[PUBLIC] KEP- 5233: Node Readiness Gates

KEP: https://github.com/kubernetes/enhancements/issues/5233
K8s Issue: https://github.com/kubernetes/kubernetes/issues/131208

Status: Based on recent feedback, this KEP is revising its approach to explore a simpler design with less operational risks. The focus is now on using node-local 'probing mechanisms' to verify readiness, rather than relying on multiple external agents with broader permissions to patch Node objects.

We will narrow the scope to well-known readiness requirements that can be defined at node provisioning time. Consequently, a decision on adding a `readinessGates` field to the `NodeSpec` is deferred until the use-cases can be justified. This KEP will also avoid introducing a new CRD at this time, to reduce the risk of global misconfiguration.

Release Signoff Checklist

Items marked with (R) are required *prior to targeting to a milestone / release*.

- [ ] (R) Enhancement issue in release milestone, which links to KEP dir in kubernetes/enhancements (not the initial KEP PR)
- [ ] (R) KEP approvers have approved the KEP status as `implementable`
- [ ] (R) Design details are appropriately documented
- [ ] (R) Test plan is in place, giving consideration to SIG Architecture and SIG Testing input (including test refactors)
    - [ ] e2e Tests for all Beta API Operations (endpoints)
    - [ ] (R) Ensure GA e2e tests meet requirements for Conformance Tests
    - [ ] (R) Minimum Two Week Window for GA e2e tests to prove flake free
- [ ] (R) Graduation criteria is in place
    - [ ] (R) all GA Endpoints must be hit by Conformance Tests
- [ ] (R) Production readiness review completed
- [ ] (R) Production readiness review approved
- [ ] "Implementation History" section is up-to-date for milestone
- [ ] User-facing documentation has been created in kubernetes/website, for publication to kubernetes.io
- [ ] Supporting documentation—e.g., additional design documents, links to mailing list discussions/SIG meetings, relevant PRs/issues, release notes

Summary

This KEP proposes a mechanism, “Node Readiness Gates”, to define custom, extensible readiness conditions for Kubernetes Nodes. The goal is to allow nodes to signal full readiness for application workloads only after specific, user-defined conditions are met. These conditions, representing the status of essential node-level components like monitoring agents, security

scanners, CNI plugins, DRA Drivers,CSI drivers or configuration patchers would be defined in the Node spec, and their status updated by external controllers. The node is considered fully-schedulable only when the components are confirmed operational by meeting the readiness conditions. This complements the existing Kubelet managed `Ready` condition by providing granular control over node-schedulability.

Motivation

Currently, a Node's `Ready` condition primarily reflects the Kubelet's health, basic network setup (via CNI config), and its ability to run a basic pod sandbox. However, many clusters often rely on critical DaemonSets or agents (e.g., for device driver readiness, monitoring, logging, security, storage, or applying runtime configurations) that need to be fully functional on a node *before* general application pods are scheduled onto it.

Scheduling application pods onto a node where these critical components are not yet fully configured the Node can lead to:

- Missing observability data during pod startup.
- Pods starting in an insecure or non-compliant state.
- Pods fail if they depend on runtime patches or configurations applied by a DaemonSet that hasn't finished yet.
- Non-preferred pods are scheduled onto the node before driver installation and readiness state is achieved, filling up valuable node capacity

Goals

The primary goals of this KEP are to:

1. Establish clear semantics using `NodeSpec` for declaring desired prerequisites (gates) and `NodeStatus` for reporting observed operational readiness (conditions) in node, aligning with Kubernetes API conventions.
2. Define a standard API field (`spec.readinessGates`) on the Node object to specify required readiness conditions beyond the default Kubelet `Ready` state.
3. Enable external controllers / agents (daemon-sets) responsible for node-level services to report the status of these conditions by patching node `status.Conditions`, requiring only `nodes/status patch` permissions.
4. Integrate the evaluation of these readiness gates into the node schedulability checks performed by the Kube-Scheduler (via a `filter-plugin`).
5. Improve scheduling correctness by also considering the reliability of the node lifecycle by preventing application pods from being scheduled onto nodes until declared readiness gates are met.

6. Allow the system to react to critical components becoming unready (eg., during restarts or upgrades) by making the node unschedulable for new pods until the critical components report ready again.

Non-Goals

This KEP does not aim to

1. Replace the existing Kubelet-managed `Ready` condition. Node Readiness Gates act as an additional check.
2. Define how *external* controllers, or agents (daemon-sets) responsible for satisfying the readiness conditions are deployed or managed. It only provides the standard API for them to report that readiness.
3. Guarantee general pod admission order on node-bootstrap or recovery. While this KEP will enable DaemonSets' pods to be deployed earlier than non-daemon-pods until readiness-requirements are met, it does not provide a comprehensive solution for all arbitrary pod startup ordering.
4. Gate DaemonSet pod scheduling. Node Readiness Gates are not intended to block or delay the scheduling or execution of DaemonSet pods themselves. DaemonSet pods (with appropriate tolerations for initial node taints) are often the very components responsible for satisfying the readiness gate conditions and therefore must be allowed to run early in the node lifecycle.
5. Directly modify Kubelet's admission or pod-startup logic for existing pods. This KEP does not implement changes within Kubelet to make it actively check NodeReadinessGates or their corresponding status.Conditions before starting or restarting pods already assigned to the Node.
6. Directly manage pod execution on the node. Node Readiness Gates are a mechanism to only gate the scheduling of new pods based on declared readiness requirements. It does not evict or manage the life-cycle of pods already running on the node beyond the initial scheduling decision influenced by the gates.

Proposal

Main-Idea:

This proposal introduces a new field `readinessGates` to the `NodeSpec` and leveraging the existing `NodeStatus.Conditions` array to make node readiness/schedulability dependent on custom criteria defined by the cluster administrator or specific workloads. This would allow critical components to directly influence when a node is considered fully available for general pod scheduling.

The core idea is to introduce a mechanism where the node's transition to a fully schedulable state depends not only on the Kubelet's default `Ready` condition but also on the successful status of custom conditions defined in the Node's spec. This should look similar to Pod Readiness Gates that the node must satisfy. Components (external controllers or agents / daemon-sets) responsible for these conditions will update their respective `NodeStatus.Conditions` using a `PATCH` to the `/status` sub-resource. DaemonSets will be exempted from NodeReadiness checks.

