Airship Project Documentation

Airship Team

Dec 06, 2021

LEARN ABOUT AIRSHIP 2

Airship is a collection of components that declaratively configure, deploy and maintain a Kubernetes environment defined by YAML documents. Airship is supported by the OpenStack Foundation.

CHAPTER

ONE

ABOUT THIS DOCUMENTATION

Airship documentation serves the entire community with resources for users and developers.

1.1 Airship 2 Basics

1.1.1 What is Airship 2 (AS2)?

Airship is a collection of interoperable open source cloud provisioning tools that provide a robust delivery mechanism for organizations that want to embrace containers as the new unit of infrastructure delivery at scale. Starting with bare metal infrastructure, Airship manages the full infrastructure life cycle to deliver and manage a Kubernetes cluster, with Helm deployed artifacts, for development and production. As the version implies, Airship 2 (AS2) is the second major release of the Airship tool set.

1.1.2 Why did Airship come into existence?

The use of open source components for cloud deployments provide many benefits:

- Open source software is often free.
- Communities of users, developers, and contributors develop cloud functionality based on practical needs and use cases requiring cloud platform technology.
- Services for orchestration of compute, network, and storage, as well as fault management are open source driven and supported.

The challenges, however, are the technical aptitude, maturity, and experience required to integrate and maintain these open source components to provide a complex cloud platform to run client applications. **Airship was born** from the need to streamline and simplify the use of open source tools for deployment and management of cloud infrastructure.

1.1.3 What does Airship 2 do?

Airship allows operators an ability to deploy complex Kubernetes-based cloud infrastructure and manage its infrastructure life cycle. Airship applies principles of application management to cloud infrastructure operation. Hard assets, such as bare metal servers and network settings, are managed alongside soft assets, such as the Helm charts and application containers.

1.1.4 How does Airship 2 (AS2) differ from Airship 1 (AS1)?

The evolution of Airship 2 includes the following improvements from Airship 1:

- Smaller, ephemeral footprint
- Adoption of established upstream Cloud Native Computing Foundation (CNCF) projects
- Expanded support for multiple platforms
- Less downstream customization of manifests
- Introduction of phases & phase plans
- Airship UI is planned and will enhance the user experience
- Improved speed of deployment vs. Airship 1.0
- Airship in a Pod (AIAP) demonstrates improved onboarding experience gained through AS2 capabilities and functionality

1.1.5 How does AS2 work?

Airship has a single workflow for managing both initial installations and updates through declarative YAML documents describing an Airship target environment. These YAML documents provide for declaration of the entire infrastructure up front. This declarative approach allows Airship to provide a repeatable and predictable mechanism for maintaining service-critical infrastructure and the software running there.

An operator only needs to make a change to an Airship YAML configuration when defining or changing cloud infrastructure based on the deployment need, and Airship performs the heavy-lifting automatically completing the needed work. When managing complex Infrastructure-as-a-Service (IaaS) projects, anything from minor service configuration updates to major upgrades are all handled in the same way: by simply modifying the YAML configuration and submitting it to the Airship runtime.

1.1.6 Benefits of AS2

Platform Integration: Airship deploys an integrated virtualization and containerization platform, utilizing Kubernetes and Helm. Airship deploys and manages the infrastructure, platform components and services. Airship is a resilient application deployment and life cycle engine that functions with any Helm chart based application.

Security at Scale: Airship 2 leverages native security tools and services such as Secure Computing (seccomp), Mandatory Access Controls (MAC), and Pod Security Policies (PSP), across the entire toolchain for delivery, development, and management of cloud based software and infrastructure. These tools coupled with Transport Layer Security (TLS)-enabled service endpoints and encrypted storage of secrets ensure a secure platform for system calls, file access, and application controls.

Scalable Operations: By leveraging Kubernetes and Helm, critical services can automatically scale under load and can robustly survive hardware failure. The deployed platform can easily bundle many tools required to operate the cloud infrastructure including, for example, leveraging OpenStack-Helm, network security policies, or tools such as for log collection, search capabilities, monitoring, alerting, and graphing.

Reliable Upgrades: Critical services can be upgraded with confidence, with gradual roll-outs (including the ability to roll-back), and guaranteed data and virtual machine integrity across container application upgrades, with minimal (or no) need to shut down services or live-migrate virtual machines through the upgrade process.

1.2 Release Notes

1.2.1 Airship Overview

Airship is a robust system for delivering container-based cloud infrastructure (or any other containerized workload) at scale on bare metal, public clouds, and edge clouds. Airship integrates best-of-class CNCF projects, such as Cluster API, Kustomize, Metal3, and Helm Operator, to deliver a resilient and predictable lifecycle experience.

Combining easy lifecycle management with zero-downtime real-time upgrade capability, Airship can handle the provisioning and configuration of the operating system, RAID services, and the network.

1.2.2 Airship 2.1 (30 November 2021)

Release 2.1 introduces the following enhancements:

- Upgrade components to parity with Cluster API v1alpha4, including Bare Metal Operator v1alpha5
- Kubernetes upgrade to version 1.21
- Docker provider upgrade to v1alpha3
- CAPD and CAPZ upgrades to versions 0.4.2 and 0.5.2, respectively
- Airship in a Pod hardening and improvements such as support for custom site manifest locations and private repositories. (477)
- Helm Controller upgrade to version 0.11.1 and Source Controller to version 0.15.3 (607)
- Kustomize upgrade to version 4.2.0
- KPT upgrade to to version 1.0.0-beta.7
- Support of iLO5 in bare metal node bootstrapping

1.2.3 Airship 2.0 (16 April 2021)

Release 2.0 introduces a variety of significant improvements:

- No-touch bootstrap for remote sites as well as local sites
- Declarative image building for both ephemeral ISO and bare metal targeted QCOWs
- Declarative cluster lifecycle
- Lifecycle for bare metal, public cloud, and edge cloud infrastructure
- Single command line “airshipctl”
- Lifecycle defined as a sequence of phases
- Introduction of a plan for the phases
- Seamless integration with CNCF projects (CAPI, Metal3, Kustomize)
- Seamless integration with security plugins like SOPS
- Generic container interface: mechanism to extend airshipctl with ad hoc functionality
- Introduction of host config operator for day 2 operations

1.2.4 Change Log

Change logs list feature and defect details for a release. For the complete set of releases including links to change logs, see the treasuremap and airshipctl Github release pages.

1.3 Airship Security Vulnerability Management

The Airship community is committed to expediently confirming, resolving, and disclosing all reported security vulnerabilities. We appreciate your cooperation and participation in our vulnerability management process outlined below.

1.3.1 Report a Vulnerability

If you discover a vulnerability in an Airship project, please treat the issue with a sense of confidentiality and disclose it to the airship-security mailing list:

airship-security@lists.airshipit.org

Additionally, please include any potential fixes, as doing so can expedite the disclosure and patching processes.

