Jobs

Jobs represent one-off tasks that run to completion and then stop.

A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created. Suspending a Job will delete its active Pods until the Job is resumed again.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot).

You can also use a Job to run multiple Pods in parallel.

If you want to run a Job (either a single task, or several in parallel) on a schedule, see CronJob.

Running an example Job

Here is an example Job config. It computes π to 2000 places and prints it out. It takes around 10s to complete.

controllers/job.yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34.0
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

You can run the example with this command:

```
kubectl apply -f https://kubernetes.io/examples/controllers/job.yaml
```

The output is similar to this:

```
job.batch/pi created
```

Check on the status of the Job with `kubectl`:

kubectl describe job pi | kubectl get job pi -o yaml

```
Name:           pi
Namespace:      default
Selector:       batch.kubernetes.io/controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c
Labels:         batch.kubernetes.io/controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c
                batch.kubernetes.io/job-name=pi
                ...
Annotations:    batch.kubernetes.io/job-tracking: ""
Parallelism:    1
Completions:    1
Start Time:     Mon, 02 Dec 2019 15:20:11 +0200
Completed At:   Mon, 02 Dec 2019 15:21:16 +0200
Duration:       65s
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  batch.kubernetes.io/controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c
           batch.kubernetes.io/job-name=pi
  Containers:
   pi:
    Image:      perl:5.34.0
    Port:       <none>
    Host Port:  <none>
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  21s   job-controller  Created pod: pi-xf9p4
  Normal  Completed         18s   job-controller  Job completed
```

To view completed Pods of a Job, use `kubectl get pods`.

To list all the Pods that belong to a Job in a machine readable form, you can use a command like this:

```
pods=$(kubectl get pods --selector=batch.kubernetes.io/job-name=pi --output=jsonpath='{.items[*].metadata.name}')
echo $pods
```

The output is similar to this:

```
pi-5rwd7
```

Here, the selector is the same as the selector for the Job. The `--output=jsonpath` option specifies an expression with the name from each Pod in the returned list.

View the standard output of one of the pods:

```
kubectl logs $pods
```

Another way to view the logs of a Job:

```
kubectl logs jobs/pi
```

The output is similar to this:

