Airship 2.0 White Paper

Life Cycle Management for Complex Cloud Infrastructure

April 2020

Authors

Alan Meadows
Rodolfo Pacheco
Ryan van Wyk

Introduction

The overall Airship goal is to enable telecommunications operators to *predictably* deliver raw infrastructure as a resilient cloud, and to easily manage the life cycle of the resulting platform, including real-time upgrades with no downtime.

This means caring for what many other solutions assume you already bring to the table before you can use them. Many of these private cloud solutions assume you are already bringing either third-party cloud infrastructure, or have done the hard work yourself of provisioning raw bare metal infrastructure including installing operating systems, configuring RAID, and setting up the network.

The challenge of doing this predictably, declaratively, and with enough flexibility for different use cases across a variety of infrastructure back-ends is substantial. This is why most tools assume this work is already done before they can be used.

Airship 2.0, in contrast, provides a complete solution in advance, providing end-to-end delivery within the following architecture:

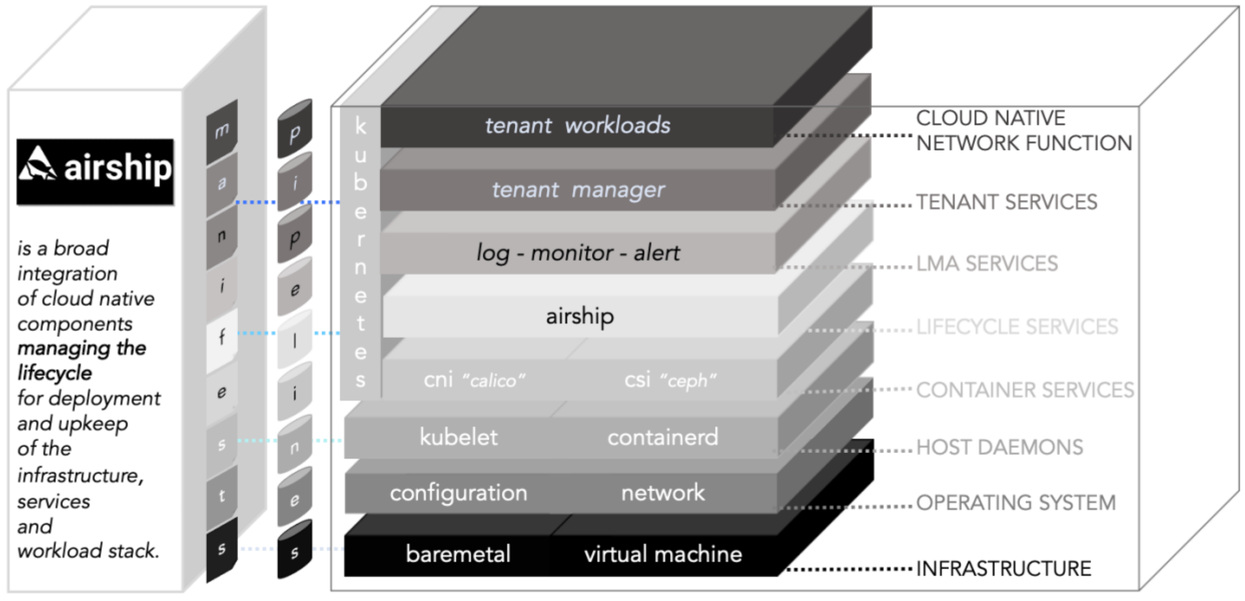

Figure 1 – Airship 2.0 Architecture

Airship 2.0 and CNCF

A challenge Airship 1.0 faced was keeping up with the frenetic pace of provisioning innovation and newly emerging lifecycle patterns occurring within CNCF and other large communities. CNCF is the Cloud Native Computing Foundation, which seeks to “build sustainable ecosystems and foster communities to support the growth and health of cloud native open source software”.

The number of infrastructure options grows daily, and the requirements on what users want to declaratively define and lifecycle for bare metal - from RAID to firmware –is constantly expanding. The only constant seems to be change itself.

Instead of attempting to customize Airship to meet every emerging solution set, **the objective of Airship 2.0 is to provide a declarative interface to assemble and orchestrate best-of-breed CNCF building blocks.** The scope of this vision encompasses provisioning and complete life cycle management of the full-stack cloud infrastructure using Kubernetes containers.

