deprecatedEpoch = 0
epoch = syncedEpoch + 1 = 2
PATCH
podA.epoch = 2
WATCH JobSet
Group restart
WATCH EVENT
podA.epoch = 2
podB.epoch = 1
NOT allEqual(epochs)
Thus
deprecatedEpoch = max(epochs) - 1 = 1
SET
deprecatedEpoch = 1
par
WATCH EVENT
syncedEpoch = 1
deprecatedEpoch = 1
2 = epoch > deprecatedEpoch = 1
Thus
Agent sidecar skips in place restart
WATCH EVENT
syncedEpoch = 1
deprecatedEpoch = 1
1 = epoch <= deprecatedEpoch = 1
Thus
Agent sidecar restarts Pod in place
Agent sidecar starts
GET JobSet
syncedEpoch = 1
deprecatedEpoch = 1
epoch = syncedEpoch + 1 = 2
PATCH
podA.epoch = 2
WATCH JobSet
Controller lifts the barrier

Appendix

Benchmark

We benchmarked the current implementation of JobSet described in section Handling failures when training ML models against a custom version of JobSet that implements in place restart. Since `RestartPod` is not available yet, we created a prototype with `restartPolicy = OnFailure` and a single container that runs both the agent and worker programs. The benchmark was executed on a Kubernetes cluster with 5000 CPU Nodes (thus 5000 worker Pods) and a failure was simulated by forcing one of the containers to exit non-zero. The current implementation of JobSet took **2m10s** to fully restart (time between the failed container exiting non-zero and the last worker container starting again). The prototype of in place restart took **10s**.

Interestingly, the prototype time could be even lower. That's because, during a restart, roughly all 5000 agent containers start running at the same time and try to create a watch over the parent JobSet. Since each watch creation counts as a request and the number of Pods is larger than the API server limit, the KCP will answer with error code 429 to most requests, which triggers a backoff in the kubernetes clients in the agent containers. If the API server limit was higher, the in place restart time could be reduced to roughly **1s**. Even if the API limits are not raised, a future improvement to `restartPolicyRules` that allows skipping the restart of the agent sidecar during an in place restart (thus skipping the watch recreation) could also reduce the total restart time.

Future improvements

Speed up failure detection

In the proposed design, the JobSet controller can only detect that a failure occurred when the agent sidecar of the failed worker is started again and updates the epoch annotation. A best effort optimization can be done by triggering a group restart when any worker container of the most recent epoch terminates. This optimization saves the time between the worker container failing and its agent sidecar starting again.

Restart only specified containers to skip the overhead of recreating all watches

The proposed RestartPod action can only support restarting all containers, but a future expansion to the `restartPolicyRules` API can be made to allow only a few specified containers to be restarted. This allows for the agent sidecar to avoid being terminated in an on demand in place restart, which skips the overhead of recreating all watches.

Alternatives

Use the JobSet controller to directly restart the Pods in place instead of using the agent sidecar

Currently, there is no way to do that. The `RestartPod` action can only be triggered by a container exiting with a specified exit code. Alternatively, a new ideal upstream API could be created but the `RestartPod` action is the path of least resistance.

Use gRPC instead of the API server to establish communication between the JobSet controller and the agent sidecars

Using the API server with a declarative Pod annotation for the epoch is better practice from the Kubernetes API conventions.

Use the annotation `jobset.sigs.k8s.io/restart-attempt` instead of `jobset.sigs.k8s.io/epoch`

The annotation `jobset.sigs.k8s.io/restart-attempt` is technically written by the JobSet controller through the Pod template, so changing it with the agent sidecar would be an anti-pattern. Besides, the annotation `jobset.sigs.k8s.io/restart-attempt` in Pods is already heavily used for observability and changing its semantics would also be problematic.

Make the barrier optional

This is technically possible, but it adds little overhead while allowing the design to make less assumptions over the worker containers.

Inject the agent sidecar instead of having the user specify it in the JobSet manifest

This is technically possible, but the improvement on simplifying the API is far less than the surprise of injecting a powerful sidecar in the user Pod.

Restart only specified Pods instead of all of them

Currently JobSet only supports restarting the whole workload. Partial restart like restarting only a single Job or a group of ReplicatedJobs is desirable but it is still being discussed upstream.