The Airship Working Committee is the sole subscriber of the airship-security mailing list and monitors it for reported vulnerabilities. The committee confirms or rejects reported vulnerabilities in correspondence with the vulnerability reporter. In the event that the Airship Working Committee does not have the expertise or availability to resolve a reported vulnerability, the committee may solicit assistance from outside contributors to better facilitate the understanding and resolution of reported security vulnerabilities.

1.3.2 Receive Early Disclosures

We prefer to disclose confirmed security vulnerabilities as soon as possible. While circumstances may not always allow immediate disclosure, vulnerabilities may be disclosed over the airship-embargo-notice mailing list when a fix becomes available. The airship-embargo-notice mailing list notifies Airship users of confirmed vulnerabilities. If you operate Airship in a production environment, we recommend subscribing to the airship-embargo-notice mailing list by contacting the Airship Working Committee. The Airship Working Committee evaluates subscription requests on a case-by-case basis.

1.3.3 Receive Public Disclosures

Within ninety days of the initial vulnerability report, except in unusual circumstances, the Airship Working Committee will publicly disclose the reported vulnerability and its mitigation over the airship-announce and airship-discuss mailing lists. If a fix merges before the aforementioned ninety day period expires, the Airship Working Committee will instead disclose the vulnerability and fix twenty-one days later. We recommend subscribing to both mailing lists in order to receive security updates.

1.4 Layering and Deduplication

1.4.1 Airship Layering Structure

Airship addresses the DRY (Don't Repeat Yourself) principle by promoting the use of conceptual "layers" of declarative intent. Some configuration is completely generic and reusable - for example, the basic Pod, Deployment, and

HelmRelease resources that make up a Kubernetes application deployment. Other information will be fully site-specific - e.g., the IP addresses and hostnames unique to a site. The site-specific information typically needs to be injected into the generic resources, which can be conceptualized as overlaying the site-specific layer on top of the generic, reusable base. Although any number of layers could be applied in arbitrary ways, Airship has chosen the following conceptual layer types to drive consistency and reusability:

- Functions: basic, independent building blocks of a Kubernetes workload. Typical examples include a single HelmRelease or an SDN.
- Composites: Integrations of multiple Functions, integrating/configuring them for a common purpose. Typical examples include OpenStack or interrelated logging and monitoring components. Composites can also pull in other composites to further customize their contents.
- Types: Prototypical deployment plans. A Type defines composites into a deployable stack, including control plane, workload, and host definition functions. A Type will define Phases of deployment, which serve to sequence a deployment (or upgrade) into stages that must be performed sequentially - e.g., the Kubernetes cluster must be created before a software workload is deployed into it. A type can pull in any number of Composites, and when appropriate can inherit from exactly one other Type. Types typically represent "use cases" such as a Network Cloud, a CI/CD system, or a basic Kubernetes deployment.
- Sites: A Site is a realization of exactly one Type, and defines (only) the site-specific configuration necessary to deploy the Type to a specific place.

The layers above are simply conventions. In practice, each layer is represented by a Kustomization (see *Kustomize Overview* below), and relationships between layers are captured by a `kustomization.yaml` referencing an external Kustomization. Additionally, Airship layers may refer to layers in other code repositories, and Airship will ensure all required projects are present and in expected locations relative to a site definition. This allows for operator-specific (upstream or downstream) repositories to inherit and reuse the bulk of declarative intent from common, upstream sources; e.g. the Airship Treasuremap project. By convention, Airship manifests can be found in a project's `manifests/` folder, further categorized by the layer type – e.g., `manifests/function/my-chart` or `manifests/site/my-site`.

TODO: add pictures

The fundamental mechanism that Kustomize provides for layering is "patching", which operates on a specific YAML resource and changes individual elements or full YAML subtrees within the resources. Kustomize supports both JSON patching and Strategic Merge patching; see Kustomize Patching for more details.

1.4.2 Kustomize Overview

Airship uses the Kustomize tool to help organize, reuse, and deduplicate declarative intent. Kustomize has widespread usage in the Kubernetes community, and provides a standalone command-line interface that lets a user start with "generic" manifests in one location, and then apply overrides from a different location. This can be used to perform operator- or site-specific customization (hence the name) of resource namespaces, patches against arbitrary YAML keys, and many other features.

Kustomize also provides a Go library interface to drive its functionality, which is used by projects such as `kubectl`, `kpt`, and `airshipctl`. Airshipctl incorporates Kustomize into its higher-order functionality, so that it's invoked as part of `airshipctl phase run`, rendering manifests into a deployable form before deploying them directly to a Kubernetes api server. However, Airship follows the design decision for its YAMLs to be fully renderable using the stock `kustomize` command.

The basic building block for Kustomize manifests is called a "kustomization", which typically consists of a directory folder containing Kubernetes resources, patches against them, and other information; and the file `kustomization.yaml`, which acts as a roadmap to Kustomize on what to do with them. A `kustomization.yaml` can also point out to other kustomization directories (each with their own `kustomization.yaml`) to pull them in as base resources, resulting in a tree structure. In an Airship context, the root of the tree would represent an Airship deployment phase, and the leaves would be functions that capture something fine-grained, like a HelmRelease chart resource.

1.4.3 Plugins

Airship extends Kustomize via plugins to perform certain activities which must be done in the middle of rendering documents (as opposed to before or after rendering). The plugins are implemented as containerized "KRM Functions", a standard which originated in the KPT project community. Kustomize invokes these plugins to create or modify YAML resources when instructed to via `kustomization.yaml`. The KRM Functions that are leveraged by airshipctl include:

ReplacementTransformer

Kustomize is very good at layering patches on top of resources, but has some weakness when performing substitution/replacement type operations, i.e., taking information from one document and injecting it into another document. This operation is very critical to Airship from a deduplication perspective, since the same information (e.g. a Kubernetes version or some networking information) would be needed in multiple resources. Replacement helps satisfy the DRY principle. Although Kustomize has a few flavors of variable replacement-type functionality, a plugin was required to consistently support simple replacement, injection of YAML trees into arbitrary paths of a target document, and substring replacement.

Airship's ReplacementTransformer function is based on the example plugin of the same name from the Kustomize project, and extends it with substring replacement, improved error handling, and other functionality.

The ReplacementTransformer is invoked by Kustomize when a transformer plugin configuration with `kind: ReplacementTransformer` and `apiVersion: airshipit.org/v1alpha1` is referenced in a `kustomization.yaml`. The plugin configuration payload contains a list of replacement instructions, each with "source" and "destination" definitions.

See the Replacement Transformer documentation for examples and usage.

Templater

While ReplacementTransformer modifies existing resources, the Templater is a Kustomize "generator" plugin that creates brand new resources based on a Go Template. This is helpful when you have a number of resources that are nearly identical, and allows the common parts to be deduplicated into a template. An example of this would be resources that are specific per-host (like Metal3 `BareMetalHost`).