In this model, Airship is an invested consumer and integrator of these CNCF building blocks, not the author.

Successful CNCF projects are generally well-scoped in their problem-domain. They support a subset of the above solution and we view that as a requirement for each of these projects to thrive.

Airship 2.0 takes each of these CNCF projects and provides the glue that not only serves to provide a functional end-to-end integration, but a smooth operator experience tying all these projects together. Most importantly, it does this without introducing customizations to any of these components such as Kubernetes itself or other CNCF projects. This ensures that Airship users are not running forked versions in order to achieve integration.

Airship is opinionated about deployments only to the point of enabling the above goals, and seeks to step out of the way as much as possible to let end-user declarations drive configurations. This enables Airship 2.0 to maintain the rapid pace of a constantly evolving CNCF landscape.

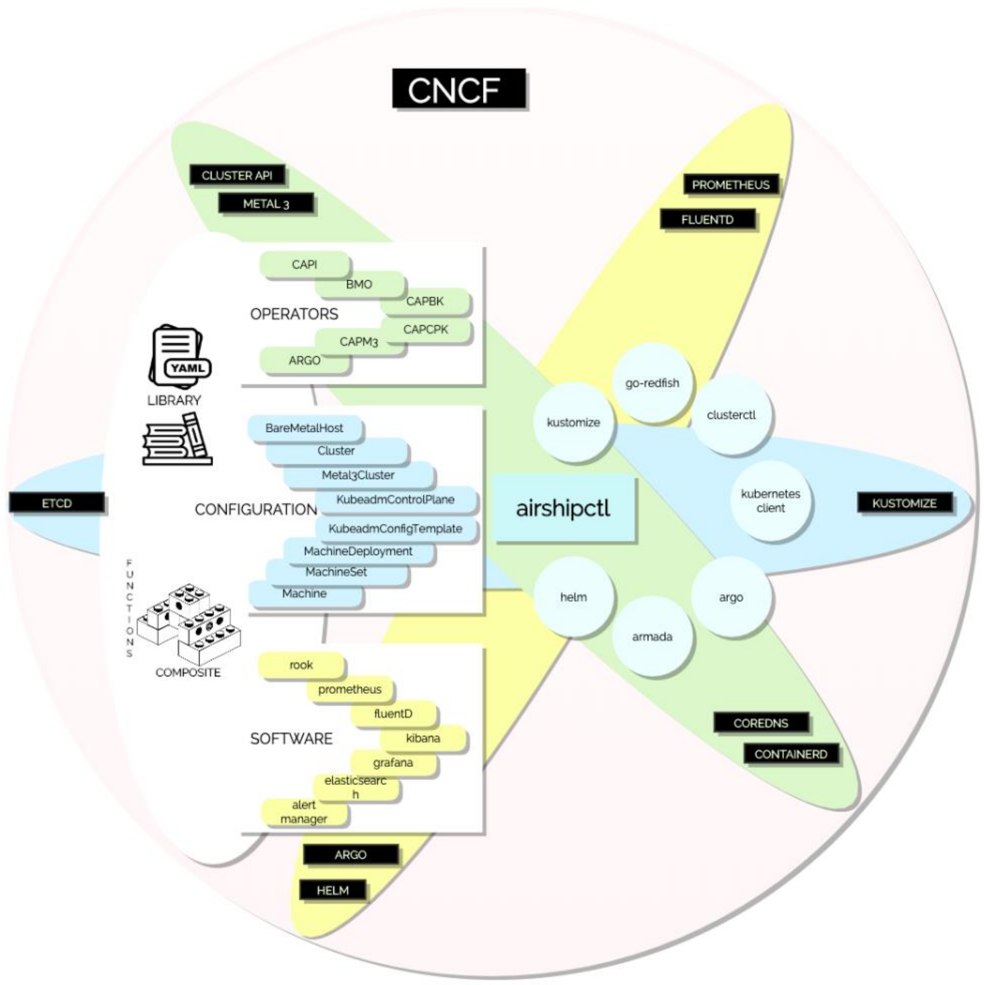

Figure 2.0 Airship with CNCF model

In order to achieve this degree of adaptability, Airship leverages a command line utility (airshipctl) to drive the deployment and life cycle management of Kubernetes clouds.

