Best practices

Last updated by | Sughasini Karmugilan | Sep 17, 2024 at 2:50 PM GMT+2

This is a collection of suggestions and best practices to squeeze all you can from your resources.
All suggestions are given in **no particular order**.

Contents

- Apply the KISS principle where possible.
- Use the right tool for the job.
  Is AKS really the best tool you can use to execute your workload?
  Do you have the will to understand it and use it properly for proper stuff?
  Are there better tools out there you could use for any of the objectives you are trying to achieve (like a DB, *notoriously difficult to manage in Kubernetes*)?
  If other tools are better, or just suit your needs better, prefer the better tool. There is no shame in working smarter.
- Learn your tools.
  You would not use a corkscrew from the pointy end, I hope. In the same fashion, you should use any tool correctly.
  Kubernetes is **a hell of a tool**: it is *complicated*, *overkill* for most common uses, and *requires* users to know what they are doing.
  Do you, know what you are doing?

Resource management

- Plan in advance.
  Spare some room for manoeuvre for yourself, like using a *reasonably* bigger network or SKU.
- Prefer using what is already there instead of making something yourself.
  There are lots of applications freely and securely available out there you can use to simplify your life, like using Helm or Kustomize instead of writing your whole manifest file yourself and managing it.
  Also, your cluster might already provide you with features you are not aware of, like Jobs and CronJobs, Webhook mode and Operators.
- Prefer non-AKS resources like Azure's managed services when something just doesn't cut it.
  Some products, specially stateful ones like DBs, are difficult to manage inside Kubernetes or are just better as managed services.
- Prefer IaC tools to deploy and manage your resources.
  Those greatly improve your experience and make it much easier.
  Refer to Stratus' examples and references (they are **not** templates).

- Prefer *stateful* to *stateless* **when it comes to tools** (and leave *stateless* to containers).
  Stateful tools like Terraform and Helm are really helpful when it comes to garbage collecting.

Internal workings

Clusters

- Keep a **minimum** number of nodes available at all times.
  This number should *usually* be **3**.
  Clusters can usually *survive* with 1 or 2 nodes for limited time frames, but it does not mean they *should*.

Kubernetes

- Prefer an updated version of Kubernetes.
  The upstream project maintains release branches for the most recent three minor releases.
  Kubernetes 1.19 and newer receive approximately 1 year of patch support. Kubernetes 1.18 and older received approximately 9 months of patch support.

- Prefer stable versions of Kubernetes and multiple nodes for production clusters.

- Prefer consistent versions of Kubernetes components throughout all nodes.
  Components support version skew up to a point, with specific tools placing additional restrictions.

- Consider keeping separation of ownership and control and/or group related resources.
  Group different sets of resources in different Namespaces.

- Consider organizing cluster and workload resources.
  Use **descriptive Labels** on all your application's definitions. `kubectl` can leverage them to operate on **groups** of Pods instead then one at a time.

- Avoid using bare pods.
  Prefer defining them as part of a replica-based resource, like Deployments, StatefulSets, ReplicaSets or DaemonSets.

- Avoid sending traffic to pods which are not ready to manage it.
  Use Readiness and Liveness Probes for your application's Pods.

- Avoid workloads and nodes fail due limited resources being available.
  Set sensible resource requests and limits on all your application's Pods. This enables much better Pod management and scaling, and could let you squeeze out more value from your Pod.
  As welcomed side effect, this can also improve the Pods' Quality of Service .

- Prefer smaller container images.
  Reduces load time.

- Prioritize workloads leveraging Pods' Quality of Service .

- Avoid overcommitting the nodes too much.
  Overcommitted Nodes could try evicting your Pods.

- Restrict traffic between objects in the cluster.
  Use Network policies to ensure applications do not receive traffic which is not meant for them.

- Reduce container privileges.
  Enhances security.

- Use **autoscaling for your Pods**.
  K8S offers the HorizontalPodAutoscaler by default, but others are available like Keda and the Prometheus adapter (though they are not supported at the time of writing).

- In D and T, consider using the VerticalPodAutoscaler in your application to leverage trials and errors to find and set a reasonable amount of resources for your workload.