The ReplacementTransformer and Templater can also be combined in a chain, with the ReplacementTransformer injecting information into a Templater plugin configuration (for example, the number and configuration of `BareMetalHost` resources to generate), so that it has all of the information it needs to generate resources for a particular site.

See the Templater documentation for examples and usage.

Encryption, Decryption, and Secret Generation

Encryption and decryption is handled via the *sops* KRM Function maintained by the KPT community. This function in turn uses the Mozilla SOPS tool to perform asymmetric key-based decryption on encrypted values within YAML resources.

In addition, passwords, keys, and other sensitive secrets can be generated from scratch on a site-by-site basis. This helps ensure that secrets are properly randomized and are unique to particular deployments. This process uses the Templater function to describe what secrets should be generated, and the SOPS plugin to encrypt them immediately, before being written to disk.

See the *Secrets generation and encryption* guide for examples and usage.

1.4.4 Replacement

The resource patching functionality that Kustomize provides out-of-box solves for many config deduplication needs, particularly when an operator wants to define some override YAML and then apply it on top of exactly one resource. However, it does not solve for the case where some piece of configuration, e.g. a Kubernetes version number, needs to be applied against multiple resources that need it. Using patches alone, this would result in duplicating the same version number defined in multiple patches within the same layer, increasing both maintenance effort and opportunity for human error. To address this, Airship uses the ReplacementTransformer plugin described above.

The ReplacementTransformer can inject individual YAML elements or trees of YAML structure into a specific path of a target resource. The source for the injected YAML can come from any arbitrary resource. For the sake of maintainability and readability, Airship uses "variable catalogue" resources as replacement sources. These catalogues group related configuration together with an expected document name and YAML structure, and their sole purpose is to serve as a replacement source. Today, these catalogues are a mix of free-form `kind: VariableCatalogue`, and well-schema'd per-catalogue kinds. The plan is to migrate all of them to well-defined schemas over time.

In general, Airship defines three things: good default configuration at the Function level, default/example catalogues that also contain (typically the same) default values, and ReplacementTransformer configuration that copies data from one to the other. This conforms to the Kustomize philosophy of base resources being "complete" in and of themselves. In practice, we encourage operators to supply their own catalogues, rather than basing on the upstream default/example catalogues.

Versions Catalogues

Software versioning is frequently an example of information that should be defined once, and be consumed in multiple locations. However, a more compelling reason to pool versioning information together into a single source catalogue, even when it will be consumed by exactly one target document, is simply to put it all in the same place. Versioning information also includes definition of registries and repositories, so by defining all of your versions in one place, it becomes more straight forward to change them all at once. For example, an operator may choose to pull all Docker containers from a downstream container registry instead of the default upstream registry. Another example would be upgrading the versions of a number of interrelated software components at the same time (e.g. a new OpenStack release). On the other hand, a monolithic versions catalog would run the risk of coupling unrelated software components together.

Airship balances these concerns by typically defining one versions catalogue per manifest repository (e.g. `airshipctl`, `treasuremap`, `openstack-helm`), with a naming convention of `versions-<repo-name>`. This keeps the definition of default versions close to the Functions that define them and the Composites that integrate them, and avoids cross-repo dependency concerns. Within their home repository, the base catalogue should live in a standalone Function (called something like `catalogues-<repo-name>`, so that it can easily be swapped out for alternate version definitions. The per-repository catalogues all share the `kind: VersionsCatalogue` schema, which is defined in the `airshipctl` repository.

The definition of the replacement "rules", captured in the ReplacementTransformer plugin configuration, will typically be done at the Composite level as part of the integration work for which they're responsible. Those Composites should each feature a README that details any catalogues needed to render them.

Note that versioning for Cluster API (CAPI) providers will be handled slightly differently, for a couple reasons:

1. CAPI provider versions are a part of the directory structure (e.g. `manifests/function/capm3/v0.3.1/`, which are also closely coupled to airshipctl itself. So, versioning of CAPI components is out-of-scope for a catalogue-driven perspective, and is left to the airshipctl project.
2. CAPI container versions/locations must be handled differently, because those resources are not pulled in via the same phase-driven approach as everything else, and are instead referenced indirectly via a `Clusterctl` resource (which is a normal, phase-included resource). Since this `Clusterctl` config can pass variables into the clusterctl rendering process, the solution will be: first, use the ReplacementTransformer plugin to substitute

container versions from the `versions-airshipctl` catalogue into the `Clusterctl` resource; then, let clusterctl itself push those values into the appropriate CAPI resources via its variable replacement functionality.

3. To perform upgrades of CAPI components, two sets of versions, the old and the new, will need to be substituted into the `Clusterctl` resource simultaneously. Therefore, their version catalogue will need to contain multiple sets of container versions, using the provider version number as part of the YAML path (just like it's part of the directory path above).

Similarly to container versioning, `HelmRelease` resources will refer out to Helm charts by location and version. This is still being defined as of this writing, but in production use cases at least, Airship will use a "Helm Collator" that will cache built charts in a single container image and then serve the charts within the cluster. We may also support deploying charts dynamically from github (as in Airship 1) if the Helm Controller continues to support that feature. In any case, the plan is for the chart locations/versions to be encoded in a versions catalogue(s), and for that to drive the collator build process and/or live chart rendering.

Note that the trigger gets pulled on replacement only at the Site/Phase level; one can apply patches to catalogues till then. For example, if a catalogue is defined at the Type level with normal default values, the values can be overridden in the catalogue resource at the Site level before the catalogue is actually used.

TODO: include a lifecycle diagram that shows documents and replacements getting aggregated, and then ultimately executed at site level.

Note that Kubernetes resource versions are a different animal, and are not addressed via catalogue replacement.

Host Catalogue and Host Generation Catalogue

TODO

Network Catalogue

Networking is another example of a set of values that all change at once: on an operator-by-operator basis (for shared network services like DNS), and on a site-by-site basis (for subnet IP address ranges). This information is extracted into a VariableCatalogue with `name: networks`. Individual functions that consume the information will provide their own replacement rules to do so.

A default/example set of values is defined in the `airshipctl-catalogues` function, and it can be patched (or duplicated) at the Type or Site levels to apply operator- and site-specific information, respectively.

Note that per-host IP addresses are generally specified in the Host Catalogue rather than the Network Catalogue.

Today, the Network Catalogue is specific to Functions in the `airshipctl` repository, which defines a `kind: NetworkCatalogue` schema for it.

Endpoint Catalogues

For internal cluster endpoints that are expected to be generally the same between use cases, operator-specific types, and sites, there may be no need to externalize the endpoint into the catalogue. In those cases, an operator may still choose override the endpoint via a Kustomize patch.