This utility articulates lifecycle management as a list of phases. For each of these phases, a YAML document set that is rendered with Kustomize, which expertly integrates and transparently utilizes the appropriate set of CNCF projects to deliver that particular phase.

For example, in order to deliver a *control plane* phase that builds a Kubernetes control plane across several machines, the phase integrates the Cluster-API components to handle infrastructure provisioning, and kubeadm to provision and configure Kubernetes on each node. With this approach, the limits of what you can achieve in terms of customization and advanced configuration is bound by the scope of these particular CNCF projects rather than Airship.

Some of these phases are built directly into *airshipctl* but users can also define their own phases and deliver them as part of an Airship document bundle and use *airshipctl* to render and deliver them. Airship provides value here as well by defining a structure to the YAML that supports this declarative and flexible interface.

Effectively, a structure for composing a library of YAML functions and compositions which make use of reusable functions to compose higher level services. Users can pull from this library of functions to build their own composites to build their own phases.

Under the Hood

Airship interfaces directly with Kubernetes to drive both initial deployments and upgrades.

The heart of that mechanism is the *airshipctl* utility which provides a one-stop command line interface to automate cloud provisioning. Because this utility works directly against Kubernetes endpoints it can ensure even the lowest set of dependencies for Airship such as the Cluster-API software components, Helm, Argo, and their related CRDs are instantiated within the site prior to delivering subsequent site-specific artifacts.

The *airshipctl* utility is effectively a “go module” that produces a single binary: *airshipctl.* This utility operates on a Kubernetes cluster security context defined within its configuration file—the same security context that you would use with the standard *kubectl* utility. In other words, a user with a basic understanding of how to use *kubectl* should have little to no difficulty learning to use *airshipctl.* The *airshipctl* utility is the main entry point for bootstrapping a cluster, collecting documents from source control, rendering and applying documents to clusters, and managing workflows.

The utility was built with flexibility and customization in mind, making it easy for organizations or even products built around Airship to add their own plugins and create their own self-contained binary builds with their plugins automatically enabled without forking the project.

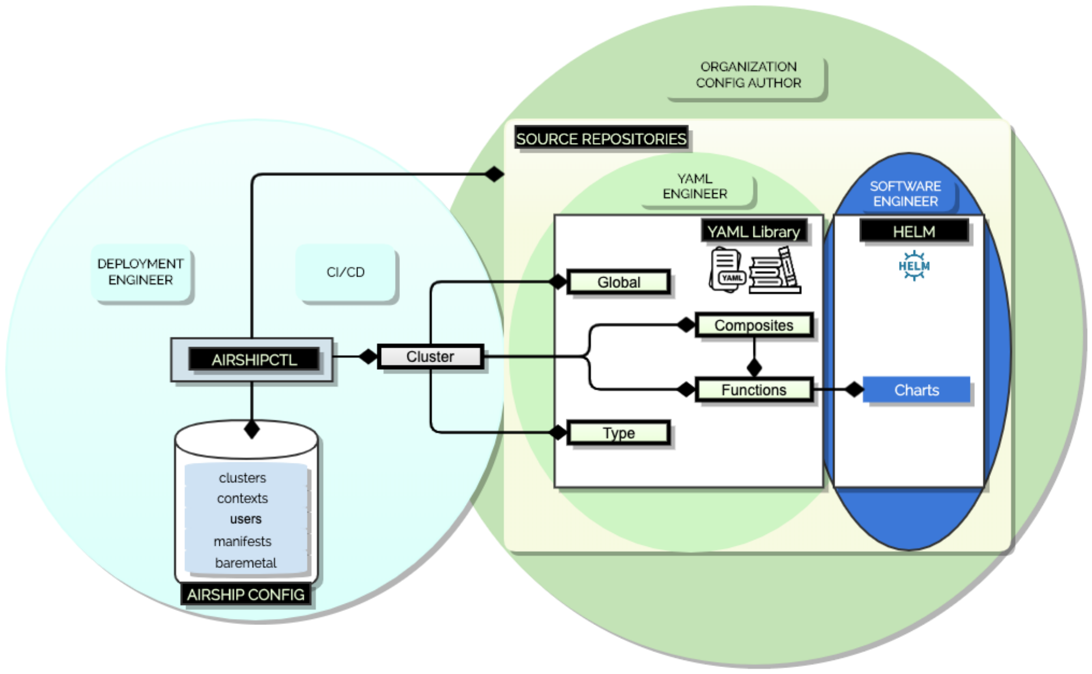

Figure 3 - AIRSHIPCTL

There are several components or resources that enable *airshipctl* to perform lifecycle management.

