Kubernetes Antipatterns

In Kubernetes, identifying and avoiding anti-patterns is crucial for maintaining a robust container orchestration environment. These misleading practices may initially appear effective but can lead to complications. This reading explores ten prevalent Kubernetes anti-patterns and recommends alternative practices for a smoother and more sustainable deployment.

1. Avoid baking configuration in container images

Containers offer the advantage of using a consistent image throughout the production process. To achieve adaptability across different environments, building images without embedding configuration directly into containers is essential.

Issue: Problems arise when images contain environment-specific artifacts that deviate from the tested version, necessitating image rebuilds and risking inadequately tested versions in production. Identification of environment-dependent container images involves spotting features like hardcoded IP addresses, passwords, and environment-specific prefixes.

Best practice: Create generic images independent of specific runtime settings. Containers enable the consistent use of a single image throughout the software lifecycle, promoting simplicity and efficiency.

2. Separate application and infrastructure deployment

Infrastructure as Code (IaC) allows defining and deploying infrastructure like writing code. While deploying infrastructure through a pipeline is advantageous, separating infrastructure and application deployment is crucial.

Issue: Using a single pipeline for both infrastructure and application deployment leads to resource and time wastage, especially when changes in application code outpace infrastructure changes.

Best practice: Split infrastructure and application deployment into separate pipelines to optimize efficiency and resource utilization.

3. Eliminate specific order in deployment

Maintaining application stability despite delays in dependencies is crucial in container orchestration. Unlike traditional fixed startup orders, Kubernetes and containers initiate components simultaneously.

Issue: Challenges arise when poor network latency disrupts communication, potentially causing pod crashes or temporary service unavailability.

Best practice: Proactively anticipate failures, establish frameworks to minimize downtime, and adopt strategies for simultaneous component initiation to enhance application resilience.

4. Set memory and CPU limits for pods problem

The default Kubernetes setting without specified resource limits allows an application to potentially monopolize the entire cluster, causing disruptions.

Best practice: Establish resource limits for all applications, conduct a thorough examination of each application's behavior under various conditions, and strike the right balance to optimize cluster performance.

5. Avoid pulling the latest tag in production problem

The utilization of the "latest" tag in production settings often results in unforeseen pod crashes, triggered by sporadic image pulls lacking specificity. This absence of clear versioning makes troubleshooting challenging, particularly during downtime resolution where swift identification of issues is paramount.

Best practice: Use specific and meaningful image tags, incorporating possibly the date and time of the build. Furthermore, it is crucial to uphold the immutability of container images and store data externally in persistent storage. Avoiding post-deployment modifications to containers ensures safer and more repeatable deployment processes.

6. Segregate production and non-production workloads problem

Relying on a single cluster for all operational needs poses challenges. Security concerns arise from default permissions and complications with non-namespaced Kubernetes resources.

Best practice: Establish a second cluster exclusively for production purposes, avoiding complexities associated with multi-tenancy. Maintain at least two clusters—one for production and one for non-production.

7. Refrain from ad-hoc deployments with kubectl edit/patch problem

Configuration drift occurs when multiple environments deviate due to unplanned deployments or changes, leading to failed deployments.

Best practice: Conduct all deployments through Git commits for comprehensive history, precise knowledge of cluster contents, and easy recreation or rollback of environments.

8. Implement health checks with liveness and readiness probes problem

Neglecting health checks can lead to various issues. Overly complex health checks with unpredictable timings can cause internal denial-of-service attacks within the cluster.

Best practice: Configure health probes for each container, use liveness and readiness probes, and prioritize robust health checks for reliable application responsiveness.

9. Prioritize secret handling and use vault problem

Embedding secrets directly into containers is poor practice. Using multiple secret handling methods or complex injection mechanisms can complicate local development and testing.

Best practice: Use a consistent secret handling strategy, consider HashiCorp Vault, handle secrets uniformly across environments, and pass them to containers during runtime for enhanced resilience and security.

10. Use controllers and avoid running multiple processes per container problem

Directly using pods in production poses limitations. Pods lack durability, automatic rescheduling, and data retention guarantees. Running multiple processes in a single container without controllers can lead to issues.

Best practice: Utilize Deployment with a replication factor, define one process per container, use multiple containers per pod if necessary, and leverage workload resources like Deployment, Job, or StatefulSet for reliability and scalability.

Conclusion

Understanding and avoiding these Kubernetes anti-patterns contribute to a more resilient and efficient container orchestration environment. Embracing best practices ensures smoother deployments, reduces technical debt, and enhances the overall stability of Kubernetes-based applications.