TODO

1.4.5 Phases

TODO

1.4.6 Treasuremap

TODO

1.5 Airship Glossary of Terms

Airship A platform that integrates a collection of best-of-class, interoperable and loosely coupled CNCF projects such as Cluster API, Kustomize, Metal3, and Helm Operator. The goal of Airship is to deliver automated and resilient container-based cloud infrastructure provisioning at scale on both bare metal and public cloud, and life cycle management experience in a completely declarative and predictable way.

Bare metal provisioning The process of installing a specified operating system (OS) on bare metal host hardware. Building on open source bare metal provisioning tools such as OpenStack Ironic, Metal3.io provides a Kubernetes native API for managing bare metal hosts and integrates with the Cluster-API as an infrastructure provider for Cluster-API Machine objects.

Cloud A platform that provides a standard set of interfaces for IaaS consumers.

Container orchestration platform Set of tools that any organization that operates at scale will need.

Control Plane From the point of view of the cloud service provider, the control plane refers to the set of resources (hardware, network, storage, etc.) configured to provide cloud services for customers.

Data Plane From the point of view of the cloud service provider, the data plane is the set of resources (hardware, network, storage, etc.) configured to run consumer workloads. When used in Airship deployment, "data plane" refers to the data plane of the tenant clusters.

Executor A usable executor is a combination of an executor YAML definition as well as an executor implementation that adheres to the `Executor` interface. A phase uses an executor by referencing the definition. The executor purpose is to do something useful with the rendered document set. Built-in executors include KubernetesApply, GenericContainer, Clusterctl, and several other executors that helps with driving Redfish, bare metal node image creation.

Hardware Profile A hardware profile is a standard way of configuring a bare metal server, including RAID configuration and BIOS settings. An example hardware profile can be found in the Airshipctl repository.

Helm Helm is a package manager for Kubernetes. Helm Charts help you define, install, and upgrade Kubernetes applications.

Kubernetes An open-source container-orchestration system for automating application deployment, scaling, and management.

Lifecycle management The process of managing the entire lifecycle of a product from inception, through engineering design and manufacture, to service and disposal of manufactured products.

Network function virtualization infrastructure (NFVi) Network architecture concept that uses the technologies of IT virtualization to virtualize entire classes of network node functions into building blocks that may connect, or chain together, to create communication services.

Openstack Ironic (OpenStack bare metal provisioning) An integrated OpenStack program which aims to provision bare metal machines instead of virtual machines, forked from the Nova bare metal driver.

Orchestration Automated configuration, coordination, and management of computer systems and software.

Phase Phase is defined as a Kustomize entrypoint and its relationship to a known Airship executor that takes the rendered document set and performs defined action on it. The goal of phases is to break up the delivery of artifacts into independent document sets. The most common example of such an executor is the built-in `KubernetesApply` executor which takes the rendered document set and applies it to a Kubernetes end-point

and optionally waits for the workloads to be in a specific state. The airship community provides a set of predefined phases in Treasuremap that allow you to deploy a Kubernetes cluster and manage its workloads. But you can craft your own phases as well.

Phase Plan A plan is a collection of phases that should be executed in sequential order. It provides the mechanism to easily orchestrate a number of phases. The purpose of the plan is to help achieve a complete end to end lifecycle with a single command. Airship Phase Plan is declared in your YAML library. There can be multiple plans, for instance, a plan defined for initial deployment, a plan for updates, and even plans for highly specific purposes. Plans can also share phases, which makes them another fairly light-weight construct and allows YAML engineers to craft any number of specific plans without duplicating plan definitions.

Software defined networking (SDN) Software-defined networking technology is an approach to network management that enables dynamic, programmatically efficient network configuration in order to improve network performance and monitoring making it more like cloud computing than traditional network management.

Stage A stage is a logical grouping of phases articulating a common purpose in the life cycle. There is no airshipctl command that relates to stages, but it is a useful notion for purposes of discussion that we define each of the stages that make-up the life cycle process.

1.6 Deploying A Bare Metal Cluster

The instructions for standing up a greenfield bare metal site can be broken down into three high-level activities:

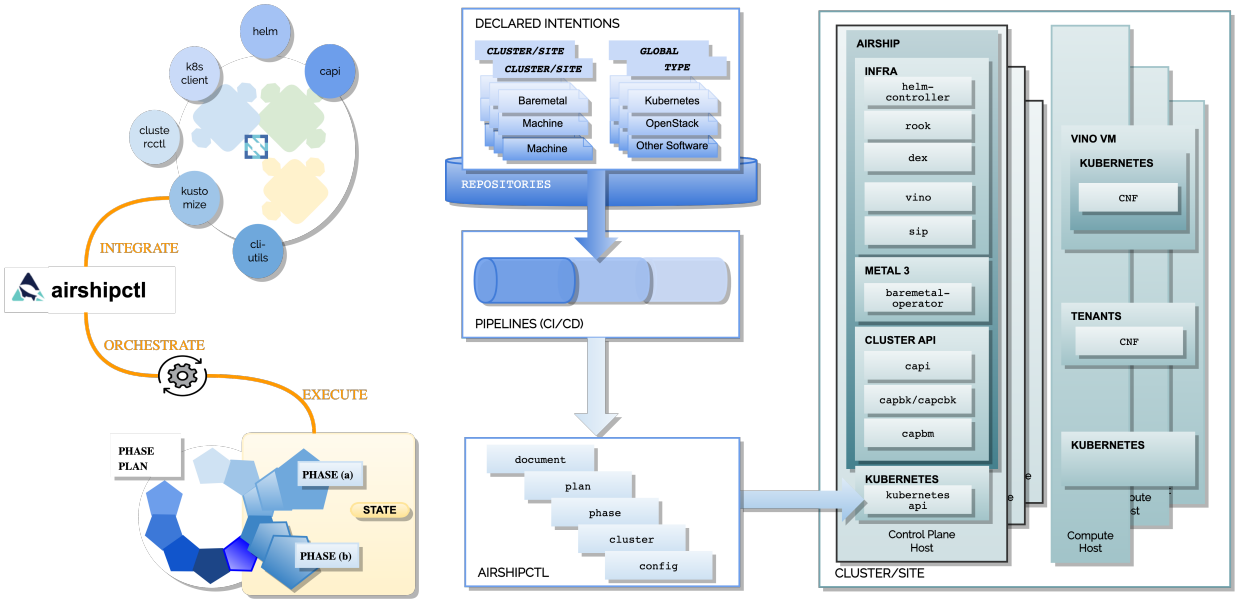
1. *System Requirement and Setup*: Contains the hardware and network requirement and configuration, and the instructions to set up the build environment.
2. *Site Authoring Guide*: Describes how to craft site manifests and configs required for a site deployment performed by Airship.
3. *Site Deployment Guide*: Describes how to deploy the site utilizing the manifests created as per the *Site Authoring Guide*.