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328
```

Writing a Job spec

As with all other Kubernetes config, a Job needs `apiVersion`, `kind`, and `metadata` fields.

When the control plane creates new Pods for a Job, the `.metadata.name` of the Job is part of the basis for naming those Pods. The name of a Job must be a valid DNS subdomain value, but this can produce unexpected results for the Pod hostnames. For best compatibility, the name should follow the more restrictive rules for a DNS label. Even when the name is a DNS subdomain, the name must be no longer than 63 characters.

A Job also needs a `.spec` section.

Job Labels

Job labels will have `batch.kubernetes.io/` prefix for `job-name` and `controller-uid`.

Pod Template

The `.spec.template` is the only required field of the `.spec`.

The `.spec.template` is a pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a pod template in a Job must specify appropriate labels (see pod selector) and an appropriate restart policy.

Only a `RestartPolicy` equal to `Never` or `OnFailure` is allowed.

Pod selector

The `.spec.selector` field is optional. In almost all cases you should not specify it. See section specifying your own pod selector.

Parallel execution for Jobs

There are three main types of task suitable to run as a Job:

1. Non-parallel Jobs
   - normally, only one Pod is started, unless the Pod fails.
   - the Job is complete as soon as its Pod terminates successfully.
2. Parallel Jobs with a *fixed completion count*:
   - specify a non-zero positive value for `.spec.completions`.
   - the Job represents the overall task, and is complete when there are `.spec.completions` successful Pods.
   - when using `.spec.completionMode="Indexed"`, each Pod gets a different index in the range 0 to `.spec.completions-1`.
3. Parallel Jobs with a *work queue*:
   - do not specify `.spec.completions`, default to `.spec.parallelism`.
   - the Pods must coordinate amongst themselves or an external service to determine what each should work on. For example, a Pod might fetch a batch of up to N items from the work queue.
   - each Pod is independently capable of determining whether or not all its peers are done, and thus that the entire Job is done.
   - when *any* Pod from the Job terminates with success, no new Pods are created.
   - once at least one Pod has terminated with success and all Pods are terminated, then the Job is completed with success.
   - once any Pod has exited with success, no other Pod should still be doing any work for this task or writing any output. They should all be in the process of exiting.

For a *non-parallel* Job, you can leave both `.spec.completions` and `.spec.parallelism` unset. When both are unset, both are defaulted to 1.

For a *fixed completion count* Job, you should set `.spec.completions` to the number of completions needed. You can set `.spec.parallelism`, or leave it unset and it will default to 1.

For a *work queue* Job, you must leave `.spec.completions` unset, and set `.spec.parallelism` to a non-negative integer.

For more information about how to make use of the different types of job, see the job patterns section.

Controlling parallelism

The requested parallelism (`.spec.parallelism`) can be set to any non-negative value. If it is unspecified, it defaults to 1. If it is specified as 0, then the Job is effectively paused until it is increased.

Actual parallelism (number of pods running at any instant) may be more or less than requested parallelism, for a variety of reasons:

- For *fixed completion count* Jobs, the actual number of pods running in parallel will not exceed the number of remaining completions. Higher values of `.spec.parallelism` are effectively ignored.
- For *work queue* Jobs, no new Pods are started after any Pod has succeeded -- remaining Pods are allowed to complete, however.
- If the Job Controller has not had time to react.
- If the Job controller failed to create Pods for any reason (lack of `ResourceQuota`, lack of permission, etc.), then there may be fewer pods than requested.
- The Job controller may throttle new Pod creation due to excessive previous pod failures in the same Job.
- When a Pod is gracefully shut down, it takes time to stop.

Completion mode

FEATURE STATE: `Kubernetes v1.24 [stable]`

Jobs with *fixed completion count* - that is, jobs that have non null `.spec.completions` - can have a completion mode that is specified in `.spec.completionMode`:

- `NonIndexed` (default): the Job is considered complete when there have been `.spec.completions` successfully completed Pods. In other words, each Pod completion is homologous to each other. Note that Jobs that have null `.spec.completions` are implicitly `NonIndexed`.
- `Indexed`: the Pods of a Job get an associated completion index from 0 to `.spec.completions-1`. The index is available through four mechanisms:
  - The Pod annotation `batch.kubernetes.io/job-completion-index`.
  - The Pod label `batch.kubernetes.io/job-completion-index` (for v1.28 and later). Note the feature gate `PodIndexLabel` must be enabled to use this label, and it is enabled by default.
  - As part of the Pod hostname, following the pattern `$(job-name)-$(index)`. When you use an Indexed Job in combination with a Service, Pods within the Job can use the deterministic hostnames to address each other via DNS. For more information about how to configure this, see Job with Pod-to-Pod Communication.
  - From the containerized task, in the environment variable `JOB_COMPLETION_INDEX`.

  The Job is considered complete when there is one successfully completed Pod for each index. For more information about how to use this mode, see Indexed Job for Parallel Processing with Static Work Assignment.

> **Note:** Although rare, more than one Pod could be started for the same index (due to various reasons such as node failures, kubelet restarts, or Pod evictions). In this case, only the first Pod that completes successfully will count towards the completion count and update the status of the Job. The other Pods that are running or completed for the same index will be deleted by the Job controller once they are detected.

Handling Pod and container failures

