Kubernetes Production Readiness Review

REVIEWER: **DANIELE POLENCIC, LearnKube**

CLIENT: **ANONYMIZED**

🌟**This sample report is based on a real Kubernetes production readiness review conducted for a client team.**

The company name, people, system names, and commercially sensitive details have been removed or generalized.

The purpose of the sample is to show the structure, depth, and style of a LearnKube readiness review: how risks are identified, explained, and translated into practical recommendations.

Some tooling references come from the original engagement and may reflect the Kubernetes ecosystem at that time. Current reviews use current Kubernetes practices and recommendations.

The following summary describes the main risks found during the review.

learnKUBE

Summary

The team has invested time and effort into automation. Changes to applications are automatically built and published as container images before deployment to the cluster. However, the same amount of automation wasn't applied to the Kubernetes platform. Clusters are created manually, and dependent managed services are manually provisioned and linked to the cluster using Secrets rather than more robust, secure strategies such as identity-based access.

Not having the cluster defined as code is hurting the team on several fronts:

- It's not possible to track configuration drift across environment clusters.
- The team can't create isolated clusters to run resource-intensive tasks such as load testing. Such tasks are executed in an environment that also carries production risk, adding unnecessary risk to live traffic.
- Upgrading the cluster must be done in-place rather than creating a copy of the cluster and gradually migrating apps. In-place upgrades are difficult to roll back if there's a regression.
- Multiple environments and tools are deployed in the same cluster. The blast radius of a single application is not mitigated, and a mistake in a lower environment could quickly propagate to production.
- Applications need to be deployed geographically, but there's no easy way to replicate cluster configurations across regions.

The team stores secrets in the CI system and in the Kubernetes cluster as standard Secret objects.
Unfortunately, there's no strategy for a centralized secret management solution, such as a managed secret manager. As a consequence:

- Some secrets live only in the Kubernetes cluster. If the Secret is accidentally deleted, it's gone forever. The problem is further exacerbated by the lack of restrictions on who can access Secrets in the cluster (no RBAC).
- Secrets for multiple environments are stored in the same cluster. Any user with broad access to that cluster can also access production secrets.

The team deployed self-managed logging and monitoring stacks. These stacks are deployed in the same cluster as the production applications, potentially introducing more risk as they scale and compete for resources with the rest of the cluster. The original reason was cost control, but the team should compare that saving with the operational

time and resources required to maintain the stacks, including Pods, storage, and network bandwidth.

The applications deployed in the cluster are not designed to be run in Kubernetes. In particular:

1. There is no mechanism to handle a shutdown gracefully. When Pods are deleted, applications should trap signals and drain connections.
2. The liveness and readiness probes are not designed to support zero-downtime deployments, adding unnecessary risk with every update.
3. The applications are designed to be deployed in order, and they assume that their dependencies are already live. Such dependencies are also hardcoded in the readiness probe. If there are temporary failures in downstream dependencies, those are propagated all the way upstream.
4. There is no retry mechanism if any of the dependencies aren't ready.

The cluster is currently not suitable for multi-tenancy, as it lacks the basic building blocks to isolate workloads, networks, and resources.

The team hasn't tackled security issues, as the report highlighted several features lacking, such as:

- Missing RBAC.
- No user segregation.
- Containers are running as root.
- Secrets are accessible to anyone.
- There is no restriction on the types of images that can be deployed to the cluster.
- No audit logging.
- No NetworkPolicies or pod security controls.

The team has invested a lot of time into fine-tuning scaling for the applications. However, it seems that there are several low-hanging fruits that they could consider, such as:

- Increasing instance types.
- Overprovisioning.
- Fine-tuning CPUs and memory requests.

Recommendations and next steps

The current setup works and is an excellent starting point for exploring, familiarising with, and testing Kubernetes.
The team can successfully deploy apps, monitor traffic and core metrics, and inspect and debug issues, which is excellent.

However, it's recommended that the team invests more resources into making the cluster production-ready. In particular:

- They should define the cluster in code to mitigate issues such as incremental cluster upgrades and reduce the blast radius by segregating environments.
- They should have a coherent and reliable secrets strategy.
- They should investigate a more concrete approach for monitoring and logging in the cluster.
- They should review and amend the applications and take full advantage of what Kubernetes has to offer.

Cluster configuration

The clusters were manually created. New settings, such as adding node pools and upgrading clusters, are applied manually to one environment and then replicated in another.

Risks

- It's hard to audit changes. Who applied what? When was the change applied?
- It's time-consuming to replicate changes. One has to click the UI to apply settings to each environment cluster.
- It's hard to inspect the current configuration. One has to explore the Web UI to review the cluster's current value.
- It's hard to recreate a cluster. The team will find it challenging to migrate to a newer version of Kubernetes with breaking changes if they can't create a copy of the infrastructure.

Mitigations

- Keep the cluster configuration under version control using a tool such as Terraform.
- Recreate a new cluster using Terraform and migrate the workloads.