1.6.1 System Requirement and Setup

Component Overview

Airship uses a command line utility airshipctl that drives the deployment and life cycling management of Kubernetes clouds and software stacks.

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



Node Overview

This document refers to several types of nodes, which vary in their purpose, and to some degree in their orchestration / setup:

- **Build node**: This refers to the environment where configuration documents are built for your environment (e.g., your laptop).
- **Ephemeral node**: The "ephemeral" or "seed node" refers to a node used to get a new deployment off the ground, and is the first node built in a new deployment environment.
- **Controller nodes**: The nodes that make up the control plane. (Note that the ephemeral node will be converted to one of the controller nodes).
- **Worker nodes**: The nodes that make up the data plane.

Hardware Preparation

The Treasuremap reference-airship-core site shows a production-worthy bare metal deployment that includes multiple disks and redundant/bonded network configuration.

Note: Airship hardware requirements are flexible, and the system can be deployed with very minimal requirements if needed (e.g., single disk, single network).

For simplified non-bonded, and single disk examples, see Airshipctl test-site.

BIOS, Redfish and PXE

1. Ensure that virtualization is enabled in BIOS.
2. Ensure that Redfish IPs assigned, and routed to the environment you will deploy into. Firmware bugs related to Redfish are common. Ensure you are running the latest firmware version for your hardware.

3. Set PXE as first boot device and ensure the correct NIC is selected for PXE.

Note:

- Airship can remotely bootstrap the nodes using Redfish. If Redfish is not available, you can mount the ephemeral ISO image via an alternate mechanism such as USB thumb drive.
- Airship 2 has been verified on Dell PowerEdge R740xd servers with iDRAC 9, BIOS Version 2.8.2, iDRAC Firmware Version 4.22.00.53 and Redfish API version 1.

Disk

1. For controller nodes including the ephemeral node:

- Two-disk RAID-1: Operating System

2. For worker nodes (tenant data plane):
    - Two-disk RAID-1: Operating System
    - Remaining disks: configuration per worker host profile
3. For nodes in the storage cluster:

    After the allocation of disks for the host OS and other uses, remaining disks can be configured as RAID-0/JBOD for Ceph. If both SSD and HDD disks are present, it is recommended to configure two Ceph clusters, one for each media technology. The number of storage disks and capacity per disk should be determined by the need of the workloads.

Network

1. Ensure that you have a dedicated PXE interface on untagged/native VLAN. 1x1G interface is recommended. The PXE network must have routability to the internet in order to fetch the provisioning disk image; alternately, you may host the image locally on the PXE network itself.
2. Ensure that you have VLAN segmented networks on all nodes. 2x25G bonded interfaces are recommended.

The table below is an opinionated example used by Treasuremap reference site `reference-airship-core`, but users can diverge from it as needed. For example, in the simplest configuration, two networks can be configured: one for PXE and one for everything else.

VLAN/ Network	Name	Routability	Quantity	MTU	Description
1023	OOB/iLO	WAN	IPv4:/26 IPv6:/64	1500	For HW Redfish addressing
eno4	PXE	Private RFC1918	IPv4:/25 IPv6:/64	1500	For bootstrap by Ironic, Metal3 or MaaS
61	OAM	WAN	IPv4:/26 IPv6:/64	9100	• Used for operational access to Hosts. • Can reach to OOB, PXE, DNS, NTP, Airship images and manifest repos • Hosts all host level endpoints
	OAM	WAN	IPv4:/29	9100	• Rack floating VIP for K8S ingress traffic • Configured as secondary subnet for VLAN 41 • Hosts all service endpoints
62	Storage	Private RFC1918	IPv4:/25 IPv6:/64	9100	Ceph storage traffic for all hosts, pods and VMs
64	Calico	Private RFC1918	IPv4:/25 IPv6:/64	9100	L2 network used by Calico for BGP peering or or IP-in-IP mesh
82	Subcluster Net	Private RFC1918	IPv4:/22 IPv6:/64	9100	Private IP ranges to VM based subclusters for K8S as a service
Private Reserve	CNI Pod Network	Zone Private	IPv4:/16 IPv6:/64	N/A	For Kubernetes Pods

See detailed network configuration example in the Treasuremap repo `manifests/site/reference-airship-core/target/catalogues/networking.yaml` configuration file.

Hardware sizing and minimum requirements

Node	Disk	Memory	CPU
Build (laptop)	10 GB	4 GB	1
Ephemeral/Control	500 GB	64 GB	24
Worker	N/A*	N/A*	N/A*

- Workload driven (determined by host profile)

See detailed hardware configuration in the Treasuremap repo `manifests/site/reference-airship-core/target/catalogues` folder.

Establishing build node

Setting Environment Variables

The Airship deployment tool requires a few environment variables that the operators need to configure on the build node. The environment variables can be persisted by setting them in your profile, or can be set in the shell session before you run the Airship commands and scripts.

Proxy

Access to external resources such as `github`, `quay.io` and `go` is required for downloading manifests, images and `go` packages. If you are behind a proxy server, the following environment variables must be configured on the build node.

- `USE_PROXY`: Boolean value to indicate if the proxy setting should be used or not.
- `http_proxy`: Proxy server for HTTP traffic.
- `https_proxy`: Proxy server for HTTPS traffic.
- `no_proxy`: IP addresses or domain names that shouldn't use the proxy.

SOPS

For security reasons the secrets in the Airship manifests should not be stored in plain-text form. Airshipctl selects Mozilla SOPS to encrypt and decrypt the manifests.

Two environment variables are needed for the encryption and decryption:

- `SOPS_IMPORT_PGP`: Contains public or private key (or set of keys).
- `SOPS_PGP_FP`: Contains a fingerprint of the public key from the list of provided keys in `SOPS_IMPORT_PGP` that will be used for encryption.

The easiest way to generate SOPS keys is to use gpg wizard:

```
gpg --full-generate-key
```

For demo purpose, you can import the pre-generated SOPs keys used by Airshipctl gate:

```
curl -fsSL -o /tmp/key.asc https://raw.githubusercontent.com/mozilla/sops/master/pgp/
↪sops_functional_tests_key.asc
export SOPS_IMPORT_PGP="$(cat /tmp/key.asc)"
export SOPS_PGP_FP="FBC7B9E2A4F9289AC0C1D4843D16CEE4A27381B4"
```

Airship Installation

- `AIRSHIP_CONFIG_MANIFEST_DIRECTORY`: File system path to the Airship manifest directory, which will be the home of all Airship artifacts, including airshipctl, treasuremap, your projects and sites. You can create the directory at a location of your choice.
- `PROJECT`: Name of the project directory to be created in the *Initializing New Site* section.
- `SITE`: Name of the site to be deployed.