A container in a Pod may fail for a number of reasons, such as because the process in it exited with a non-zero exit code, or the container was killed for exceeding a memory limit, etc. If this happens, and the `.spec.template.spec.restartPolicy = "OnFailure"`, then the Pod stays on the node, but the container is re-run. Therefore, your program needs to handle the case when it is restarted locally, or else specify `.spec.template.spec.restartPolicy = "Never"`. See pod lifecycle for more information on `restartPolicy`.

An entire Pod can also fail, for a number of reasons, such as when the pod is kicked off the node (node is upgraded, rebooted, deleted, etc.), or if a container of the Pod fails and the `.spec.template.spec.restartPolicy = "Never"`. When a Pod fails, then the Job controller starts a new Pod. This means that your application needs to handle the case when it is restarted in a new pod. In particular, it needs to handle temporary files, locks, incomplete output and the like caused by previous runs.

By default, each pod failure is counted towards the `.spec.backoffLimit` limit, see pod backoff failure policy. However, you can customize handling of pod failures by setting the Job's pod failure policy.

Additionally, you can choose to count the pod failures independently for each index of an Indexed Job by setting the `.spec.backoffLimitPerIndex` field (for more information, see backoff limit per index).

Note that even if you specify `.spec.parallelism = 1` and `.spec.completions = 1` and `.spec.template.spec.restartPolicy = "Never"`, the same program may sometimes be started twice.

If you do specify `.spec.parallelism` and `.spec.completions` both greater than 1, then there may be multiple pods running at once. Therefore, your pods must also be tolerant of concurrency.

When the feature gates `PodDisruptionConditions` and `JobPodFailurePolicy` are both enabled, and the `.spec.podFailurePolicy` field is set, the Job controller does not consider a terminating Pod (a pod that has a `.metadata.deletionTimestamp` field set) as a failure until that Pod is terminal (its `.status.phase` is `Failed` or `Succeeded`). However, the Job controller creates a replacement Pod as soon as the termination becomes apparent. Once the pod terminates, the Job controller evaluates `.backoffLimit` and `.podFailurePolicy` for the relevant Job, taking this now-terminated Pod into consideration.

If either of these requirements is not satisfied, the Job controller counts a terminating Pod as an immediate failure, even if that Pod later terminates with `phase: "Succeeded"`.

Pod backoff failure policy

There are situations where you want to fail a Job after some amount of retries due to a logical error in configuration etc. To do so, set `.spec.backoffLimit` to specify the number of retries before considering a Job as failed. The back-off limit is set by default to 6. Failed Pods associated with the Job are recreated by the Job controller with an exponential back-off delay (10s, 20s, 40s ...) capped at six minutes.

The number of retries is calculated in two ways:

- The number of Pods with `.status.phase = "Failed"`.
- When using `restartPolicy = "OnFailure"`, the number of retries in all the containers of Pods with `.status.phase` equal to `Pending` or `Running`.

If either of the calculations reaches the `.spec.backoffLimit`, the Job is considered failed.

> **Note:** If your job has `restartPolicy = "OnFailure"`, keep in mind that your Pod running the Job will be terminated once the job backoff limit has been reached. This can make debugging the Job's executable more difficult. We suggest setting `restartPolicy = "Never"` when debugging the Job or using a logging system to ensure output from failed Jobs is not lost inadvertently.

Backoff limit per index

FEATURE STATE: `Kubernetes v1.29 [beta]`

> **Note:** You can only configure the backoff limit per index for an Indexed Job, if you have the `JobBackoffLimitPerIndex` feature gate enabled in your cluster.

When you run an indexed Job, you can choose to handle retries for pod failures independently for each index. To do so, set the `.spec.backoffLimitPerIndex` to specify the maximal number of pod failures per index.

When the per-index backoff limit is exceeded for an index, Kubernetes considers the index as failed and adds it to the `.status.failedIndexes` field. The succeeded indexes, those with a successfully executed pods, are recorded in the `.status.completedIndexes` field, regardless of whether you set the `backoffLimitPerIndex` field.

Note that a failing index does not interrupt execution of other indexes. Once all indexes finish for a Job where you specified a backoff limit per index, if at least one of those indexes did fail, the Job controller marks the overall Job as failed, by setting the Failed condition in the status. The Job gets marked as failed even if some, potentially nearly all, of the indexes were processed successfully.

You can additionally limit the maximal number of indexes marked failed by setting the `.spec.maxFailedIndexes` field. When the number of failed indexes exceeds the `maxFailedIndexes` field, the Job controller triggers termination of all remaining running Pods for that Job. Once all pods are terminated, the entire Job is marked failed by the Job controller, by setting the Failed condition in the Job status.

Here is an example manifest for a Job that defines a `backoffLimitPerIndex`:

/controllers/job-backoff-limit-per-index-example.yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: job-backoff-limit-per-index-example
spec:
  completions: 10
  parallelism: 3
  completionMode: Indexed  # required for the feature
  backoffLimitPerIndex: 1  # maximal number of failures per index
  maxFailedIndexes: 5      # maximal number of failed indexes before terminating the Job execution
  template:
    spec:
      restartPolicy: Never # required for the feature
      containers:
      - name: example
        image: python
        command:           # The jobs fails as there is at least one failed index
                           # (all even indexes fail in here), yet all indexes
                           # are executed as maxFailedIndexes is not exceeded.
        - python3
        - -c
        - |
          import os, sys
          print("Hello world")
          if int(os.environ.get("JOB_COMPLETION_INDEX")) % 2 == 0:
            sys.exit(1)
```