They are:

- **The airshipctl binary itself.** This is what drives lifecycle requests from outside the target cluster, from initial provisioning to subsequent lifecycle updates.
- **The airshipctl configuration file.** This configuration file contains a cluster-context for each environment you will manage along with their document source repository URLs in source control that it should pull down for each site. It also includes any bare metal site-specific bootstrap configuration, like where the generated ISO can be remotely pulled from when using bare metal infrastructure. The cluster-contexts allow airshipctl to authenticate and make Kubernetes API requests against the environments defined. You can easily switch between several environments defined in a single configuration file by setting your *airshipctl* current context.
- **The source repositories containing YAML libraries.** This is where site specific resources as well as resources shared across Airship environments is declared as YAML and stored. Source repositories are composed of one or more source control repository locations. The repository URLs that are specified in the airshipctl configuration are cloned or updated automatically as part of

airshipctl document pull. There is a specific directory structure to these repositories that enable *airshipctl* to expose deployment and lifecycle updates of your site as distinct phases you can launch individually or as part of a holistic pipeline. Some of these phases are builtin to *airshipctl*, and source repositories can also bring their own phase definitions.

- **Helm Artifacts.** While phases can deploy any object that can be *"kubectl applied"* to a cluster, Airship encourages managing the life-cycle of most resources outside the core Kubernetes control plane as helm charts. This means that helm is not a requirement but is highly encouraged. By including a helm-operator or Argo workflow approach that can understand how to process Custom Resources specifying helm chart locations, their values, tests to run, and so on that get applied to the cluster during any phase, you can simplify the contents of the YAML source repositories and further shift the software deployment and automation to the application owners themselves.

Phases, Functions, Composites, and YAML Libraries

Airship expresses lifecycle management as a series of phases that simplify how a Kubernetes cluster is built and subsequently managed.

The lesson learned with Airship 1.0 was that expressing a site as the *complete* series of document declarations for that site was difficult to manage and mentally digest. Phases helps people deal with a much smaller document set that is much more granularly scoped to a particular objective.

The concept of a phase is fundamental to the extensibility and flexibility of Airship. All of the phases of Airship can be used, or a subset. A phase is a simple concept in Airship: render a collection of YAML for that phase, apply it to a target Kubernetes cluster, and wait until all resources that were a part of that phase are successfully up and running.