Download Airship

1. On the build node, install the Git package:

```
sudo apt update
sudo DEBIAN_FRONTEND=noninteractive apt -y install git
```

2. Create the Airship home directory and clone the `airshipctl` and `treasuremap` repository:

```
mkdir -p $AIRSHIP_CONFIG_MANIFEST_DIRECTORY
cd $AIRSHIP_CONFIG_MANIFEST_DIRECTORY
git clone https://opendev.org/airship/airshipctl.git
pushd airshipctl
git checkout <release-tag|branch|commit-hash>
popd
git clone https://opendev.org/airship/treasuremap.git
pushd treasuremap
git checkout <release-tag|branch|commit-hash>
popd
```

Install Essential Tools

1. Install the essentials tools, including kubectl, kustomize, pip, and yq.

   From the *treasuremap* directory, run:

```
./tools/deployment/airship-core/01_install_essentials.sh
# Recommend to add the user to the docker group
sudo usermod -aG docker $USER
```

2. Install airshipctl executable.

```
./tools/deployment/airship-core/21_systemwide_executable.sh
```

2. (Optional) Install Apache Web server.

   Airship 2 deployment requires a web server to host the generated ephemeral ISO image. If you don't have an existing web server, you can install an Apache server on the build node.

```
sudo apt install apache2
```

Note: The Apache Web server must be accessible by the ephemeral host.

After the build node is established, you are ready to start creating your site manifests and deploying the site.

1.6.2 Site Authoring Guide

This guide describes the steps to create the site documents needed by Airship to deploy a standard greenfield bare metal deployment according to your specific environment. In the form of YAML comments, the `reference-airship-core` site manifests in the Treasuremap git repository contain the tags and descriptions of the required site specific information that the users must provide.

Airship Layering Approach

Following the DRY (Don't Repeat Yourself) principle, Airship uses four conceptual layer types to drive consistency and reusability:

- Function: An atomic, independent building block of a Kubernetes workload, e.g., a HelmRelease, Calico.
- Composite: A logical group of multiple Functions integrated and configured for a common purpose. Typical examples include OpenStack or interrelated logging and monitoring components. Composites can also pull in other Composites to further customize their configurations.
- Type: A prototypical deployment plan that represents a typical user case, e.g., network cloud, CI/CD pipeline, basic K8S deployment. A Type defines the collection of Composites into a deployable stack, including control plane, workload, and host definitions. A Type also defines the Phases of deployment, which serve to sequence a deployment (or upgrade) into stages that must be performed sequentially, e.g., the Kubernetes cluster must be created before a software workload is deployed. A type can inherit from another type.
- Site: A Site is a realization of exactly one Type, and defines (only) the site-specific configurations necessary to deploy the Type to a specific place.

To learn more about the Airship layering design and mechanism, it is highly recommended to read *Layering and Deduplication*.

Initializing New Site

It is a very complex and tedious task to create a new site form scratch. Therefore, it is strongly recommended that the user creates the new site based on a reference site. The reference site can be a site that has been already created and deployed by the user, or an example site in the treasuremap git repository.

The reference-airship-core site from the treasuremap may be used for such purpose. It is the principal pipeline for integration and continuous deployment testing of Airship on bare metal.

To create a new site definition from the `reference-airship-core` site, the following steps are required:

1. Clone or checkout the `treasuremap` repository at the specified reference in the Airship home directory.
2. Create a project side-by-side with `airshipctl` and `treasuremap` directory.
3. Copy the reference site manifests to ${PROJECT}/manifests/site/${SITE}.
4. Update the site's metadata.yaml appropriately.
5. Create and update the Airship config file for the site in `${HOME}/.airship/config`.

Airshipctl provides a tool `init_site.sh` that automates the above site creation tasks.

```
export AIRSHIPCTL_REF_TYPE=tag # type can be "tag", "branch" or "commithash"
export AIRSHIPCTL_REF=v2.1.0 # update with the git ref you want to use
export TREASUREMAP_REF_TYPE=tag # type can be "tag", "branch" or "commithash"
export TREASUREMAP_REF=v2.1.0 # update with the git ref you want to use
export REFERENCE_SITE=../treasuremap/manifests/site/reference-airship-core
export REFERENCE_TYPE=airship-core # the manifest type the reference site uses

./tools/init_site.sh
```

Note: The environment variables have default values that point to airshipctl release tag v2.1.0 and treasuremap release tag v2.1.0. You only need to (re)set them in the command line if you want a different release version, a branch or a specific commit.

To find the Airship release versions and tags, go to *Versioning*. In addition to release tag, the user can also specify a branch (e.g., v2.1) or a specific commit ID when checking out the `treasuremap` or `airshipctl` repository.

Preparing Deployment Documents

After the new site manifests are initialized, you will then need to manually make changes to these files. These site manifests are heavily commented to identify and explain the parameters that need to change when authoring a new site.

The areas that must be updated for a new site are flagged with the label `NEWSITE_CHANGEME` in YAML comments. Search for all instances of `NEWSITE_CHANGEME` in your new site definition. Then follow the instructions that accompany the tag in order to make all needed changes to author your new Airship site.

Because some files depend on (or may repeat) information from others, the order in which you should build your site files is as follows.

Note: A helpful practice is to replace the tag `NEWSITE_CHANGEME` with `NEWSITE_CHANGED` along the way when each site specific value is entered. You can run a global search on `NEWSITE_CHANGEME` at the end to check if any site fields were missed.

Network

Before you start, collect the following network information:

- PXE network interface name
- The name of the two 25G networks used for the bonded interface
- OAM, Calico and Storage VLAN ID's
- OAM, Calico and Storage network configuration
- PXE, OAM, Calico and Storage IP addresses for ephemeral/controller nodes and worker nodes
- Kubernetes and ingress virtual IP address (on OAM)
- DNS servers
- NTP servers

First, define the target and ephemeral networking catalogues.

- `manifests/site/${SITE}/target/catalogues/shareable/networking.yaml`: Contains the network definition in the entire system.
- `manifests/site/${SITE}/target/catalogues/shareable/networking-ha.yaml`: Defines the Kubernetes and ingress virtual IP addresses as well as the OAM interface.
- `manifests/site/${SITE}/ephemeral/catalogues/shareable/networking.yaml`: Provides only the overrides specific to the ephemeral nodes.