User-Stories

Story 1: Ensuring Comprehensive Network Readiness

As a cluster administrator, I want to prevent application pods from being scheduled onto nodes until all crucial network components are fully operational (ref: kubernetes/kubernetes#130594)

Story 2: Reliable Readiness Signal for Autoscaling

As a cluster operator, I want to ensure the scaling decisions are accurate to prevent overprovisioning and stuck pods awaiting resources:

- Enable custom resources (eg: GPUs) and DRA resource drivers to self-publish their ready states, so the Pods can't schedule on it until they are available, and Cluster-Autoscaler does not perform unnecessary scale-up. (ref: kubernetes/autoscaler/7780)
- Allow CSI plugins to signal their operational readiness, providing accurate CSI node awareness to the Scheduler and Cluster-Autoscaler to prevent overcommitting. (ref: kubernetes/autoscaler#8083)

Story 3: Security Agent / Policy Readiness

As a cluster administrator, I need a kubernetes native mechanism to prevent application pods from being scheduled onto nodes where essential security or compliance-enforcing components are not yet fully operational.

Example Walkthrough

Let's consider a scenario where a node needs CNI installed, Datadog agent to be healthy and a custom runtime patch applied before being used for business pods.

Use-case 1:

- kubernetes user is running Datadog and doesn't want the node to be marked ready until verification from Datadog that the observability pieces are up and running.

    - `datadog-agent-ds`: daemonset with `hostnetwork=False` and `nodeSelector: readiness-gate.datadog.com/AgentReady="true"`
    - Patch node status: `datadog.com/AgentReady`

Use-case 2:

- kubernetes user wants to patch some runtime configuration in the node using a daemonset before running a business pod. Since this is a runtime patch, this needs to happen before other pods.
    - `runtime-patcher-ds`: daemonset with `hostNetwork=False` and `nodeAffinity: matchExpressions: {key: readiness-gate.ai-corp.com/RuntimePatchApplied, operator: Exists}`
    - Execute a patch script and sets node status `ai-corp.com/RuntimePatchApplied`

Use-case 3:

- kubernetes user intends to install Cilium CNI for pod-network. This needs to be ready before the pods that are `hostNetwork: false` could be admitted.
    - `cilium-cni-install-ds`: daemonset with `hostNetwork=True` and `nodeSelector: readiness-gate.network.kubernetes.io/CNIReady="true"`
    - Configure CNI and sets CNI readiness as node status `network.kubernetes.io/CNIReady=True`

Detailed flow

1. Node is created with configuration:

Kubelet Configuration:

Kubelet upon self-registration will add the `nodeReadinessGates` to the node-spec. Kubelet itself will not evaluate these conditions, it will only populate this in node-spec for external controllers / agents to consume.

```
None
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
```

```
# -- Proposed Configuration --
# `nodeReadinessGates` specifies the conditions that will be added
#  by kubelet to nodespec after self-registration.
nodeReadinessGates:
- conditionType: datadog.com/AgentReady
  timeoutSeconds: 180
  failureAction: BypassWithWarning
- conditionType: "ai-corp.com/RuntimePatchApplied"
  timeoutSeconds: 300
  failureAction: Taint
  readinessTaint:
    key: ai-corp.com/runtime-patch-not-installed
    effect: NoSchedule
    value: "true"
- conditionType: network.kubernetes.io/CNIReady
  timeoutSeconds: 180
  failureAction: Taint
  readinessTaint:
    key: node.cilium.io/agent-not-ready       # example for existing taint
based implementation migration
    effect: NoSchedule
```

NodeSpec:

None

```
apiVersion: v1
kind: Node
metadata:
  name: node-123
  labels:
       readiness-gate.datadog.com/AgentReady: true
       readiness-gate.ai-corp.com/RuntimePatchApplied: true
       readiness-gate.network.kubernetes.io/CNIReady: true
spec:
  readinessGates:
    - conditionType: datadog.com/AgentReady
      timeoutSeconds: 180
      failureAction: BypassWithWarning
    - conditionType: ai-corp.com/RuntimePatchApplied
```

```
      timeoutSeconds: 300
      failureAction: Taint
      readinessTaint:
        key: ai-corp.com/runtime-patch-not-installed
        effect: NoSchedule
        value: "true"
    - conditionType: network.kubernetes.io/CNIReady
      timeoutSeconds: 180
      failureAction: Taint
      readinessTaint:
        key: node.cilium.io/agent-not-ready
        effect: NoSchedule
```

2. Kubelet starts and registers node

```
None
status:
  conditions:
    - type: Ready
      status: "False"
      reason: KubeletStarting
      message: Kubelet is starting.
```

- Node is **not** schedulable because `Ready` is not `True` and the conditions defined in `readinessGates` are missing from the status.
- DaemonSet controller adds default tolerations to `datadog-agent-ds`, `runtime-patcher-ds` and `cilium-cni-install-ds` and these are scheduled at the node.

3. Kubelet becomes ready

```
None
status:
  conditions:
    - type: Ready
      status: "True"
      reason: KubeletReady
```

```
      message: Kubelet is ready.
```

- The node is still **not** considered fully schedulable because the `readinessGates` conditions are not yet met.

4. Network Controller (eg: CNI daemon-set) reports CNI is ready
    - CNI plugin initializes on the node and handles network-setup.
    - Associated controller updates the node-status `network.kubernetes.io/CNIReady`

```
None
status:
  conditions:
    - type: Ready
      status: "True"
      reason: KubeletReady
      message: Kubelet is ready.
    - type: network.kubernetes.io/CNIReady # Added/Updated
      status: True
      reason: CNIPluginReady
      message: "Cilium version: quay.io/cilium/cilium:v1.9.1"
```

- The node is **not** considered schedulable because the readiness-gates conditions are not met.

6. External Controller (eg: Daemonset): Datadog agent is ready
    - Datadog agent (daemonset) starts on `node-123`.
    - Agent / operator monitors status of the agent health
    - Operator uses kubernetes api to update node `/status`

```
None
status:
  conditions:
    - type: Ready
      status: "True"
      # ...
    - type: "network.kubernetes.io/CNIReady"
```

```
      status: "True"
       # ...
    - type: "datadog.com/AgentReady" # Added/Updated
      status: "True"
      reason: AgentHealthy
      message: "Datadog agent started successfully."
```

7. External Controller: Runtime is patched / ready

```
None
status:
  conditions:
    - type: Ready
      status: "True"
      # ...
    - type: "network.kubernetes.io/CNIReady"
      status: "True"
       # ...
    - type: "datadog.com/AgentReady"
      status: "True"
      # ...
    - type: "ai-corp.com/RuntimePatchApplied" # Added/Updated
      status: "True"
      reason: PatchSucceeded
      message: "Runtime patch v1.2 applied successfully."
```

8. Node becomes fully-schedulable.

Design Details

API Changes

1. `NodeSpec.ReadinessGates` **(New Field):**
    - Add an optional field `nodeReadinessGates` to `NodeSpec`.
    - Type: `[]NodeReadinessGate`
    - `NodeReadinessGate` struct