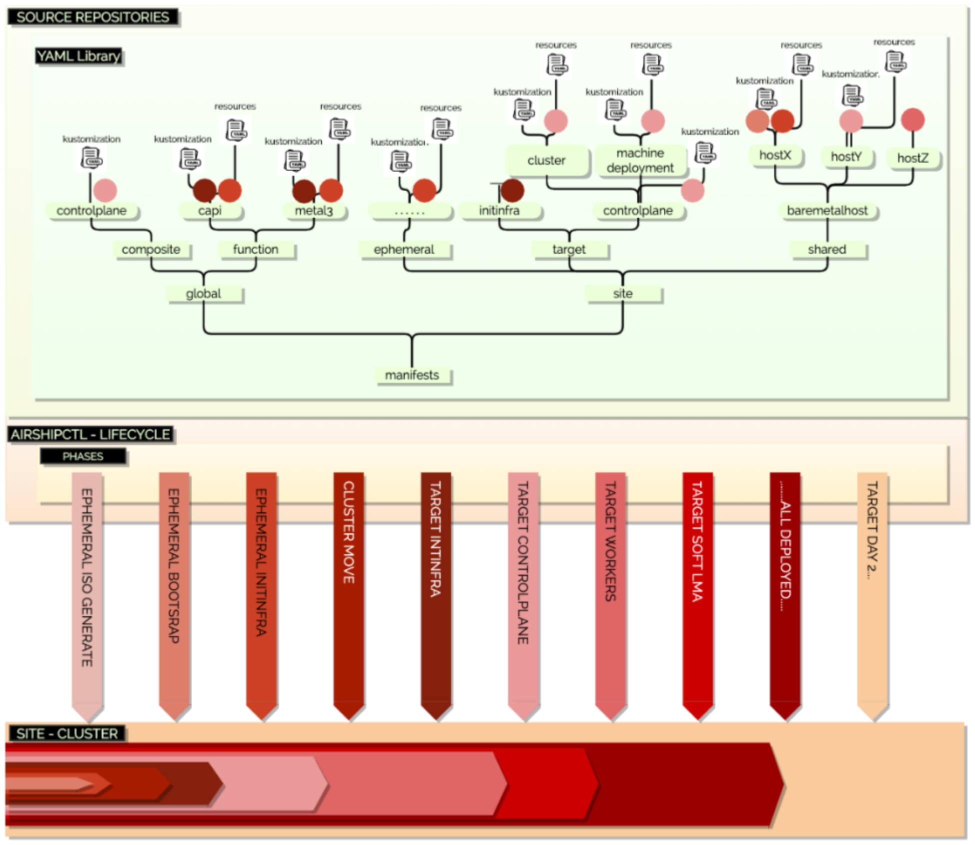

Figure 4 – An Airship Phase

This concept can best be imagined as breaking the resources you need to deliver to Kubernetes into distinct and isolatable units; these units can be a mix of custom resources for that phase and re-usable resources across phases.

Under the hood, Airship leverages Kustomize to render the resources for a given phase. This provides every phase with the power of variable replacement, layering, substitution, resource manipulation, and validation that is built into the CNCF Kustomize project.

From a YAML engineering perspective, this allows Airship users to bring their Kustomization knowledge to the table when working with Airship, and it also means that you can work with and test a given phase in isolation--even without Airship using just *kustomize*--without having to render and examine every resource that *airshipctl* might apply to an environment across all phases.

This makes it much easier to understand where a particular phase is getting its input from especially when dealing with replacements and substitution and what the generated resources look like when being rendered by Kustomize.

A phase is always tied to a specific Kustomize entry point. This is effectively a specific *kustomization.yaml* file in a particular directory.

An Airship *Phase Map* is declared in your YAML library that allows YAML Engineers to expose the available phases they want to publish along with human readable descriptions of what those phases do and provides the link between a phase name and a particular directory containing a *kustomization.yaml.* This allows *airshipctl* to show end-users what phases are available and allow them to run the phase a deployment engineer or CI/CD pipeline wants to target.

As previously described, **phases are essentially a YAML collection which is rendered by Kustomize.**

The structure that Airship provides is that the *kustomization.yaml* for a phase generally points to one or more composites, and any phase specific customizations. Composites themselves are pointers to various function definitions, again with any composite specific customizations.

The point of composites is to be able to define a function once and only once. For instance, it may define how to deploy a workload within Kubernetes, and then re-use that definition without duplicating the YAML in higher level composites that define a business level need by simply re-using these functions.

Another example: the deployment YAML can be defined once for a critical operator as a function, but define several different ways of instantiating that operator as composites. Individual sites can also define as much or as little site-specific phase YAML as they want.

Organizations that deploy extremely homogeneous environments will find that most of their definitions beyond site specific resources like *Bare metalHosts* and *CIDR definitions* can live outside an individual environment's entrypoint and be re-used across all environments but the flexibility exists for an individual environment to be as custom as needed including introducing completely arbitrary phases for that one environment.

In order to provide a curated lifecycle experience, *airshipctl* provides several builtin phases that are available out of the box. These assist users with driving the Cluster-API provisioning process, bootstrapping a control plane, deploying worker nodes, and delivering core infrastructure software such as logging, monitoring, and alerting. Users can choose to leverage these, override them, extend them, or even ignore them. Users can also pull from an upstream YAML library of additional composites and functions the Airship community has curated upstream to build their own phases that more closely align with what they are trying to do with their Kubernetes clusters.

Within the Airship curated phases that are built-in, Airship promotes the operator and CRD pattern which means that all state is stored with Kubernetes and following the progress of operations--from the infrastructure provisioning process to Helm software installation--is as simple as querying the status of objects in Kubernetes with *kubectl*. This pattern ensures that Kubernetes maintains an easy to query state about the progress of any given deployment effort with a low barrier to entry for users and maintains complete ownership of reconciling changes instead of any outside automation. This also ensures that extensions, custom tooling, and other systems that Airship users build on top that coalesce, monitor, and otherwise depend on the status of Kubernetes objects whether for human or CI/CD tooling consumption can remain completely Airship agnostic. They are simply integrating directly with the CNCF projects that Airship employs under the hood.

