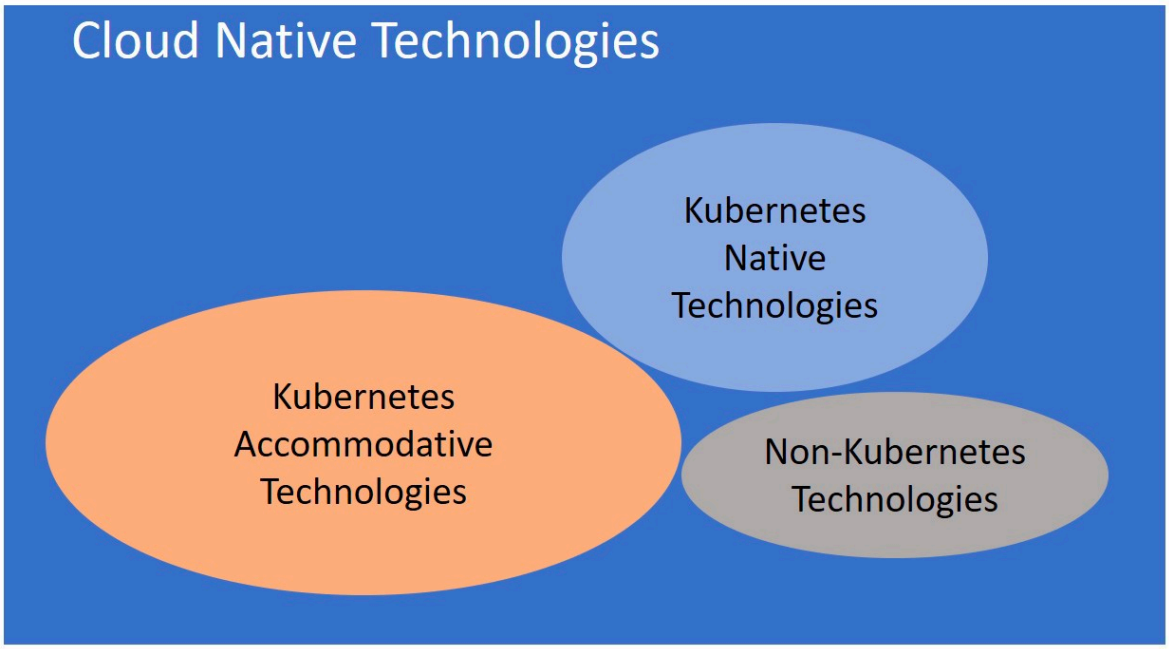
https://cloudark.medium.com/towards-a-kubernetes-native-future-3e75d7eb9d42

Towards a Kubernetes native Future

Cloud native technologies have now become mainstream. Enterprises are increasingly looking at CNCF and its flagship conference KubeCon/CloudNativeCon to keep up-to-date on the latest advancements in this fast moving field. The landscape of Cloud native technologies is quite big and new projects/standards are appearing on the scene in each KubeCon — for example Service Mesh Interface and OpenTelemetry were announced at the recently concluded KubeCon/CloudNativeCon in Barcelona. For enterprises this is great, as there are lots of options when building their cloud native platform stacks. However, the plethora of choices also creates confusion and anxiety. This is especially true if an enterprise has already decided on Kubernetes as the foundation of its cloud native approach. In this post we argue that if you have chosen Kubernetes, then stop thinking in terms of 'Cloud native'. Instead, change your mindset to think in terms of 'Kubernetes native'.

Cloud native vs. Kubernetes native
According to the original definition of Cloud native, some of the properties of such systems include, high availability, support for web-scale workloads, containers as the packaging mechanism, etc. From this generic cloud native technology definition, multiple container orchestration engines evolved e.g. Kubernetes, Amazon ECS, Docker Swarm, Apache Mesos. However, Kubernetes has now become the de-facto standard for container orchestration. Hence for enterprises that have chosen Kubernetes, it is useful to split the Cloud native technology landscape broadly into three categories based on how Kubernetes is supported — 'Kubernetes native' technologies, 'Kubernetes accommodative' technologies, and 'Non Kubernetes' technologies. This is shown in the following picture:

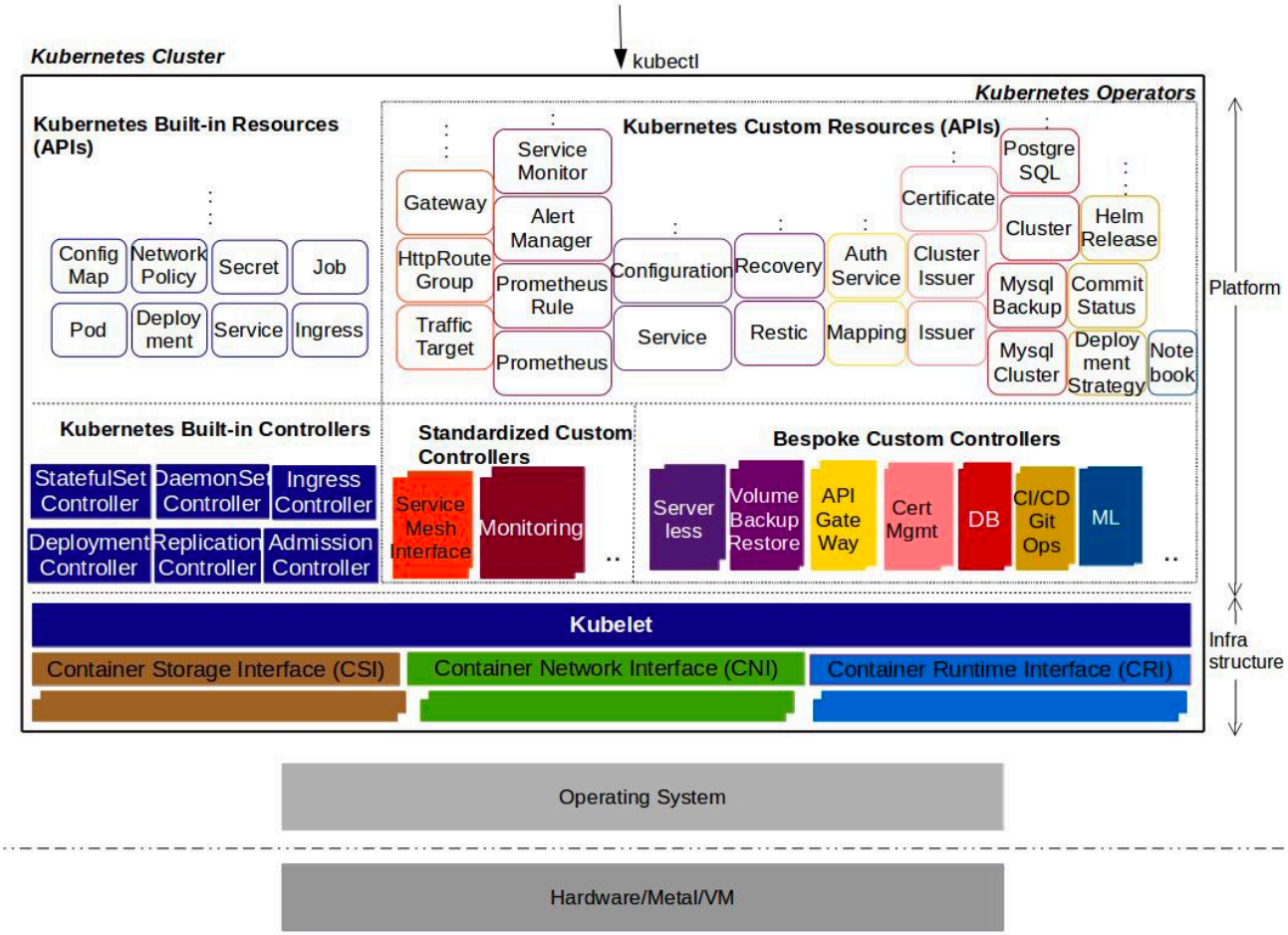


Kubernetes native technologies (tools/systems/interfaces) are those that are primarily designed and built for Kubernetes. They don't support any other container or infrastructure orchestration systems. Kubernetes accommodative technologies are those that embrace multiple orchestration mechanisms, Kubernetes being one of them. They generally existed in pre-Kubernetes era and then added support for Kubernetes in their design. Non-Kubernetes technologies are Cloud native but don't support Kubernetes. A Kubernetes native technology offers its functionality by deeply integrating with Kubernetes's core. This can manifest in multiple ways such as, extending the functionality of a Kubernetes cluster by adding new custom APIs and controllers, or by providing infrastructure plugins for the core components of networking, storage, and container runtime. Kubernetes native technologies generally work with Kubernetes's CLI ('kubectl'), can be installed on the cluster with the Kubernetes's popular package manager Helm, and they can be seamlessly integrated with Kubernetes features such as RBAC, Service accounts, Audit logs, etc.