None

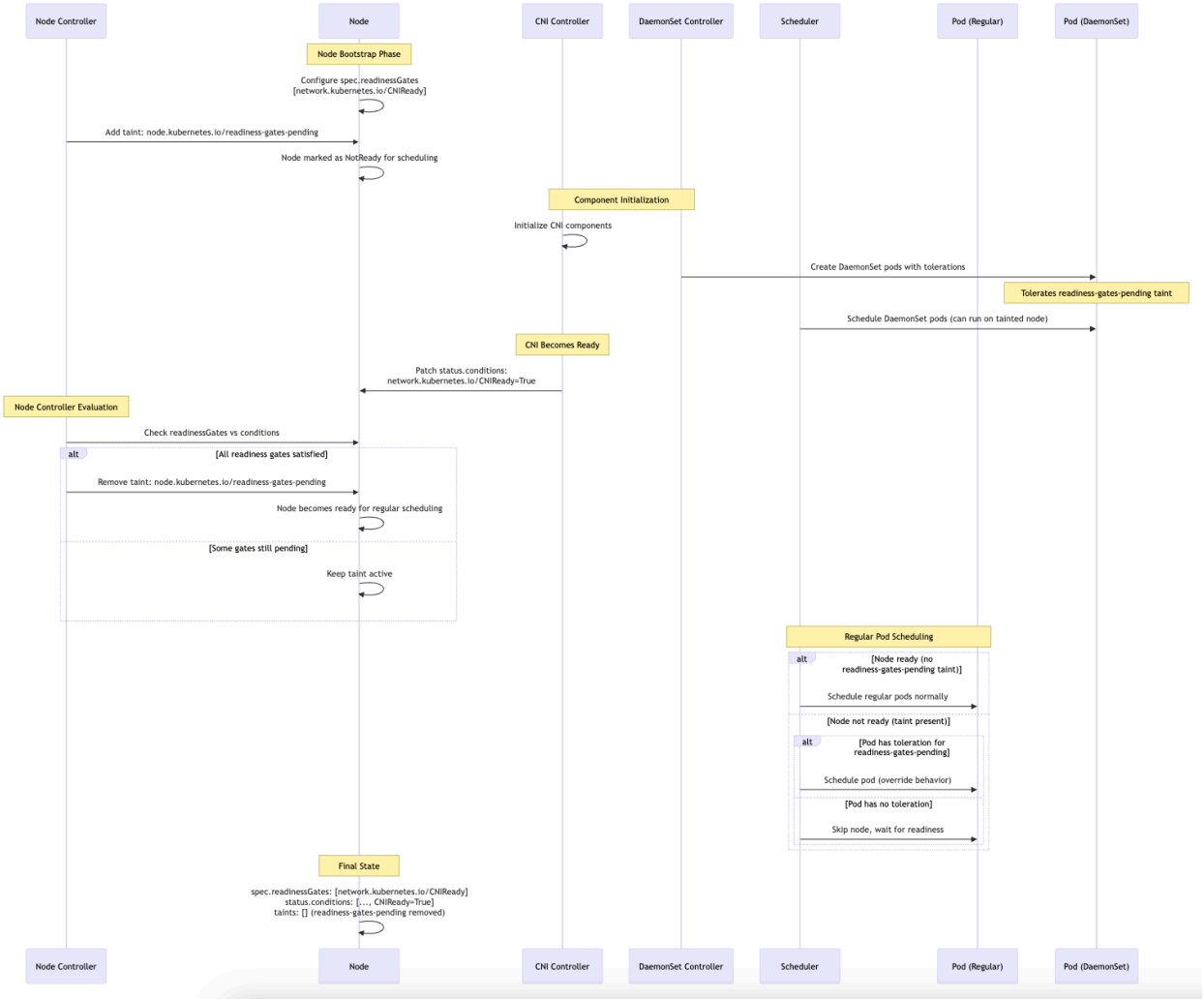
```
// NodeReadinessGate specifies a condition that  must be true for the node to
be considered fully-schedulable.
type NodeReadinessGate {
  // ConditionType refers to a condition in the Node's `status.Condition` array
with matching type.
  // Each conditionType must be unique within node.spec.readinessGates and must
be a valid dns-domain
  // name (eg: domainname.com/MyCondition).
  // +required
  ConditionType v1.NodeConditionType `json:”conditionType”`

  // TimeoutSeconds is the duration in seconds the system should wait for the
condition to be satisfied
  // before taking the configured failure action.
  // +required
  TimeoutSeconds *int32 `json:"timeoutSeconds,omitempty"`

  // FailureAction defines what action to take when the condition is not
satisfied within the timeout period.
  // Valid values are: Taint, BypassWithWarning.
  // Default is Taint.
  // +required
  FailureAction string `json:"failureAction,omitempty"`

  // ReadinessTaint provides the taint to apply when the failure action is
Taint.
  // Required if failureAction is Taint, ignored otherwise.
  // +optional
  ReadinessTaint *v1.Taint `json:"readinessTaint,omitempty"`
}

// NodeSpec describes the attributes that a node is created with
type NodeSpec struct {
  // Existing fields..

  // ReadinessGates
  ReadinessGates []NodeReadinessGate `json:"readinessGates,omitempty"
patchStrategy:"merge" patchMergeKey:"conditionType"`
}
```

- **API Validation:**
  - `ConditionType` within `ReadinessGates` must be unique.
  - `ConditionType` must adhere to dns-domain conventions.

    - Node `readinessGates` are fully mutable.

2. `NodeStatus.Conditions` **(Existing Field):**
    - This existing array would be used by external controllers to report the status of the conditions listed in `spec.readinessGates`.
    - The external controller would add or update entries in this array, setting the `type` to match the gate's `conditionType`, and setting the `status` field to `"True"`, `"False"`, or `"Unknown"`, along with `reason` and `message` fields for details.

High Level Design



Labels for Readiness Gates

To improve the integration of node readiness-gates, each gate supported by a node will be represented by a unique label within a reserved namespace:

`readiness-gate.<gate-name>: true`. This approach is consistent with existing node metadata practices, such as topology labels and NFD feature discovery labels. This will enable users to be able to easily select nodes based on specific readiness gates using label selectors and node affinity rules.

Kubelet will dynamically manage these labels during node-status synchronization. Kubelet will detect the configured readiness gates and ensure corresponding labels are set and cleaned up if necessary. This ensures advertised node readiness-gates are accurate throughout the node-lifecycle.

Example:

```
None
// labels
readiness-gate.datadog.com/AgentReady: true
readiness-gate.ai-corp.com/RuntimePatchApplied: true
readiness-gate.network.kubernetes.io/CNIReady: true
```

Pros

- Allows users to target essential DaemonSets satisfying readiness-conditions based on specific readiness–gate labels.