Last, update network references (e.g., interface name, IP address, port) in the target cluster deployment documents:

- `manifests/site/${SITE}/phases/phase-patch.yaml`
- `manifests/site/${SITE}/target/catalogues/shareable/versions-airshipctl.yaml`
- `manifests/site/${SITE}/target/controlplane/metal3machinetemplate.yaml`
- `manifests/site/${SITE}/target/controlplane/versions-catalogue-patch.yaml`
- `manifests/site/${SITE}/target/initinfra-networking/patch_calico.yaml`
- `manifests/site/${SITE}/target/workers/provision/metal3machinetemplate.yaml`
- `manifests/site/${SITE}/kubeconfig/kubeconfig.yaml`

Host Inventory

Host inventory configuration requires the following information for each server:

- host name
- BMC address
- BMC user and password
- PXE NIC mac address
- OAM | Calico | PXE | storage IP addresses

Update the host inventory and other ephemeral and target cluster documents:

- `manifests/site/${SITE}/host-inventory/hostgenerator/host-generation.yaml`: Lists the host names of the all the nodes in the host inventory.
- `manifests/site/${SITE}/target/catalogues/shareable/hosts.yaml`: The host catalogue defines the host information such as BMC address, credential, PXE NIC, IP addresses, hardware profile name, etc., for every single host.
- `manifests/site/${SITE}/ephemeral/bootstrap/baremetalhost.yaml`: Contains the host name and bmc address of the ephemeral bare metal host.
- `manifests/site/${SITE}/ephemeral/bootstrap/hostgenerator/host-generation.yaml`: Defines the single host in the ephemeral cluster.
- `manifests/site/${SITE}/ephemeral/controlplane/hostgenerator/host-generation.yaml`: Defines the host name of the first controller node to bootstrap ion the target cluster.
- `manifests/site/${SITE}/phases/phase-patch.yaml`: Updates the ephemeral node host name and ISO URL.
- `manifests/site/${SITE}/target/controlplane/hostgenerator/host-generation.yaml`: Defines the list of hosts to be deployed in the target cluster control plane.

- `manifests/site/${SITE}/target/workers/hostgenerator/host-generation.yaml`: Defines the list of hosts of the worker nodes.
- `manifests/site/${SITE}/target/workers/provision/machinedeployment.yaml`: Configures the total number of worker nodes.

Storage

Using a general purpose Ceph cluster configuration in Treasuremap manifests, Rook will deploy Ceph cluster using all the remaining disks, i.e., raw devices with no partitions or formatted filesystems, on all the hosts (both controller and worker nodes).

The user can customize the Ceph cluster configuration based on specific use cases. The following is an example to specify which hosts to include in the storage cluster and which devices to use for Ceph:

- `treasuremap/manifests/site/{SITE}/target/catalogues/shareable/storage.yaml`

For more information, please refer to Rook Ceph Storage document.

Downstream Images and Binaries

For a production environment, the access to external resources such as the `quay.io` or various `go` packages may not be available, or further customized security hardening is required in the images.

In those cases, the operator will need to host their pre-built images or binaries in a downstream repository or artifactory. The manifests specifying image locations for the Kustomize plugins will need to be updated prior to running airshipctl commands, e.g., replacement-transformer, templater, sops, etc.

Here is an example `sed` command on the cloned airshipctl and treasuremap manifests for updating the image locations:

```
find ./airshipctl/manifests/ ./treasuremap/manifests/ -name "*.yaml" -type f -
↪readable -writable -exec sed -i \
  -e "s,gcr.io/kpt-fn-contrib/sops:v0.1.0,docker-artifacts.my-telco.com/upstream-
↪local/kpt-fn-contrib/sops:v0.1.0,g" -i \
  -e "s,quay.io/airshipit/templater:latest,docker-artifacts.my-telco.com/upstream-
↪local/airshipit/templater:latest,g" -i \
  -e "s,quay.io/airshipit/replacement-transformer:latest,docker-artifacts.my-telco.
↪com.com/upstream-local/airshipit/replacement-transformer:latest,g" {} +;
```

Now the manifests for the new site are ready for deployment.

1.6.3 Site Deployment Guide

This document is the Airship 2 site deployment guide for a standard greenfield bare metal deployment. The following sections describes how to apply the site manifests for a given site.

Prerequisites

Before starting, ensure that you have completed *system requirements and set up*, including the the BIOS and Redfish settings, hardware RAID configuration etc.

> **Warning:** Ensure all the hosts are powered off, including ephemeral node, controller nodes and worker nodes.

Airshipctl Phases

A new concept with Airship 2 is *phases* and *phase plans*. A phase is a step to be performed in order to achieve a desired state of the managed site. A plan is a collection of phases that should be executed in sequential order. The use of phases and phase plan is to simplify executing deployment and life cycle operations.

The Airship 2 deployment uses heavily the `airshipctl` commands, especially the `airshipctl plan run` and `airshipctl phase run` commands. You may find it helpful to get familirized with the airshipctl command reference and example usage.

To faciliate the site deployment, the Airship Treasuremap project provides a set of deployment scripts in the `tools/deployment/{TYPE_NAME}` directory. These scripts are wrappers of the *airshipctl* commands with additional flow controls. They are numbered sequentially in the order of the deployment operations.

Environment Variables

The deployment steps below use a few additional environment variables that are already configured with default values for a typical deployment or inferred from other configuration files or site manifests. In most situations, users do not need to manually set the values for these environment variables.

- `KUBECONFIG`: The location of the kubeconfig file. Default value: `$HOME/.airship/kubeconfig`.
- `KUBECONFIG_TARGET_CONTEXT`: The name of the kubeconfig context for the target cluster. Default value: "target-cluster". You can find it defined in the Airshipctl configuration file.
- `KUBECONFIG_EPHEMERAL_CONTEXT`: The name of the kubeconfig context for the ephemeral cluster. Default value: "ephemeral-cluster". You can find it defined in the Airshipctl configuration file.
- `TARGET_IP`: The control plane endpoint IP or host name. Default value: derived from site documents for the `controlplane-target` phase. You can run the following command to extract the defined value:

```
airshipctl phase render controlplane-target \
-k Metal3Cluster -l airshipit.org/stage=initinfra \
2> /dev/null | yq .spec.controlPlaneEndpoint.host | sed 's/"//g'
```

- `TARGET_PORT`: The control plane endpoint port number. Default value: derived from site documents for the `controlplane-target` phase. You can run the following command to extract the defined value:

```
airshipctl phase render controlplane-target \
-k Metal3Cluster -l airshipit.org/stage=initinfra 2> /dev/null | \
yq .spec.controlPlaneEndpoint.port
```

- `TARGET_NODE`: The host name of the first controller node. Default value: derived from site documents for the `controlplane-ephemeral` phase. You can run the following command to extract the defined value:

```
airshipctl phase render controlplane-ephemeral \
-k BareMetalHost -l airshipit.org/k8s-role=controlplane-host 2> /dev/null | \
yq .metadata.name | sed 's/"//g'
```

- `WORKER_NODE`: The host name of the worker nodes. Default value: derived from site documents for the `workers-target` phase. You can run the following command to extract the defined value:

```
airshipctl phase render workers-target -k BareMetalHost 2> /dev/null | \
yq .metadata.name | sed 's/"//g'
```

Configuring Airshipctl

Airship requires a configuration file set that defines the intentions for the site that needs to be created. These configurations include such items as manifest repositories, ephemeral and target cluster context and bootstrap information. The operator seeds an initial configuration using the configuration initialization function.