Kubernetes native Platform Stacks

Advantages of using Kubernetes native technologies in your platform stacks are several. Such entities are best suited to leverage all Kubernetes's inherent features. They are not burdened with the need to support other systems. And because they are Kubernetes native, they can run on any Kubernetes cluster (public or private) offering hybrid multi-cloud behavior by default. Given these advantages, a natural question to ask then is — is it possible to build entire platform stacks that are Kubernetes native?

To answer this question lets first understand different parts of your Kubernetes-based platform stack. Following picture shows different parts in a Kubernetes-based stack:



The picture is conceptually divided into two parts — infrastructure and platform. The infrastructure layer consists of CSI, CNI and CRI type of Kubernetes plugins. These are standards that have emerged around container storage, container networking, and container runtime, respectively. These days, storage, networking, and container runtime vendors who want to support Kubernetes, need to implement these standards as plugins for their products. Above this infrastructure layer lies the Kubernetes's venerated controller and resource/API layer. This layer can be further divided into three parts -

- built-in components
- standardized custom components
- bespoke custom components

Built-in components consist of various standard controllers and resources such as Deployment, Pod, Service, Admission controller, Ingress controller, etc. Standardized custom components consist of Custom Controllers and Custom Resources/APIs around which some standardization has started to happen. Bespoke custom component category consists of Custom Controllers and Custom Resources/APIs for which there are no standards (yet). The components in the latter two categories corresponds to the popular 'Kubernetes Operator' pattern.

In order to build your Kubernetes native platform stack, you need to evaluate and choose right tools in both infrastructure and platform layers. Towards this, here are some questions that enterprises adopting

Kubernetes need to find answers to in collaboration between Platform engineering and Application development teams:

1. Which CSI/CNI/CRI plugin implementation do we need in our setup?
2. What all operations does our platform stack need to support? Do we need any Kubernetes Operators for it? If so, how many? Are they available already as community Open source projects? Or we need to develop or enhance some ourselves?
3. Who in the team is going to install the Operators? Who is going to use them?
4. What Custom Resources will exist in our stack? How will we find capabilities of different Custom Resources?
5. What Service Mesh technology are we using? Does it matter if our choice is compatible with Service Mesh Interface or not?
6. What API Gateway are we using? Does it support other systems besides Kubernetes? If so, what are its advantages over an equivalent Kubernetes-native API Gateway?
7. Are we going to need any admission controllers? If so, are there any custom validating/mutating webhooks that we need? Is the webhook written by our team? If not, how are we going to modify/support it when the need arises?
8. What is our authorization scheme? What different Service Accounts and RBAC policies do we need? Are the Custom Resource Definitions deployed with appropriate permissions for each namespace?
9. Have we enabled Kubernetes audit logs? Are the logs tracking REST calls on Custom Resources?
10. What aspects of our stack are cloud dependent? How difficult/easy it is to recreate the stack on different Kubernetes clusters in multi-cloud environments?

Conclusion

As the Cloud native and Kubernetes native tooling evolves, enterprises are facing plethora of choices for building their platform stacks on top of Kubernetes. Our suggestion is to use ‘Kubernetes native’ lense to evaluate different systems, tools, and vendors. This will help you constrain the problem significantly. Also, keep an eye out for standards and standardization in this space. Some tools have themselves become standards, such as Prometheus and KubeFlow. For other aspects of a platform stack, standardization will remain a recurring theme. Already standards have emerged for infrastructure elements such as networking, storage, and container runtime. This trend will continue upwards in the platform layer as evidenced by the recent announcements of Service Mesh Interface and OpenTelemetry. Don’t be surprised if standards start to emerge for other parts of the platform stack such as Serverless, API Gateways, Databases, etc. to run natively on Kubernetes.