  Example:

```
None
// matching readiness-gate labels allow targeting specific nodes and integrates
with existing kubernetes tooling.

spec:
  nodeSelector:
    environment: test
    nvidia.com/gpu.product: A100
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - {key: kubernetes.io/os, operator: In, values: [linux]}
            - {key: readiness-gate.ai-corp.com/RuntimePatchApplied, operator:
Exists}
  containers:
```

```
- name: my-script
image: debian
command: ["/bin/sh", "-c", "path/to/run-patch-script.sh"]
```

- Facilitate testing of new readiness-gate requirements on a subset of nodes by applying labels selectively, thus avoiding cluster-wide impact.

Cons

- `conditionType` already a subdomain with length restrictions, risks truncation. This can be lessened by reducing the maximum allowed `conditionType` length to 238 to include the `readiness-gate.` label prefix requirement.
- Labels are already cluttered, adding a number of 'readiness-gate' labels will only make it worse.

Standardized Condition Patterns

To promote consistency for common / standard readiness scenarios, this KEP proposes the adoption of standardized `conditionType` prefixes for certain readinessGates.

Example: Node Network-Readiness

Network readiness can involve multiple sub-components / plugins (CNI, IPAM, Network-Policy etc.). This can be expressed with individual readiness-conditions for improving node-visibility as follows,

- Granular conditions related to network sub-systems would use the prefix `network.kubernetes.io/`.
- Cluster administrators would list these as network readiness-gates, eg: `network.kubernetes.io/CNIReady` and `network.kubernetes.io/NetworkProxyReady`, if they are direct prerequisites for general workload scheduling.
- An external controller could optionally aggregate on these conditions to determine the network-health of the node based on these standardized `network.kubernetes.io/` prefix listed in `spec.readinessGates`.
- Meaning: `network.kubernetes.io/CNIReady` condition will signal the pod-network is configured and operational on the node.
- Node-Behavior: Kubelet will not monitor or aggregate external conditions as it should not be responsible for watching arbitrary conditions set by various other agents.

Node Readiness Timeout

The primary risk with custom readiness-gates is that if a condition never becomes satisfied due to misconfiguration, buggy controller or component failures, nodes could remain in a perpetually “not-ready for business workloads” state, effectively locking them out of the cluster’s usable capacity.

This proposal addresses this risk by introducing a configurable timeout duration for each readiness-gate. After the timeout, the condition will transition to `{status:“Unknown”, reason:“TimeoutExceeded”}` state following a failure action as fallback. The possible failure actions are:

a. `Taint`: applies a specific taint when timeout occurs (default).
b. `BypassWithWarning`: adds a warning at the node-object

Pros:

- Integrates well with existing tooling support for taints.
- Layered readiness can be implemented with established methods, such as ‘tolerations’ in pods, and can address specific missing readiness conditions.
- By allowing a specific fallback taint for each readiness-gate, this approach allows existing taint-toleration based components to easily be migrated to readiness-gates.
- Extensible failure-action allows for future actions beyond tainting for node readiness-gates. For instance, cordon or automated remediation-actions for node recovery.
- Taint as a side-effect to readiness-gate failure allows components to implement more aggressive workload-protection when severe node issues requiring maintenance occur. Eg: driver detecting device failure can apply `NoSchedule` or `NoExecute` effects on the node by linking their health status with their readiness conditions.

Cons:

- Kubelet 'bypassing’ `status.conditions` for external controllers is problematic as it can hide underlying problems to automation. This is a compromise for protecting node-lockouts. For conditions that are critical, taint failure-action could be configured for the readiness-gate for stronger guarantees.
- Lack of global timeout means the node schedulability will have to wait until the longest readiness timeout. This was a deliberate design choice to accommodate varying initialization durations across different components.
- Tight coupling with pods when used with taint as failure-action, but is a known pattern. Application-owners need to add tolerations for critical readiness-requirements.

Alternate options evaluated for handing node-lockout risks:

- Single default timeout in kubelet-configuration for all readiness-gates eg: `node-readiness-gate-timeout`.

    - Drawback: lack of granular control over readiness-gates. Readiness-gates and timeouts are domain specific, condition owners are best positioned to define the timeout requirements for specific conditions.
- A cluster-level API to force-satisfy specific readiness-gates.
    - Drawback: implementation complexity. This API action has to reflect at the scheduler plugin to apply for all pods.
- Partial readiness scheduling by allowing pods to define the node readiness-requirements they require. This option is more about optimizing scheduling when nodes are generally ready than unblocking a stuck node.
    - **Advantages**:
        - enables selective scheduling and improves node utilization (eg: non-gpu workloads don't need to be blocked on GPUReady).
        - pod / application owners only need to define their direct node requirements, and don't need to know what else it has to tolerate.
    - **Disadvantages**:
        - weakens node-level readiness guarantees, as pods might land and wait for only their specified requirements.
        - while this is helpful for situations such as pod-failures due to missing-dependencies (eg: missing driver or storage not ready), it could conflict with infrastructure admin readiness requirements (eg: security-policies). This could be mitigated by handling cluster-scoped requirements in pod admission-controllers.
    - This optimization falls outside the scope of this KEP and could be explored in a future KEP focusing on additive workload-level node-readiness requirements.

Implementation considerations:

1. Timeouts will be handled as node-local actions without involving external actors to avoid node-lockout in network partition scenarios.
2. Individual readiness-timeouts begin when the node reports kubelet-managed `"Ready=True"` condition, so if there's a kubelet restart, it will reset all the timer for conditions.
3. Kubelet: Track individual timeouts and update the corresponding status.Conditions with `{status:"Unknown"`, `reason:"TimeoutExceeded"}`.
4. NodeLifeCycleController: If the failureAction was `BypassWithWarning`, NLC will record a warning event at the node-object. If the failureAction was `TaintNode`, NLC will create a taint (as per configuration) against the node on timeout.
5. If this is a result of delayed action, the agent will eventually update the condition correctly. If there's a side-effect (taint), NLC will be required to recover the node by untainting (similar to handling existing node-conditions today).
6. When a new ReadinessGate gets added after the node is already Ready, NLC will detect the new gate is added from the node watch and will start the timeout calculation when the gate is added.

Why Kubelet shouldn't handle the taints (along with timeouts) -

a. Kubelet doesn't watch its own node object; this means that external condition updates will be visible to kubelet only after the node status sync cycle. This will cause a node-update latency for nodes based on `nodeStatusUpdateFrequency`.
b. Kubelet handling taints for timeouts will introduce additional complexity and weaken security-posture for nodes.
c. In line with the current responsibility model (eg: node pressure related conditions), kubelet will set node-managed conditions, and the node-lifecycle-controller will manage taints.

Cons:

a. Delay in Kubelet observing the externally set readiness conditions (API server dependency) on the node, could cause incorrect / early 'node-timeout' determination by kubelet, especially when the timeout configurations are small.
b. Node-restart in a slow network (or partition) could make this even more a concern with kubelet on bootstrap will not be able to reliably handle timeouts. But both of these concerns can be accommodated by having a longer buffer for timeout conditions.