In the example above, the Job controller allows for one restart for each of the indexes. When the total number of failed indexes exceeds 5, then the entire Job is terminated.

Once the job is finished, the Job status looks as follows:

```
kubectl get -o yaml job job-backoff-limit-per-index-example
```

```yaml
  status:
    completedIndexes: 1,3,5,7,9
    failedIndexes: 0,2,4,6,8
    succeeded: 5          # 1 succeeded pod for each of 5 succeeded indexes
    failed: 10            # 2 failed pods (1 retry) for each of 5 failed indexes
    conditions:
    - message: Job has failed indexes
      reason: FailedIndexes
      status: "True"
      type: Failed
```

Additionally, you may want to use the per-index backoff along with a pod failure policy. When using per-index backoff, there is a new `FailIndex` action available which allows you to avoid unnecessary retries within an index.

Pod failure policy

FEATURE STATE: `Kubernetes v1.26 [beta]`

> **Note:** You can only configure a Pod failure policy for a Job if you have the `JobPodFailurePolicy` feature gate enabled in your cluster. Additionally, it is recommended to enable the `PodDisruptionConditions` feature gate in order to be able to detect and handle Pod disruption conditions in the Pod failure policy (see also: Pod disruption conditions). Both feature gates are available in Kubernetes 1.29.

A Pod failure policy, defined with the `.spec.podFailurePolicy` field, enables your cluster to handle Pod failures based on the container exit codes and the Pod conditions.

In some situations, you may want to have a better control when handling Pod failures than the control provided by the Pod backoff failure policy, which is based on the Job's `.spec.backoffLimit`. These are some examples of use cases:

- To optimize costs of running workloads by avoiding unnecessary Pod restarts, you can terminate a Job as soon as one of its Pods fails with an exit code indicating a software bug.
- To guarantee that your Job finishes even if there are disruptions, you can ignore Pod failures caused by disruptions (such as preemption, API-initiated eviction or taint-based eviction) so that they don't count towards the `.spec.backoffLimit` limit of retries.

You can configure a Pod failure policy, in the `.spec.podFailurePolicy` field, to meet the above use cases. This policy can handle Pod failures based on the container exit codes and the Pod conditions.

Here is a manifest for a Job that defines a `podFailurePolicy`:

/controllers/job-pod-failure-policy-example.yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: job-pod-failure-policy-example
spec:
  completions: 12
  parallelism: 3
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: main
        image: docker.io/library/bash:5
        command: ["bash"]       # example command simulating a bug which triggers the FailJob action
        args:
        - -c
        - echo "Hello world!" && sleep 5 && exit 42
  backoffLimit: 6
  podFailurePolicy:
    rules:
    - action: FailJob
      onExitCodes:
        containerName: main     # optional
        operator: In            # one of: In, NotIn
        values: [42]
    - action: Ignore            # one of: Ignore, FailJob, Count
      onPodConditions:
      - type: DisruptionTarget  # indicates Pod disruption
```

In the example above, the first rule of the Pod failure policy specifies that the Job should be marked failed if the `main` container fails with the 42 exit code. The following are the rules for the `main` container specifically:

- an exit code of 0 means that the container succeeded

- an exit code of 42 means that the **entire Job** failed
- any other exit code represents that the container failed, and hence the entire Pod. The Pod will be re-created if the total number of restarts is below `backoffLimit`. If the `backoffLimit` is reached the **entire Job** failed.

> **Note:** Because the Pod template specifies a `restartPolicy: Never`, the kubelet does not restart the `main` container in that particular Pod.

The second rule of the Pod failure policy, specifying the `Ignore` action for failed Pods with condition `DisruptionTarget` excludes Pod disruptions from being counted towards the `.spec.backoffLimit` limit of retries.

> **Note:** If the Job failed, either by the Pod failure policy or Pod backoff failure policy, and the Job is running multiple Pods, Kubernetes terminates all the Pods in that Job that are still Pending or Running.

These are some requirements and semantics of the API:

- if you want to use a `.spec.podFailurePolicy` field for a Job, you must also define that Job's pod template with `.spec.restartPolicy` set to `Never`.
- the Pod failure policy rules you specify under `spec.podFailurePolicy.rules` are evaluated in order. Once a rule matches a Pod failure, the remaining rules are ignored. When no rule matches the Pod failure, the default handling applies.
- you may want to restrict a rule to a specific container by specifying its name in `spec.podFailurePolicy.rules[*].onExitCodes.containerName`. When not specified the rule applies to all containers. When specified, it should match one the container or `initContainer` names in the Pod template.
- you may specify the action taken when a Pod failure policy is matched by `spec.podFailurePolicy.rules[*].action`. Possible values are:
  - `FailJob`: use to indicate that the Pod's job should be marked as Failed and all running Pods should be terminated.
  - `Ignore`: use to indicate that the counter towards the `.spec.backoffLimit` should not be incremented and a replacement Pod should be created.
  - `Count`: use to indicate that the Pod should be handled in the default way. The counter towards the `.spec.backoffLimit` should be incremented.
  - `FailIndex`: use this action along with backoff limit per index to avoid unnecessary retries within the index of a failed pod.

> **Note:** When you use a `podFailurePolicy`, the job controller only matches Pods in the `Failed` phase. Pods with a deletion timestamp that are not in a terminal phase (`Failed` or `Succeeded`) are considered still terminating. This implies that terminating pods retain a tracking finalizer until they reach a terminal phase. Since Kubernetes 1.27, Kubelet transitions deleted pods to a terminal phase (see: Pod Phase). This ensures that deleted pods have their finalizers removed by the Job controller.

> **Note:** Starting with Kubernetes v1.28, when Pod failure policy is used, the Job controller recreates terminating Pods only once these Pods reach the terminal `Failed` phase. This behavior is similar to `podReplacementPolicy: Failed`. For more information, see Pod replacement policy.

Job termination and cleanup

When a Job completes, no more Pods are created, but the Pods are usually not deleted either. Keeping them around allows you to still view the logs of completed pods to check for errors, warnings, or other diagnostic output. The job object also remains after it is completed so that you can view its status. It is up to the user to delete old jobs after noting their status. Delete the job with `kubectl` (e.g. `kubectl delete jobs/pi` or `kubectl delete -f ./job.yaml`). When you delete the job using `kubectl`, all the pods it created are deleted too.

By default, a Job will run uninterrupted unless a Pod fails (`restartPolicy=Never`) or a Container exits in error (`restartPolicy=OnFailure`), at which point the Job defers to the `.spec.backoffLimit` described above. Once `.spec.backoffLimit` has been reached the Job will be marked as failed and any running Pods will be terminated.

Another way to terminate a Job is by setting an active deadline. Do this by setting the `.spec.activeDeadlineSeconds` field of the Job to a number of seconds. The `activeDeadlineSeconds` applies to the duration of the job, no matter how many Pods are created. Once a Job reaches `activeDeadlineSeconds`, all of its running Pods are terminated and the Job status will become `type: Failed` with `reason: DeadlineExceeded`.

Note that a Job's `.spec.activeDeadlineSeconds` takes precedence over its `.spec.backoffLimit`. Therefore, a Job that is retrying one or more failed Pods will not deploy additional Pods once it reaches the time limit specified by `activeDeadlineSeconds`, even if the `backoffLimit` is not yet reached.

Example:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-timeout
spec:
  backoffLimit: 5
  activeDeadlineSeconds: 100
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34.0
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

Note that both the Job spec and the Pod template spec within the Job have an `activeDeadlineSeconds` field. Ensure that you set this field at the proper level.

Keep in mind that the `restartPolicy` applies to the Pod, and not to the Job itself: there is no automatic Job restart once the Job status is `type: Failed`. That is, the Job termination mechanisms activated with `.spec.activeDeadlineSeconds` and `.spec.backoffLimit` result in a permanent Job failure that requires manual intervention to resolve.