Airship and Bare Metal

Airshipctl leverages the Cluster-API to support bootstrapping and life cycle management of Kubernetes clusters. This means that any infrastructure back-end supported by the Cluster-API will be supported by Airship.

While there is no assumption of bare metal infrastructure within Airship 2.0, there are a number of pain points when using the Cluster-API to provision bare metal infrastructure that *airshipctl* addresses.

The airshipctl utility provides a specific set of functionality that helps with the additional burden of deploying and managing Kubernetes clouds directly on bare metal and standing up new bare metal environments using the Cluster-API.

When performing a Greenfield bare metal site deployment, *airshipctl* delivers an ephemeral Kubernetes cluster within the environment so that there is a foothold within the physical bare metal environment allowing standard services such as DHCP, PXE, and TFTP to function normally but still be delivered as software deployed via Kubernetes to establish a target control plane. This allows the standard Cluster-API process of provisioning to be performed without forcing users to build their own ephemeral cluster or build out a centralized management cluster that can remotely provision all environments.

The logic behind this approach is to effectively be able to provide a bare metal cluster the same kind of ephemeral management cluster as you could easily fabricate locally with *minikube* or *kubernetes-in-docker (kind)* when working with a third-party cloud but can be difficult to create for bare metal environments.

This functionality is enabled by first by exposing an *`airshipctl bare metal ephemeral isogen`* command. This fabricates an ISO that is tailor made for the target environment using the declared document set for that site. It functions by calling a user-defined container that adheres to a specific contract of inputs and outputs. Namely, it must accept certain environment variables that instruct the ephemeral host on how to configure its networking and first-boot user-data commands to construct the ephemeral Kubernetes cluster and must output specific artifacts such as the ISO itself. The container itself is responsible for manufacturing the bootable ISO and ensuring the cloud-init user-data provided is launched at boot within the ISO without any interactive input by the user.

This containerized approach serves two purposes:

> *First,* building bootable ISOs requires a number of target operating system specific utilities (e.g. debootstrap for Debian) to not only manufacture an ISO but support injecting things like cloud-init into the ISO artifact so that it can come up running Kubernetes without any interaction. It would be restricting to insist that the underlying machine running *airshipctl* have all of these utilities installed directly.
>
> *Second,* building this directly into *airshipctl* means that supporting alternative approaches to ISO building and many operating systems would become challenging. Instead, Airship provides a reference Debian based container upstream that can be used out of the box, but the approach allows users of any operating system or wishing to adopt an ISO building approach that makes sense for their particular organization to use this reference to build their own container that can easily be called instead.

As a reminder, several things are input into the ISO generating container via environment variables when *airshipctl* launches the container. The two primary elements are (1) the network configuration for the ephemeral bare metal host, and (2) the cloud-init user-data that will launch an in-memory Kubernetes instance on the host.

Airship locates these two pieces of data using special labels within the Airship document bundle. It extracts the networking configuration for the ephemeral host from the *Bare metalHost* object that is labeled as the ephemeral host. This enables the user to simply elect to use a host within your declared set of bare metal target infrastructure, without providing a separate ephemeral host or network configuration.

Airshipctl will extract the ephemeral hosts target network configuration that would be used when that host is properly deployed later in the process, and use that same configuration within the ISO, reducing the need for a separate configuration to maintain and allowing users to easily change which host is used as the ephemeral host by moving a label. *Airshipctl* will also extract the ephemeral cloud-init user-data from a *Secret* object within your document bundle that has an ephemeral user-data label.

Once the ISO is generated, airshipctl can again be invoked via *`airshipctl baremetal ephemeral remotedirect`* to force the ephemeral host to remotely boot from the ISO generated in the previous step which will boot into an in-memory single-node Kubernetes cluster, Airshipctl uses this to instantiate the dependencies necessary to drive subsequent Cluster-API based provisioning within that environment and build out a permanent target cluster.

When at least one control plane node is provisioned within the target cluster, *airshipctl* will pivot the provisioning process into the target cluster, and continue to deploy bare metal hosts from there, abandoning the ephemeral host and eventually absorb the ephemeral node into the target cluster when it is targeted for provisioning.