Post Taint operations:

detail the admin and agent flow to recover this?

Evaluation Logic

Scheduling Condition:

A Node would be considered fully ready for scheduling general workloads only if:

1. The standard `Ready` condition in `node.status.conditions` has `status: "True"`.
2. **AND** all declared gates are satisfied:
    a. For *every* `conditionType` listed in `node.spec.readinessGates`, there is a corresponding entry in `node.status.conditions`. If a declared gate's condition is missing from the status, the gate is considered as not satisfied.
    b. The found conditions must satisfy one of the following:
        i. its `status: "True"`.
        ii. **OR** `status: "Unknown" and reason: "TimeoutExceeded"`

Scope of Configuration

Declaring Gates (`spec.readinessGates`):

This KEP defines the API field on the `Node` object for listing required `conditionTypes`. There will also be standardized `conditionTypes` in `kubernetes.io/` and subdomains.

Reporting Status (`status.conditions`):
This KEP relies on components using `node.status.conditions` array to report their readiness for the declared gate types.

Methods of Populating `spec.readinessGates`:
While crucial for usability, this KEP does *not* mandate a specific method for populating `spec.readinessGates` on nodes. It focuses on the *existence* and *evaluation* mechanism of the `spec.readinessGates` field, assuming it gets populated by one of the below (or other) mechanisms:

- Statically defined at *KubeletConfiguration* at node / kubelet bootstrap.
- Dynamically configured via kubernetes-native, cluster-level mechanisms such as:
    - Mutating Admission Webhooks
    - CEL Admission control (xref: #3488)
    - NodeClass

Addressing Specific Scenarios:

- Kubelet/containerd customization requiring restarts:
    - The restart itself is out of scope.
    - A Readiness Gate, such as `node.my-corp.com/CustomConfigApplied`, can be explicitly set by an agent or controller to manage this. Following a kubelet restart, the agent verifies the operational status of the new configuration and then sets the gate to `True`.
- Changing kernel command line which requires reboot:
    - The reboot and kernel param change are out of scope.
        - This fits more in 'Node Capability' which differs from 'Node Readiness'. Capabilities define the "what" while Readiness Gates aims to answer the question "is this prerequisite component operational now", both address different layers of node suitability.
    - A Readiness Gate (eg: `feature.my-corp.com/KernelFeatureXActive`) could be used. An agent would check for the feature post-reboot and update the condition.
- Installing a systemd service:
    - The installation (eg., by DaemonSet) is out of scope.
    - A Readiness Gate (eg: `systemd.my-corp.com/MyServiceHealthy`) would be set to `True` by an agent (or side-car to the DaemonSet handling installation) once the systemd service is confirmed running and healthy.

Handling Ready → Not Ready Transitions

Node Readiness Transition:

The primary `Ready` condition of a Node object reflects fundamental node operation. When the kubelet's health signal changes from `True` to `False` (or `Unknown`), the basic node functionality is considered impacted. At such times, the status of custom component readiness conditions should also be reliably re-evaluated. However, these custom conditions are "owned" by specific agents/DaemonSets, and the responsibility lies with these agents to also manage the conditions. The kube-system cannot accurately determine the true-state for these domain-specific conditions.

When there is a node reboot or kubelet restart, the node will get the persisted status.Conditions from etcd. However, this information might be outdated and requires components to reassert their current status.

Principles:

1. The agent / controller managing a specific `conditionType` is the sole-authority for its readiness condition.
2. Upon its own startup or restart, an agent MUST ensure its managed condition reflects the right state (status: “True”) upon successful verification.
3. For dynamic conditions, the agents MUST promptly update their condition if the underlying component’s readiness changes at any time.
4. Patches to NodeStatus MUST only occur on meaningful changes in status, reason or message to avoid API load.

Scenarios:

Node Reboots

1. When a node reboots (BootID is detected as changed), the persisted node conditions from etcd could be stale. Kubelet will update any readiness-gate conditions that are present to `status: “Unknown”` and `reason: “NodeRestarted”` to ensure the components reassert their conditions.
2. During node reboot, all pods in it, including the DaemonSets responsible for readiness-gate conditions are also restarted.
3. Upon startup, these agents ***must*** re-evaluate their readiness states.
4. The agents upon starting will set their respective status.Condition to an appropriate initial state, typically `status:”False”` and `reason: “Initializing”`.
5. After they successfully initialized and verified their functionality should they update their readiness condition to `status:”True”`.

Kubelet Restart

1. Kubelet will not influence the readiness conditions as there is no BootID change.
2. The agent pods might still be running.

3. If an agent's readiness status is independent of the Kubelet's running state, its condition will correctly remain as "True".
4. If the agent's readiness status depends on the Kubelet's availability, the agent will detect the kubelet-restart, considers itself as not ready and update its condition `status: "False"`. When kubelet is back and the agent re-establishes its connection, it will update the condition status to `"True"`.

Network partition

1. The agents on the node are running and consider themselves ready locally.
2. When there is loss of network, and the node-isolation lasts more than `node-monitor-grace-period` (default `50s`), the node-lifecycle-controller sets the node as 'unhealthy' (`Ready="Unknown"`), and also applies the `unreachable` taints.
3. The scheduler will therefore not schedule new pods onto this node, regardless of the status (stale or not) of the custom gate conditions. The main `Ready` condition acts as a fundamental gate.
4. The agents **cannot** update their corresponding `status.Condition` on the API server. The last reported status of the condition remains in the etcd.
    a. Agents running on the partitioned node will not be able to reach the API server for specific `Condition` updates.
    b. For a controller running off-node (eg: operator), it will likely not be able to determine the true state of the components running on the partitioned node. If it has network connectivity to the API server, it could still send API updates about the node, but should follow good-practice to verify the actual status on the node before setting the status of the condition. Depending on the use-case, it could transition the condition to 'Unknown' or leave as the last-known status if valid.
    c. If the external controller cannot reach the API server for condition update, it will retry to set the right status of the node.
5. If the `pod-eviction-timeout` (default `5m`) expires before network-connectivity recovers, the node-controller applies `NoExecute` taints on the node and the pods get evicted if they do not tolerate the specific taints. Critical system daemon-sets that usually tolerate these taints avoid eviction.
6. When network-connectivity is restored, Kubelet will update the main `Ready` condition to `"True"`. The agents will update their new state to their respective conditions. The scheduler will resume placing pods at the node once all the conditions are satisfied.

Component Readiness Transition:

Possible Extensions

Node Admission Control (Out Of Scope)

To provide node-recoverability guarantees upon restart, kubelet could be improved to enforce particular admission restrictions for fundamental conditions such as `node.kubernetes.io/NetworkReady`.