Shared release and production environments

The current setup has limited environment separation. Pre-production and production workloads share infrastructure.

Risks

- Any spike in traffic to a pre-production environment, such as load testing, can affect production and vice versa.
- Anyone with access to the shared environment may also have access to production data.

Mitigations

- Separate release, staging, and production environments when the risk justifies the operational overhead.
- Use more clusters.

Templating with sed

The current script for deploying into Kubernetes uses `sed` and a simple bash script.

Risks

- It's hard to onboard new members.
- The script replaces values, but it does not understand YAML. It could produce invalid YAML if misconfigured.

Mitigations

- Migrate to a tool such as `yq`, `kustomize`, or `Helm`.

Several reverse proxies

The current architecture has three layers of reverse proxies:

1. The cloud provider ingress controller.
2. The service mesh ingress controller.
3. An application reverse proxy.

Risks

- The traffic is bounced several times in the infrastructure before it reaches the Pod. The latency adds up.
- The service-mesh ingress and the application reverse proxy must be scaled in concert.
- The extra proxy layers consume additional resources and increase costs and complexity.
- It's hard to debug the flow of a single request.

Mitigations

- Remove one proxy layer and move the routing rules into the ingress layer that the team intends to operate long term.
- Alternatively, expose the service mesh gateway directly and define routes there, so traffic follows a simpler and easier-to-debug path.

Secret store

Secrets are stored in:

- Kubernetes
- The CI system

Risks

- Configuration drift — secrets are manually synchronized in the clusters.
- If the cluster is accidentally destroyed, all the secrets are lost.

Mitigations

- Use a managed secret manager, SOPS, or a dedicated secret management platform to secure your secrets.
- Use the Secrets Store CSI Driver or External Secrets Operator to inject or synchronize secrets safely.
- Restrict access to Kubernetes Secrets with RBAC and separate production secrets from lower environments.

Promotions

Artifacts are not promoted through the pipeline; instead, they are rebuilt for the release environment.

Risks

- Regressions can be introduced during container rebuilds.
- Lead time — it takes time to rebuild the container.

Mitigations

- Promote the same container image between environments. Use immutable tags or digests to identify when the artifact was generated.

Container registry

Containers are uploaded and downloaded from an external public registry.

Risks

- There's an extra charge for egress and ingress (container downloads and uploads).
- Since the registry might not be in the same region as the cluster, downloading the container can take time, increasing pod startup latency during autoscaling.

Mitigations

- Use a private registry in the same cloud provider or region as the cluster.

Rollbacks

The team has two mechanisms to roll back a deployment: `kubectl rollback` and committing to the repository.

Risks

- `kubectl rollback` has the side effect of updating the cluster state in place. Your YAML files in the repo will go out of sync with the cluster.
- It's difficult to audit `kubectl rollback`.

Mitigations

- Do not use `kubectl rollback`.

Requests and limits

The current CPU request is high relative to the available CPU on each node.

The current node size leaves limited room for scheduling multiple application replicas.

Risks

- Only a handful of Pods can be scheduled on each node. This can leave resources underutilized and trigger scaling earlier than expected.

Mitigations

- Evaluate larger node types or smaller requests based on real workload usage.
- Use the VPA autoscaler to choose safer request values for CPU and memory.
- Reduce requests for CPU and Memory.

Node sizing

The current node type has limited CPU and memory for the number and shape of workloads running on it.

Risks

- The instance can only accommodate a handful of Pods, since some memory is reserved for the Kubelet, the operating system, and the eviction threshold.
- Since the node is relatively small, some Pods may not fit alongside existing workloads, forcing the autoscaler to add a new node.
- The cluster autoscaler might trigger more frequently, increasing the overall time needed to scale the application because new nodes take time to create.

Mitigations

- Consider using larger nodes or dedicated node pools to improve bin packing, utilization, and scale-up behavior.

PVC reclaim

The current StorageClass setup doesn't include a ReclaimPolicy.

Risks

- The Persistent Volume is deleted when the PVC is detached. The data is lost.
- If the data is lost in the search cluster, recreating the index can take several hours.

Mitigations

- Change the reclaim policy to Retain.
- Use "WaitForFirstConsumer" as Volume Binding Mode.

Backups for the search cluster

The search cluster can be re-indexed, but creating a new index from scratch takes several hours.

Risks

- If the data is lost, the users won't be able to search for several hours.

Mitigations

- Add backups to the search cluster using the operator or platform-native backup mechanism OR
- Use the VolumeSnapshot feature in Kubernetes to take snapshots of the underlying volumes when the storage provider supports it.

Minimising disruptions

The application should be developed and deployed to minimize disruption. The current setup has:

- No mechanism to handle graceful shutdown.
- No disruption budgets.
- No pod anti-affinity.

Risks

- Planned and unplanned downtime could affect the service's uptime/downtime.
- Scaling down and new (rolling) updates could cause 500 errors for end users.