This ensures every host, even the first host, are provisioned exactly the same way at the end of the process.

Comparing Airship to the Industry Alternatives

When it comes to Kubernetes lifecycle management, there are only a handful of realistic off-the-shelf options to build and manage Kubernetes environments.

These include OpenShift, Rancher, and Pivotal PKS. All of these are mature and widely used products.

There are several reasons we believe Airship 2.0 provides a better path, especially for Telco providers:

- **Telco providers today are heavily invested in VNFs and OpenStack today.** Airship excels at delivering Helm based software and so a pairing of Airship, and the upstream OpenStack-Helm project to deploy OpenStack via Helm on top of Kubernetes allows Telco's a way to deliver a cloud-native solution that aligns with where they are today, but enables their path to the future when workloads are entirely container-based. We provide many of the same core features as the platforms above, but in a way that helps enable Telco CNF transformation. This makes it easy for them to build hybrid VNF and CNF use cases that run on bare metal and public clouds right now.
- **None of the solutions above are purpose-built for Telco needs.** While Airship can be used for any type of workload on top of Kubernetes and is not built specifically for Telco's, we have a set of curated functions and composite YAML definitions that can be leveraged to help enable Telco-grade security, networking, and topology configurations to support things such as CPU and NUMA Pinning, Multus for multiple interfaces, SR-IOV, and so on which is simply not an offering from most of the products above.
- **Airship is highly decomposable.** As discussed above, Airship delivers infrastructure provisioning and lifecycle management as a series of discrete phases. End-users can choose to use as much or as little of Airship as they want. For instance, you could use Airship to simply provision your infrastructure and Kubernetes, but switch to your own automation for anything beyond a core Kubernetes control plane.
- **Operating System Agnostic.** Airship is operating system agnostic for any infrastructure backend that is supported by the Cluster-API as long as that operating system can run cloud-init. Most vendor products have an extremely tight coupling to their own operating system without any real option to support alternatives.

- **Truly No Lock-In.** Probably one of the most important features of Airship is that there is truly no lock-in. As detailed in this paper, Airship orchestrates best-of-breed CNCF projects to enable the provisioning and life-cycle of Kubernetes clusters. If you want to stop using Airship to manage your environments and manage the underlying CNCF components using your own tooling you can do so at any time. There is no customization or Airship forked versions, so you can switch to managing them yourself easily.
- **Open Approach to Vendor Support.** Airship is not a vendor product--it is built by operators for operators--so a variety of existing and established support vendors you are already working with can easily step-in to a production support role for your organization.

In summary …

Summary

Airship solves the operational challenge of integrating and managing the lifecycle of the various CNCF projects that enable end-to-end provisioning and maintenance of clouds.

Airship is effectively a proven approach and leveraged at scale in production today. The fact that end-users could technically stitch together these various CNCF components using their own in-house automation is evidence of the sustainability of this approach.

There is no intention for Airship to step outside the role of simply being the glue that helps streamline the integration of various CNCF projects and the resulting operational experience. When the Airship community identifies enhancements or issues with the upstream projects it works with, we seek to resolve those upstream and consume then consume the new functionality within Airship rather than own those solutions within the Airship project.

We believe this is what a true Open Source ecosystem looks like and how Airship can thrive in an ever-evolving landscape.

References

Airship 2.0 Intro Video: https://www.youtube.com/watch?v=13v3z4EIK9I

Airshipctl: https://github.com/airshipit/airshipctl

Airship UI: https://github.com/airshipit/airshipui

Airship 2.0 Technical Blog Series:

- https://www.airshipit.org/blog/airship-blog-series-1-evolution-towards-2.0/
- https://www.airshipit.org/blog/airship-blog-series-2-an-educated-evolution/
- https://www.airshipit.org/blog/airship-blog-series-3-airship-2.0-architecture-high-level/
- https://www.airshipit.org/blog/airship-blog-series-4-shipyard-an-evolution-of-the-front-door/
- https://www.airshipit.org/blog/airship-blog-series-5-drydock-and-its-relationship-to-cluster-api/
- https://www.airshipit.org/blog/airship-blog-series-6-armada-growing-pains/

Airship YouTube Channel (Demos etc.):
https://www.youtube.com/playlist?list=PLKqaoAnDyfgp8YjZbzjVrmZBJR9thV27y

Airship 1.0 White Paper: PDF Document