For instance,

1. if `network.kubernetes.io/*` readiness gates are present in the node-spec, kubelet will monitor for all the corresponding `node.status.conditions` are present with `status:"True"`.
2. Kubelet will refuse to admit pods with `hostNetwork: false` until all these conditions are met.
3. Kubelet will create the `node.kubernetes.io/NetworkReady` entry once these requirements are ready, and unblock pod admission for non-host-network pods.

Local updates:

1. Holding pod-admission from specific node conditions has risks. Pod-admission could be blocked by factors external to the node (eg: operator updating the CNI condition or API server unavailable) due to failures or network partition. Kubelet will determine `node.kubernetes.io/NetworkReady` (or equivalent local state for gating existing pods) purely based on local signals that do not require API server or external interaction during critical Kubelet startup path. This node-admission enhancement should be preferred only when a local update pattern is standardized for kube-critical paths.

Admission Timeouts:

1. Kubelet will wait for these network-conditions during admission with configurable timeouts (120s) after it becomes `Ready=True`. This compromise is required to avoid node-lockout risks.
2. If the timeout expires, and `node.kubernetes.io/NetworkReady` is not `True`, Kubelet proceeds to start 'existing' non-host-network pods anyway ("best-effort").
3. Kubelet will set another condition `KubeletNetworkPrerequisiteTimeout=True` indicating it started pods in a potentially network-degraded state.

Flapping Conditions:

1. To prevent unstable node readiness caused by frequently changing critical conditions like `node.kubernetes.io/NetworkReady`, Kubelet will wait for these conditions to remain `"True"` for *X* seconds before considering the overall network readiness as True.
2. Kubelet will update the `node.kubernetes.io/NetworkReady` condition with appropriate `reason` and `message` indicating the unexpected behavior. There will also be kubelet metrics to record the frequency of updates to capture the volatility.

Implementation Considerations:

Pod-Level Node-Readiness API (Out of Scope)

Pods could declare their 'readiness requirements' to specify their dependencies on particular node readiness conditions, offering an alternative approach to node readiness-gates. This would provide finer-grained control over the node conditions that workloads require before being scheduled.

A new field, potentially named `nodeReadinessRequirements`, could be added to the `PodSpec`. This field would allow a pod to optionally list the node conditions (e.g., `nvidia.com/GPUDriverReady`) that must be `True` for it to run on a node.

The pod scheduling logic is:

a. If `pod.spec.nodeReadinessRequirements` is defined, the scheduler would evaluate only these requirements against the `node.status.Conditions`. If a required condition, such as `nvidia.com/GPUDriversReady`, is currently `False`, the Scheduler would prevent the pod from being scheduled on that node.
b. If `pod.spec.nodeReadinessRequirements` is not defined, the scheduler falls back to evaluating for all conditions listed in the `node.spec.readinessGates`

For example, pods that do not require GPUs would not need to wait for the `GPUDriversReady` condition to become `True`.

```
None
// Need GPU ready
apiVersion: v1
kind: Pod
metadata:
  name: gpu-app-1
spec:
  nodeReadinessRequirements:
  - conditionType: "network.kubernetes.io/NetworkProxyReady"
```

```
    status: "True"
  - conditionType: "nvidia.com/GPUDriversReady"
    status: "True"
  containers:
    ...

// Does not need GPU ready
apiVersion: v1
kind: Pod
metadata:
  name: cpu-app-1
spec:
  nodeReadinessRequirements:
  - conditionType: "network.kubernetes.io/NetworkProxyReady"
    status: "True"
  containers:
    ...

// If no specific readiness requirements are specified,
// pod will not be scheduled on a node where readiness-gates are not met.
apiVersion: v1
kind: Pod
metadata:
  name: legacy-pod
spec:
  containers:
    ...
```

Pros

- Granular control and flexibility leading to better node-utilization while also ensuring critical node-dependencies.
- Pod clearly expresses its intent on node specific dependencies.

Cons

- Application developers need to be aware of the node-level dependencies. This concern could be mitigated with cluster level abstraction such as mutating webhooks based on workload characteristics.

Why is this not in readiness-gates scope?

Gating is

Test Plan

<TBD>

Graduation Criteria

<TBD>

Upgrade / Downgrade Strategy

Upgrade

Gate Disabled → Enabled:

1. Existing nodes will not have `spec.readinessGates` defined. They will continue to function as before.
2. New nodes or existing nodes where the field is added will start enforcing the gates. External controllers if involved may need deploying / updating to manage the conditions.
3. Adding new readiness-gate to existing nodes should preferably be gradual.
4. Existing nodes can co-exist with new readiness-gates enabled nodes.

Component Update:

1. When a daemonset / agent is handling a rollout following `RollingUpdate` strategy, kubernetes will terminate old pods and create new ones on a node-by-node basis (respecting `maxUnavailable` settings).
2. The new agent should ideally set its `status.condition` as `"True"` only when the new version is fully operational.
3. If the new agent updates its `status.condition` as `"False"` (with reason as `'Upgrading'`) during the upgrade process, the node will become unschedulable for *new* application pods during the upgrade window when the daemon-set is upgrading and its associated gate condition remains `"False"`.
4. Existing pods already running on the node will not be evicted by temporary Condition transition on the node.
5. Once the upgrade on the node is complete and the new version sets its condition to `"True"` the node will become schedulable again.
6. If daemonset rolling update fails based on pod's readiness probes, the daemon-set rollout on that node will stall. This is standard kubernetes behavior and requires admin intervention to unblock. If this failing agent also fails to maintain its `status.condition` for the readiness-gate, the node will remain unschedulable until the deployment is resolved and the condition is "True".

Downgrade

1. When readiness-gates are removed from a node, the scheduler will stop evaluating the `readinessGates` field. Nodes previously blocked by gates may become schedulable (based only on `Ready` condition and taints).
2. When the scheduler plugin is disabled, the `readinessGates` field remains in the spec but has no effect. External controllers managing the conditions may become redundant for scheduling purposes but can still be useful for observability. Status patching will continue but won't affect scheduling via this mechanism.

Production Readiness Review

<TBD>

Implementation History

<TBD>

Benefits

- Provides a standardized, declarative API to handle kubernetes node initialization dependencies.
- Ensures application pods are only scheduled on nodes where prerequisite services are confirmed ready.
- Decreases the need for cluster components to hold broad `nodes/patch` permissions.

Drawbacks

- Requires external controllers / agents to be written / configured correctly to manage the status conditions. Misconfigured controllers could render nodes perpetually unschedulable.

  Mitigation: Node administrations can set up monitoring to alert on specific gate failures.

- Potential for increased `node.status` patch requests to the API server, although targeted patching of conditions should be manageable.

  Mitigation: Promote standardized client-libraries and standard implementations for agents to patch `NodeStatus` on meaningful state changes, not as heart-beats. This could go in controller-runtime or a separate node-project itself under the

`kubernetes-sigs` organization and maintained by `sig-node`.