- Avoid like the Plague enabling **both** the HorizontalPodAutoscaler and the VerticalPodAutoscaler in your application.
  Talking from experience, here: your Pods will die horrible, **horrible deaths** due to the two scalers competing with each other, with little to no advice or guidance on it.

  > Update: K8S 1.27 introduced the ability (in alpha) to resize Pods' resources on the fly , and this *might*™ solve this particular issue.

- Force your app's Pods to run on tailored or dedicated Nodes when possible.
  Leverage Affinity and Anti-Affinity and/or Topology Spread Constraints in your application's definition.

- Leverage Topology Spread Constraints to encourage or force Replicas to spread on more or different Nodes.
  This avoids all of an application's Pods being executed on a single Node, and the downtimes caused by that Node becoming unavailable for some reason.

- Specify a DisruptionBudget for your application .
  This ensures a minimum number of Replicas are active at all times in the cluster.

- Continuously audit events and logs regularly, also for control plane components.

AKS

- Check Kubernetes' and AKS' release notes periodically.
  Learn about new features and the deprecation of old ones, while keeping your applications' resources up-to-date.
- Use one or more **user node pools** for your workload.
  Your system nodes can be left managing only the core services, this allows your application to use tailored Nodes, and limits the impact of ill-behaved workloads on other node pools.
  Force your app's Pods to run on the dedicated Nodes leveraging Affinity and Anti-Affinity and/or Topology Spread Constraints in your application's definition.
- Enable **autoscaling for your node pools**.
- Never ever **shutdown** your cluster, just **scale** it down.
  Read save resources (and money) on clusters for details.
- Run your cluster in **multiple Availability Zones** [1] [2] , [3] .

Azure

- Use **descriptive Tags** on all your resources.
- Keep an eye on your resources' quotas.

Application development and deployment

- Ensure your files have no *stupid* errors.
  Have **linters** check them before use, like yamllint (for yaml files), kubeconform or kube-linter (for kubernetes manifests).
- Check you are *effectively* getting what you want.
  See changes in the manifests using `kubectl --diff` or the helm-diff plugin.
- Check your deployments *can* be deployed correctly.
  Simulate the deployment using `kubectl`'s `--dry-run=client` or `--dry-run=server` option.
- Instrument applications to terminate with grace detecting and responding at least to the SIGTERM signal.
  This will let them safely shutdown, will allow for better scaling and will reduce your general management pain.
- Consider instrumenting your application (at least in Dev) with telemetry.
  Tools like OpenTelemetry and Jaeger can facilitate you to trace and visualize request flows and to analyze your application's performance and behavior.

⚠️ Currently GitOps way of deploying application is not recommended due to the below reasons:

- For every application deployment, a change request has to be raised. This is not possible in GitOps way of working.
- For GitOps to work effectively, PAT (Personal Access Token) is required to connect with azure repos. It is difficult for devops teams to provide governance of PAT and ensure it is not misused. PAT has a limited expiry ranging from 15 days to 90 days, due to this there is a risk of access continuity.
- In use of pipeline application teams can perform test and see test results but in Gitops there is no such mechanism present. This is a step back for Devops teams to setup quality control prior to deployment.
- For any accidental events like change in code, deletion of repos could cause a disaster. Also ensure version management to trace the changes to the AKS cluster is complicated.

In case you have intereset on using FluxCD please connect with CD team (Alma Stravers) to have a discussion.

Costs management

- Use the correct size, type and architecture for all your resources.

  > It's not the size which is important, it's **how** you use it 😉

  Does your Redis cache in Dev *really* need to be the same size as your Prod one?
  Are you using all the features of the VM behind it?
  Probably not, and costs are very different depending on the sizes and locations you choose to provision your resources with. Go for lower or cheaper SKUs if it does not impact your development.

- Read save resources (and money) on clusters.
  TL;DR: **scale down** your workload (Pods) first, then your cluster (Nodes) when not used.

- Stop **non-AKS** resources that do not need to run 24/7.
  Many Azure resources can be stopped when not used to save costs.

MariaDB and MySql servers, for instance (*pun intended*), support being stopped using the Azure CLI from the same pipeline you might use to interact with your AKS cluster.

- Clean up after yourself.
  Maybe you provisioned one more cluster or service just to try something it out and forgot about it. Those resources are probably bumping up your Azure bill.