Clean up finished jobs automatically

Finished Jobs are usually no longer needed in the system. Keeping them around in the system will put pressure on the API server. If the Jobs are managed directly by a higher level controller, such as CronJobs, the Jobs can be cleaned up by CronJobs based on the specified capacity-based cleanup policy.

TTL mechanism for finished Jobs

FEATURE STATE: `Kubernetes v1.23 [stable]`

Another way to clean up finished Jobs (either `Complete` or `Failed`) automatically is to use a TTL mechanism provided by a TTL controller for finished resources, by specifying the `.spec.ttlSecondsAfterFinished` field of the Job.

When the TTL controller cleans up the Job, it will delete the Job cascadingly, i.e. delete its dependent objects, such as Pods, together with the Job. Note that when the Job is deleted, its lifecycle guarantees, such as finalizers, will be honored.

For example:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-ttl
spec:
  ttlSecondsAfterFinished: 100
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34.0
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

The Job `pi-with-ttl` will be eligible to be automatically deleted, `100` seconds after it finishes.

If the field is set to `0`, the Job will be eligible to be automatically deleted immediately after it finishes. If the field is unset, this Job won't be cleaned up by the TTL controller after it finishes.

> **Note:**
> It is recommended to set `ttlSecondsAfterFinished` field because unmanaged jobs (Jobs that you created directly, and not indirectly through other workload APIs such as CronJob) have a default deletion policy of `orphanDependents` causing Pods created by an unmanaged Job to be left around after that Job is fully deleted. Even though the control plane eventually garbage collects the Pods from a deleted Job after they either fail or complete, sometimes those lingering pods may cause cluster performance degradation or in worst case cause the cluster to go offline due to this degradation.
>
> You can use LimitRanges and ResourceQuotas to place a cap on the amount of resources that a particular namespace can consume.

Job patterns

The Job object can be used to process a set of independent but related *work items*. These might be emails to be sent, frames to be rendered, files to be transcoded, ranges of keys in a NoSQL database to scan, and so on.

In a complex system, there may be multiple different sets of work items. Here we are just considering one set of work items that the user wants to manage together — a *batch job*.

There are several different patterns for parallel computation, each with strengths and weaknesses. The tradeoffs are:

- One Job object for each work item, versus a single Job object for all work items. One Job per work item creates some overhead for the user and for the system to manage large numbers of Job objects. A single Job for all work items is better for large numbers of items.
- Number of Pods created equals number of work items, versus each Pod can process multiple work items. When the number of Pods equals the number of work items, the Pods typically requires less modification to existing code and containers. Having each Pod process multiple work items is better for large numbers of items.
- Several approaches use a work queue. This requires running a queue service, and modifications to the existing program or container to make it use the work queue. Other approaches are easier to adapt to an existing containerised application.
- When the Job is associated with a headless Service, you can enable the Pods within a Job to communicate with each other to collaborate in a computation.

The tradeoffs are summarized here, with columns 2 to 4 corresponding to the above tradeoffs. The pattern names are also links to examples and more detailed description.

Pattern	Single Job object	Fewer pods than work items?	Use app unmodified?
Queue with Pod Per Work Item	✓		sometimes
Queue with Variable Pod Count	✓	✓	
Indexed Job with Static Work Assignment	✓		✓
Job with Pod-to-Pod Communication	✓	sometimes	sometimes
Job Template Expansion			✓

When you specify completions with `.spec.completions`, each Pod created by the Job controller has an identical `spec`. This means that all pods for a task will have the same command line and the same image, the same volumes, and (almost) the same environment variables. These patterns are different ways to arrange for pods to work on different things.

This table shows the required settings for `.spec.parallelism` and `.spec.completions` for each of the patterns. Here, `W` is the number of work items.

Pattern	`.spec.completions`	`.spec.parallelism`
Queue with Pod Per Work Item	W	any

Pattern	`.spec.completions`	`.spec.parallelism`
Queue with Variable Pod Count	null	any
Indexed Job with Static Work Assignment	W	any
Job with Pod-to-Pod Communication	W	W
Job Template Expansion	1	should be 1

Advanced usage

Suspending a Job

FEATURE STATE: `Kubernetes v1.24 [stable]`

When a Job is created, the Job controller will immediately begin creating Pods to satisfy the Job's requirements and will continue to do so until the Job is complete. However, you may want to temporarily suspend a Job's execution and resume it later, or start Jobs in suspended state and have a custom controller decide later when to start them.

To suspend a Job, you can update the `.spec.suspend` field of the Job to true; later, when you want to resume it again, update it to false. Creating a Job with `.spec.suspend` set to true will create it in the suspended state.

When a Job is resumed from suspension, its `.status.startTime` field will be reset to the current time. This means that the `.spec.activeDeadlineSeconds` timer will be stopped and reset when a Job is suspended and resumed.

When you suspend a Job, any running Pods that don't have a status of `Completed` will be terminated. with a SIGTERM signal. The Pod's graceful termination period will be honored and your Pod must handle this signal in this period. This may involve saving progress for later or undoing changes. Pods terminated this way will not count towards the Job's `completions` count.

An example Job definition in the suspended state can be like so:

```
kubectl get job myjob -o yaml
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: myjob
spec:
  suspend: true
  parallelism: 1
  completions: 5
  template:
    spec:
      ...
```

You can also toggle Job suspension by patching the Job using the command line.

Suspend an active Job:

```
kubectl patch job/myjob --type=strategic --patch '{"spec":{"suspend":true}}'
```

Resume a suspended Job:

```
kubectl patch job/myjob --type=strategic --patch '{"spec":{"suspend":false}}'
```

The Job's status can be used to determine if a Job is suspended or has been suspended in the past:

```
kubectl get jobs/myjob -o yaml
```

```yaml
apiVersion: batch/v1
kind: Job
# .metadata and .spec omitted
status:
  conditions:
  - lastProbeTime: "2021-02-05T13:14:33Z"
    lastTransitionTime: "2021-02-05T13:14:33Z"
    status: "True"
    type: Suspended
  startTime: "2021-02-05T13:13:48Z"
```

The Job condition of type "Suspended" with status "True" means the Job is suspended; the `lastTransitionTime` field can be used to determine how long the Job has been suspended for. If the status of that condition is "False", then the Job was previously suspended and is now running. If such a condition does not exist in the Job's status, the Job has never been stopped.

Events are also created when the Job is suspended and resumed:

```
kubectl describe jobs/myjob
```

```
Name:           myjob
...
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  12m   job-controller  Created pod: myjob-hlrpl
  Normal  SuccessfulDelete  11m   job-controller  Deleted pod: myjob-hlrpl
  Normal  Suspended         11m   job-controller  Job suspended
  Normal  SuccessfulCreate  3s    job-controller  Created pod: myjob-jvb44
  Normal  Resumed           3s    job-controller  Job resumed
```

The last four events, particularly the "Suspended" and "Resumed" events, are directly a result of toggling the `.spec.suspend` field. In the time between these two events, we see that no Pods were created, but Pod creation restarted as soon as the Job was resumed.

Mutable Scheduling Directives

FEATURE STATE: `Kubernetes v1.27 [stable]`

In most cases, a parallel job will want the pods to run with constraints, like all in the same zone, or all either on GPU model x or y but not a mix of both.

The suspend field is the first step towards achieving those semantics. Suspend allows a custom queue controller to decide when a job should start; However, once a job is unsuspended, a custom queue controller has no influence on where the pods of a job will actually land.

This feature allows updating a Job's scheduling directives before it starts, which gives custom queue controllers the ability to influence pod placement while at the same time offloading actual pod-to-node assignment to kube-scheduler. This is allowed only for suspended Jobs that have never been unsuspended before.

The fields in a Job's pod template that can be updated are node affinity, node selector, tolerations, labels, annotations and scheduling gates.

Specifying your own Pod selector

Normally, when you create a Job object, you do not specify `.spec.selector`. The system defaulting logic adds this field when the Job is created. It picks a selector value that will not overlap with any other jobs.

However, in some cases, you might need to override this automatically set selector. To do this, you can specify the `.spec.selector` of the Job.

Be very careful when doing this. If you specify a label selector which is not unique to the pods of that Job, and which matches unrelated Pods, then pods of the unrelated job may be deleted, or this Job may count other Pods as completing it, or one or both Jobs may refuse to create Pods or run to completion. If a non-unique selector is chosen, then other controllers (e.g. ReplicationController) and their Pods may behave in unpredictable ways too. Kubernetes will not stop you from making a mistake when specifying `.spec.selector`.

Here is an example of a case when you might want to use this feature.

Say Job `old` is already running. You want existing Pods to keep running, but you want the rest of the Pods it creates to use a different pod template and for the Job to have a new name. You cannot update the Job because these fields are not updatable. Therefore, you delete Job `old` but *leave its pods running*, using `kubectl delete jobs/old --cascade=orphan`. Before deleting it, you make a note of what selector it uses:

```
kubectl get job old -o yaml
```

The output is similar to this:

```
kind: Job
metadata:
  name: old
  ...
spec:
  selector:
    matchLabels:
      batch.kubernetes.io/controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002
  ...
```

Then you create a new Job with name `new` and you explicitly specify the same selector. Since the existing Pods have label `batch.kubernetes.io/controller-uid=a8f3d00d-c6d2-11e5-9f87-42010af00002`, they are controlled by Job `new` as well.

You need to specify `manualSelector: true` in the new Job since you are not using the selector that the system normally generates for you automatically.