- Defining the right set of gates requires careful consideration by the cluster administrator.

  Mitigation: Pre-defined profiles or common readiness-gate patterns for critical node-services eg: CNI, service-mesh use-cases.

Alternatives

Custom Taint/Toleration controller

Existing mechanisms such as taints can be used to mark the node as 'not-schedulable' during node bootstrap. Subsequently, a custom taint-management controller will monitor the node for a specific side-effect. Once this effect is detected, the controller will untaint the node to make it available for scheduled pods.

Compare Taints and Tolerations

The common workaround involves:

1. Applying a `NoSchedule` taint (e.g.,`node.kubernetes.io/unschedulable`) to new nodes.
2. Configuring critical DaemonSets/workloads to tolerate this taint.
3. Implementing an external controller (or a daemonset / process) that monitors the status of these critical workloads on each node.
4. Once the critical workloads are deemed ready, this controller removes the taint from the node, allowing general pods to be scheduled.

Readiness gates provides advantages over existing taints-based approach:

	Taints / Tolerations **(custom taint-lifecycle controller)**	**Taints / Tolerations** **(component specific taint-controller agents)**	**Node Readiness Gates**
	Two scenarios: **a.** 'npd' like node-agent on each node, acts as a taint-controller that needs to have high-privileges. The sole purpose of this controller is to handle taint management for the node. b. An operator that watches nodes and relies on 'annotations' or 'labels' for node-understanding and patches different taint life-cycles.	Each critical agent/DaemonSet has its own taint-controller (or built-in logic) e.g., CNI agent controller, GPU driver agent controller, Istio agent controller, running on the node. Each is responsible for: a. Knowing its own readiness state locally. b. If it's not ready, ensuring a specific taint it "owns" (e.g., `cni.example.com/agent-not-ready:NoSchedule`) is present on its `node.spec.taints`. c. If it becomes ready, removing its specific taint from `node.spec.taints`.	Each component reports its health / status as an update to node.status.condition. a. For one-time run needs (eg: a patch script), this would be an init-container that runs to completion. b. For 'agents' this could be a standard side-car pattern that will integrate with their health-signal. This could be a standard implementation / helper library that could be provided as reference for implementation. It will look something like

			None
			`# side-car periodically checks the main container`
			`# and reports its readiness condition.`
			`name: readiness-reporter`
			`image: condition-reporter:v1.0`
			`securityContext:`
			`  runAsNonRoot: true`
			`env:`
			`- name: NODE_NAME`
			`  valueFrom:`
			`  fieldRef:`
			`  fieldPath: spec.nodeName`
			`- name: CONDITION_TYPE`
			`  value: "network.kubernetes.io/cni-configured"`
			`- name: CHECK_POINT`
			`  value: "http://localhost:8080/healthz"`
			`- name: POLL_INTERVAL`
			`  value: 30s`
			`- name: REPORT_STRATEGY`
			`  value: ReportOnSuccessAndTerminate`

High Privileges	`nodes/patch` privileges needed for taint controller, which is seen as security-risk for wider enablement.	Each agent needs its own high-privilege (node/patch) controller for managing taints exposing the surface further wide.	Reduces the concern by needing only `nodes/status patch` permission
Visibility	Taints are opaque; hard to debug *why* a node is still tainted. Taints are either present or not. Doesn't tell anything about the failures, which are buried deep into the agent logs.	Same as previous	Give better / granular visibility into which components are ready by looking at node-conditions. Single glass of pane access to readiness failures, providing better debuggabililty exposing the observed failure conditions and reasons directly on the node object.
Standardization	Taints are not designed for expressing node-prerequisites. Domain specific custom taints with different logic hinders standard tooling: eg: ambiguous meaning for prefixes or taint labels might conflict	Same as previous	Readiness-gates specifically designed for expressing readiness information with standardized dns-style conditions provide a hierarchical structure and ownership isolation.
Semantic clarity	NodeSpec is for 'desired' node state, node observations don't fit in there.	Same as previous	Node Readiness Gates and Conditions capture declarative intent and observed component statuses separately.
Complexity	Separate custom controllers for taint lifecycle-management in a cluster gets especially complex when multiple component readiness / dependencies are involved. This gets harder especially in a practical enterprise setup where	Reduces the 'shared' burden of maintaining a common external controller by isolating Multiple agent controllers need to coordinate on 'removing' taints. One agent could remove taint not knowing other components could still need it.	Simplify agents / external-controllers to only report their own health / status without complex coordination or taint management logic. Each component reports only its status. Shifts readiness evaluation to a built-in component (scheduler) for

	multiple-agents are owned by different teams and need to collaborate to achieve this. Need reconciliation loops to ensure taints are correct, or complex tolerations in each of your workload pods. App teams need visibility into the infrastructure components and other agents - alternatively separate mutating webhook for ingesting tolerations for dependency requirements.	Taint isolation could be achieved at a cost: each component using unique taints for various states could lead to a proliferation of tolerations that would need to be managed for every pod. There's no single entity owning the 'readiness' responsibility, making it hard to manage and debug 'why node is not ready'. Each agent (likely third-party) needs to build their own non-trivial 'taint-management' logic, in addition to business logic, which is impractical.	individual -dependencies. Provide granular readiness control on the node.

Taint/Toleration Controller Ensure Readiness with Satisfying Conditions

This option considers only part of the broader solution, including the aspect where components update the node object's status with observed conditions and a Kubernetes controller manages taints that correspond to readiness-requirements.

API comparison:
To compare the option where a 'built-in' enhanced readiness aware controller manages readiness-guarantees using the existing 'taint' api and node.status.conditions.

	Taints API	**Node Readiness Gates API**
Functional Difference	Since the readiness-controller will remove the original user-declared readiness-taints upon fulfilling conditions, the intent will be lost after establishing the initial bootstrap readiness. This will have two consequences: 1. No traces on what were the established readiness on a node. 2. Node readiness gate will be working only during node-startup time. On a bootstrapped node, when corresponding readiness-condition changes to NotReady, it will not have any effect.	User readiness intent is available as an api and can be reliably established for component restarts. Key differences: 1. Node spec captures user declared readiness intent. 2. Readiness condition changes can be guarded by readiness-gates. eg: new pods will not be scheduled during security-agent upgrade when node is in non-compliant state.
User experience	1. The application owners will specify the tolerations (for its own and other required readiness-taints) that need to be tolerated for their pods at bootstrap. 2. Different teams / component owners need to coordinate with cluster-admin knowing ahead of time what taints their pods are tolerating.	1. Cluster-Admin will specify the infrastructure readiness requirements that will be applicable for all nodes. 2. There are no ordering guarantees on the daemonsets. If specific ordering is required, cluster-admin will continue to use other existing mechanisms (eg: init-containers).