Mitigations

- Introduce a `preStop` hook with a 15-second pause to hotfix the deployment and handle graceful shutdown.
- Include Pod disruption budgets. As a first iteration, you could set the pod disruption budget to the number of nodes in your cluster minus two.
- Include pod anti-affinity for the deployment so that two pods don't end up on the same node (consider using a soft requirement instead of a hard requirement).

Multitenancy

When using fewer, larger clusters, it is usually better to segregate workloads and provide mechanisms to limit resources and enforce workload isolation. Kubernetes has several controls designed to help you achieve that:

1. RBAC
2. LimitRanges
3. Quotas
4. NetworkPolicies
5. Pod Security Standards

Risks

- The cluster lacks the main workload isolation and resource governance controls.
- There is no restriction on network traffic. Any Pod can talk to any Pod unrestricted.
- Namespaces can grow indefinitely. Administrators and developers can create any resource without any restrictions.
- Users can access sensitive Kubernetes resources in a namespace, including Secrets.

- Workloads are accepted into the cluster even if they don't have CPU or memory requests or limits. This could lead to nodes being oversubscribed and (eventually) random crashes.
- Workloads can (and do) run as:
    - Root permissions.
    - Privileged pods.
- Misuse of shared cluster components such as DNS, the API server, or networking components.

Mitigations

- Consider using NetworkPolicies to restrict egress and ingress traffic.
- Consider using Quotas to limit resources such as Services of type LoadBalancer.
- Consider restricting access to sensitive resources, such as Secrets, with RBAC rules.
- Consider setting LimitRanges when creating a Namespace.
- Consider using Pod Security Admission, Gatekeeper, or Kyverno to limit which Pods can be deployed in the cluster, such as disallowing root containers and privileged Pods.

Governance

It's usually a good practice to limit the types of workloads accepted by the cluster and enforce internal policies.
As an example, you might want to:

- Restrict what images can be deployed into the cluster.
- Reject all workloads that don't have requests and limits.
- Reject all workloads that don't adhere to the internal naming conventions.

Risks

- Any public image can be deployed into the cluster. This could be exploited to deploy compromised images and gain access to the cluster.
- A workload with no memory or CPU requests can be submitted to the cluster without raising any alert.
- Docker images without a tag can be submitted to the cluster.

Mitigations

- Consider using Gatekeeper or Kyverno to limit what Docker images can be submitted to the cluster.
- Consider rejecting workloads that don't include CPU and memory requests.
- Reject workloads that don't include a tag in the Docker image.

Security

Risks

- Docker images run as root.
- Users have full admin access.
- Users can read and write secrets without restriction.
- There is no audit log enabled. Users can attach to running containers undetected.

Mitigations

- Don't use root as a user in your Docker images.
- Use Pod Security Admission, SecurityContext settings, or policy engines to enforce non-root containers.
- Use RBAC to restrict what users can see and do in the cluster.
- Enable audit logging.
- Restrict who can attach to running containers in production.
- Use secure base images such as Distroless.
- Use OIDC or SSO-integrated authentication to map users and groups into the cluster with least-privilege RBAC.

Liveness and readiness probes

Liveness and readiness probes in Kubernetes govern when an application can receive traffic and when it should be restarted.

Risks

- Liveness and readiness are set to the same HTTP endpoint. When the endpoint fails, the application is removed from the Service and restarted simultaneously.

There may not be enough time to propagate the Endpoints, so there could be 500 responses.
- The probe's refresh rate is 30 seconds. If the application requests removal from the Service, there may be a 30-second delay.
- The readiness probe depends on other (micro)services. A failure downstream is propagated all the way upstream.

Mitigations

- You should have two separate endpoints for liveness and readiness.
- The readiness should not have any external dependencies.
- Review the refresh rate for the probes.

External resources

Applications deployed to the cluster connect to external services such as message queues, caches, and managed databases, which require the appropriate credentials.

Risks

- Currently, the credentials are stored in Secrets.

Mitigations

- Consider using workload identity or service-account-based access instead of static credentials where supported.

Monitoring and logging infrastructure

Having the logging and monitoring infrastructure in the same cluster as the apps is convenient but challenging.
What happens when the cluster is unresponsive, and you rely on the monitoring infrastructure to alert you? What happens when an application generates many logs that need to be ingested by the log aggregator? Could those further slow down the cluster?

Risks

- Production apps might be affected by the monitoring and logging stack.

- Logs are collected in multiple logging systems. There is no single central aggregation strategy.

Mitigations

- The monitoring and logging infrastructure should be isolated. This could be accomplished by using a dedicated cluster or, where appropriate, by relying on the cloud provider's managed logging and monitoring.
- Use a single log aggregator.

Miscellaneous

- Unpinned base images and dependencies in Dockerfiles can make container builds non-deterministic.
- Consider NodeLocal DNSCache if DNS traffic becomes a bottleneck as the cluster grows.
- If the standard HPA scaling loop is too slow or too coarse for the workload, evaluate custom metrics, KEDA, or another autoscaling controller before changing production behavior.