```
kind: Job
metadata:
  name: new
  ...
spec:
  manualSelector: true
  selector:
    matchLabels:
      batch.kubernetes.io/controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002
  ...
```

The new Job itself will have a different uid from `a8f3d00d-c6d2-11e5-9f87-42010af00002`. Setting `manualSelector: true` tells the system that you know what you are doing and to allow this mismatch.

Job tracking with finalizers

FEATURE STATE: `Kubernetes v1.26 [stable]`

The control plane keeps track of the Pods that belong to any Job and notices if any such Pod is removed from the API server. To do that, the Job controller creates Pods with the finalizer `batch.kubernetes.io/job-tracking`. The controller removes the finalizer only after the Pod has been accounted for in the Job status, allowing the Pod to be removed by other controllers or users.

> **Note:** See My pod stays terminating if you observe that pods from a Job are stuck with the tracking finalizer.

Elastic Indexed Jobs

FEATURE STATE: `Kubernetes v1.27 [beta]`

You can scale Indexed Jobs up or down by mutating both `.spec.parallelism` and `.spec.completions` together such that `.spec.parallelism == .spec.completions`. When the `ElasticIndexedJob` feature gate on the API server is disabled, `.spec.completions` is immutable.

Use cases for elastic Indexed Jobs include batch workloads which require scaling an indexed Job, such as MPI, Horovord, Ray, and PyTorch training jobs.

Delayed creation of replacement pods

FEATURE STATE: `Kubernetes v1.29 [beta]`

> **Note:** You can only set `podReplacementPolicy` on Jobs if you enable the `JobPodReplacementPolicy` feature gate (enabled by default).

By default, the Job controller recreates Pods as soon they either fail or are terminating (have a deletion timestamp). This means that, at a given time, when some of the Pods are terminating, the number of running Pods for a Job can be greater than `parallelism` or greater than one Pod per index (if you are using an Indexed Job).

You may choose to create replacement Pods only when the terminating Pod is fully terminal (has `status.phase: Failed`). To do this, set the `.spec.podReplacementPolicy: Failed`. The default replacement policy depends on whether the Job has a `podFailurePolicy` set. With no Pod failure policy defined for a Job, omitting the `podReplacementPolicy` field selects the `TerminatingOrFailed` replacement policy: the control plane creates replacement Pods immediately upon Pod deletion (as soon as the control plane sees that a Pod for this Job has `deletionTimestamp` set). For Jobs with a Pod failure policy set, the default `podReplacementPolicy` is `Failed`, and no other value is permitted. See Pod failure policy to learn more about Pod failure policies for Jobs.

```yaml
kind: Job
metadata:
  name: new
  ...
spec:
  podReplacementPolicy: Failed
  ...
```

Provided your cluster has the feature gate enabled, you can inspect the `.status.terminating` field of a Job. The value of the field is the number of Pods owned by the Job that are currently terminating.

```
kubectl get jobs/myjob -o yaml
```

```yaml
apiVersion: batch/v1
kind: Job
# .metadata and .spec omitted
status:
  terminating: 3 # three Pods are terminating and have not yet reached the Failed phase
```

Alternatives

Bare Pods

When the node that a Pod is running on reboots or fails, the pod is terminated and will not be restarted. However, a Job will create new Pods to replace terminated ones. For this reason, we recommend that you use a Job rather than a bare Pod, even if your application requires only a single Pod.

Replication Controller

Jobs are complementary to Replication Controllers. A Replication Controller manages Pods which are not expected to terminate (e.g. web servers), and a Job manages Pods that are expected to terminate (e.g. batch tasks).

As discussed in Pod Lifecycle, `Job` is *only* appropriate for pods with `RestartPolicy` equal to `OnFailure` or `Never`. (Note: If `RestartPolicy` is not set, the default value is `Always`.)

Single Job starts controller Pod

Another pattern is for a single Job to create a Pod which then creates other Pods, acting as a sort of custom controller for those Pods. This allows the most flexibility, but may be somewhat complicated to get started with and offers less integration with Kubernetes.

One example of this pattern would be a Job which starts a Pod which runs a script that in turn starts a Spark master controller (see spark example), runs a spark driver, and then cleans up.

An advantage of this approach is that the overall process gets the completion guarantee of a Job object, but maintains complete control over what Pods are created and how work is assigned to them.

What's next

- Learn about Pods.
- Read about different ways of running Jobs:
  - Coarse Parallel Processing Using a Work Queue
  - Fine Parallel Processing Using a Work Queue
  - Use an indexed Job for parallel processing with static work assignment
  - Create multiple Jobs based on a template: Parallel Processing using Expansions
- Follow the links within Clean up finished jobs automatically to learn more about how your cluster can clean up completed and / or failed tasks.
- `Job` is part of the Kubernetes REST API. Read the Job object definition to understand the API for jobs.
- Read about `CronJob`, which you can use to define a series of Jobs that will run based on a schedule, similar to the UNIX tool `cron`.
- Practice how to configure handling of retriable and non-retriable pod failures using `podFailurePolicy`, based on the step-by-step examples.

Feedback

Was this page helpful?

Yes No

Last modified December 24, 2023 at 9:00 PM PST: fix typos (d536e46dbd)