	3. Sequencing of applications is not inherently solved, but cluster-admin can manage the ordering with tolerations.
	
	Example:
	
	Cluster Admin:
	
	Added initial taints on the node:
	
	None
	`kind: KubeletConfiguration`
	
	`taints:`
	`# existing categorical taints`
	`- key: nvidia.com/gpu`
	`  value: true`
	`  effect: NoSchedule`
	`- key: team`
	`  value: ml-infra`
	`  effect: NoSchedule`
	`- key: security-level`
	`  value: high`
	`  effect: NoExecute`
	
	`# taint configuration for readiness requirements`
	`# this assumes new readiness taints follow some structure.`
	`- key: readiness-taint.datadog.com/agent-not-ready`
	`  value: true`
	`  effect: NoSchedule`
	`- key: readiness-taint.cni.example.com/pending`
	`  value: true`
	`  effect: NoSchedule`
	`- key: readiness-taint.security-agent.corp.com/pending`
	`  value: true`
	`  effect: NoSchedule`
	`- key: readiness-taint.nvidia.gpu.com/driver-not-installed`
	`  value: true`
	`  effect: NoSchedule`
	
	Optional: New Taint-Management Controller Config (`ReadinessTaintRule`) to map readiness Conditions to Taints:
	
	Why is this config required?
	
	- Conditions are owned by agents (possibly third-party) however ‘taints’ cannot be decided by the controller for the end-user - should not conflict with existing behavior. eg: auto-scaling taint considerations during Cilium installation.
	- ‘effect’ cannot be determined by the controller.
		Example:
		
		Cluster Admin:
		
		Added initial taints and readiness-gates on the node:
		
		None
		`kind: KubeletConfiguration`
		
		`# existing categorical taints`
		`- key: nvidia.com/gpu`
		`  value: true`
		`  effect: NoSchedule`
		`- key: team`
		`  value: ml-infra`
		`  effect: NoSchedule`
		`- key: security-level`
		`  value: high`
		`  effect: NoExecute`
		
		`# -- Proposed Configuration --`
		`nodeReadinessGates:`
		`-  "datadog.com/AgentReady"`
		`-  "network.kubernetes.io/CNIReady"`
		`-  "security-agent.corp.com/AgentReady"`
		`-  "device.kubernetes.io/nvidia/GPUReady"`
		
		No separate `ReadinessTaintRule` CRD is needed. `spec.readinessGates` is the single source-of-truth for prerequisites.

```
None
// hypothetical CRD ReadinessTaintRule for
// mapping taint with condition.
apiVersion: node.taintcontroller.k8s.io/v1alpha
kind: ReadinessTaintRule
metadata:
  name: cni-readiness-rule
spec:
  # Condition watched by the taint-management-controller
  readinessCondition: "cni.example.com/NetworkReady"
  # Taint to remove when condition is True
  targetTaint:
    key: "cni.example.com/pending"
    effect: "NoSchedule"
```

ML App Developer (targeted node):

```
None
// PodSpec for ML App
spec:
  tolerations:
  # Must tolerate GPU categorical taints (existing
tolerations)
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  - key: ml-infra
    operator: Exists
    effect: NoSchedule
```

Other teams (managing critical components)

- admission-controller to query for 'ReadinessTaintRule' CRDs and inject tolerations for all readiness-taints for critical daemon-sets.
- alternatively, craft individual tolerations in the pod-spec as below

```
None
// Networking DaemonSet
// PodSpec for CNI installation
spec:
  tolerations:
  # categorical taints
  - key: nvidia.com/gpu
    ...
```

ML App Developer (targeted node):

```
None
// PodSpec for ML App
spec:
  tolerations:
  # Must tolerate the GPU categorical taints (existing tolerations)
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  - key: ml-infra
    operator: Exists
    effect: NoSchedule
```

```
  - key: ml-infra
    ...

  # its own readiness-taint
  - key: readiness-taint.cni.example.com/pending
    operator: Equal
    value: True
    effect: NoSchedule
  # need to be aware-of/handle all other readiness-taints
  - key: readiness-taint.security-agent.corp.com/pending
    operator: Exists
    effect: NoSchedule
  - key: readiness-taint.datadog.com/agent-not-ready
    operator: Exists
    effect: NoSchedule

// Security DaemonSet
// PodSpec for security-agent installation
spec:
  tolerations:
  # categorical taints
  - key: nvidia.com/gpu
    ...
  - key: ml-infra
    ...

  # its own readiness-taint
  - key: readiness-taint.security-agent.corp.com/pending
    operator: Equal
    value: True
    effect: NoSchedule
  # need to be aware-of/handle all other readiness-taints
  - key: readiness-taint.datadog.com/agent-not-ready
    operator: Exists
    effect: NoSchedule


// Observability DaemonSet
// PodSpec for logging-agent installation
spec:
  tolerations:
  # categorical taints
  - key: nvidia.com/gpu
    ...
  - key: ml-infra
    ...

  # its own readiness-taint
  - key: readiness-taint.datadog.com/agent-not-ready
    operator: Equal
```

	<pre>value: True
	effect: NoSchedule</pre>	

Semantics and Intent	Node is assumed “not ready for workloads” because of a repulsive property (taints). The intent is realized by an external controller by removing these negative signals upon certain ‘status.conditions’ are fulfilled.	Node explicitly declares its prerequisites for readiness in ‘spec.readinessGates’. This intent is realized by positive confirmation from each required component (Condition=True).
API Load	Each readiness-taint has a multiplicative effect on the cluster: readiness-actors x nnn nodes. 1. Agent patches NodeStatus with Condition update. 2. TaintController is informed and reads updates from the node object. 3. TaintController patches NodeSpec to remove *each* mapping taint. Note: status.condition patch is a light-weight patch at /status subresource compared to taint-removal at /node.	The load from condition patches remains unchanged. But the ‘taint removal’ is not present. 1. Agent patches NodeStatus with Condition update. 2. TaintController is informed and reads updates from the node object.

Eventual Consistency

Why is it even necessary to address this? Like any distributed system, can we simply allow failures and rely on retries until the operations eventually succeed.

Compare Simple Retries

1. The fundamental difference between node readiness-gates and retry until success is proactive vs reactive management - proactive systems prevent the wastage of cluster resources leading to significant cost savings. In contrast, solely relying on a reactive, retry-until-success strategy can create cascading failures, particularly in large-scale workloads like ML training jobs. Readiness Gates provide critical guardrails for AI/ML workflows ensuring custom drivers, model weights, and special configurations are fully prepared prior to scheduling.
2. Node Readiness Gates bring in a lot of value outside of just scheduling correctness. For example, auto-scaling systems can leverage richer operational insights from granular readiness conditions to make better scaling decisions on different failure modes.