The default location of the configuration files is `$HOME/.airship/config` and `$HOME/.airship/kubeconfig`.

When you run the init_site script in the *Initializing New Site* section, the `.airship/config` file has been already created for you.

Warning: If the Redfish api uses self-signed certificate, the user must run:

```
airshipctl config set-management-config default --insecure
```

This will inject the `insecure` flag to the Airship configuration file as follows:

```
managementConfiguration:
  default:
    insecure: true
    systemActionRetries: 30
    systemRebootDelay: 30
    type: redfish
```

Now let's create the `.airship/kubeconfig`. If you plan to use an existing external kubeconfig file, run:

```
airshipctl config import <KUBE_CONFIG>
```

Otherwise, create an empty kubeconfig that will be populated later by airshipctl:

```
touch ~/.airship/kubeconfig
```

More advanced users can use the Airshipctl config commands to generate or update the configuration files.

To generate an Airshipctl config file from scratch,

```
airshipctl config init [flags]
```

To specify the location of the manifest repository,

```
airshipctl config set-manifest <MANIFEST_NAME> [flags]
```

To create or modify a context in the airshipctl config files,

```
airshipctl config set-context <CONTEXT_NAME> --manifest <MANIFEST_NAME> [flags]
```

Full details on the `config` command can be found here.

Generating and Encrypting Secrets

Airship site manifests contain different types of secrets, such as passwords, keys and certificates in the variable catalogues. Externally provided secrets, such as BMC credentials, are used by Airship and Kubernetes and can also be used by other systems. Secrets can also be internally generated by Airshipctl, e.g., Openstack Keystone password, that no external systems will provide or need.

To have Airshipctl generate and encrypt the secrets, run the following script from the treasuremap directory:

```
./tools/deployment/airship-core/23_generate_secrets.sh
```

The generated secrets will be updated in:

- `${PROJECT}/manifests/site/${SITE}/target/generator/results/generated/secrets.yaml`
- `${HOME}/.airship/kubeconfig.yaml`

It is recommended that you save the generated results, for example, commit them to a git repository along with the rest of site manifests.

To update the secrets for an already deployed site, you can re-run this script and apply the new secret manifests by re-deploying the whole site.

For more details and trouble shooting, please refer to Secrets generation and encryption how-to-guide.

Validating Documents

After constituent YAML configurations are finalized, use the document validation tool to lint and check the new site manifests. Resolve any issues that result from the validation before proceeding.

```
./tools/validate_docs
```

Caution: The validate_docs tool will run validation against all sites found in the `manifests/site` folder. You may want to (temporarily) remove other sites that are not to be deployed to speed up the validation.

To validate a single site's manifest,

```
export MANIFEST_ROOT=./${PROJECT}/manifests
export SITE_ROOT=./${PROJECT}/manifests/site
cd airshipctl && ./tools/document/validate_site_docs.sh
```

Estimated runtime: **5 minutes**

Building Ephemeral ISO Image

The goal for this step is to generate a custom targeted image for bootstrapping an ephemeral host with a Kubernetes cluster installed. This image may then be published to a repository to which the ephemeral host will have remote access. Alternatively, an appropriate media delivery mechanism (e.g. USB) can be used to bootstrap the ephemeral host manually.

Note: The generate ISO image content includes:

- Host OS Image
- Runtime engine: Docker/containerd
- Kubelet
- Kubeadm
- YAML file for KubeadmConfig

First, create an output directory for ephemeral ISO image and run the `bootstrap-iso` phase:

```
sudo mkdir /srv/images
airshipctl phase run bootstrap-iso
```

Or, run the provided script from the treasuremap directory:

```
./tools/deployment/airship-core/24_build_images.sh
```

Then, copy the generated ephemeral ISO image to the Web hosting server that will serve the ephemeral ISO image. The URL for the image should match what is defined in `manifests/site/{SITE}/phases/phase-patch.yaml`.

For example, if you have installed the Apache Web server on the jump host as described in the earlier step, you can simply execute the following:

```
sudo cp /srv/images/ephemeral.iso /var/www/html/
```

Estimated runtime: **5 minutes**

Deploying Site

Now that the ephemeral ISO image in place, you are ready to deploy the site. The deployment involves the following tasks:

- Deploying Ephemeral Node: Creates an ephemeral Kubernetes instance where the `cluster-api` bootstrap flow can be executed subsequently. It deploys the ephemeral node via Redfish with the ephemeral ISO image generated previously `Calico`, `metal3.io` and `cluster-api` components onto the ephemeral node. Estimated runtime: **20 minutes**
- Deploying Target Cluster: Provisions the target cluster's first control plane node using the cluster-api bootstrap flow in the ephemeral cluster, deploys the infrastructure components inlcuding Calico and meta3.io and `cluster-api` components, then complete the target cluster by provisioning the rest of the control plane nodes. The ephemeral node is stopped as a result. Estimated runtime: **60-90 minutes**
- Provisioning Worker Nodes: uses the target control plane Kubernetes host to deploy, classify and provision the worker nodes. Estimated runtime: **20 minutes**
- Deploying Workloads: The Treasuremap type `airship-core` deploys the following workloads by default: ingress, storage-cluster. Estimated runtime: Varies by the workload contents.

The phase plan `deploy-gating` in the `treasuremap/manifests/site/reference-airship-core/phases/baremetal-plan.yaml` defines the list of phases that are required to provision a typical bare metal site. Invoke the phase plan run command to start the deployment:

```
airshipctl plan run deploy-gating --debug
```

Note: If desired or if Redfish is not available, the ISO image can be mounted through other means, e.g. out-of-band management or a USB drive. In such cases, the user should provide a patch in the site manifest to remove the `remotedirect-ephemeral` phase from the phases list in the `treasuremap/manifests/site/reference-airship-core/phases/baremetal-plan.yaml`.

Note: The user can add other workload functions to the target workload phase in the `airship-core` type, or create their own workload phase from scratch.

Adding a workload function involves two tasks. First, the user will create the function manifest(s) in the `$PROJECT/manifest/function` directory. A good example can be found in the ingress function from Treasuremap. Second, the user overrides the kustomization of the target workload phase to include the new workload function in the `$PROJECT/manifests/site/$SITE/target/workload/kustomization.yaml`.

For more detailed reference, please go to Kustomize and airshipctl phases documentation.

Warning: When the second controller node joins the cluster, the script may fail with the error message `"etcdserver: request timed out"`. This is a known issue. You can just wait until all the other controller nodes join the cluster before executing the next phase. To check the list of nodes in the cluster, run:

```
kubectl --kubeconfig ${HOME}/.airship/kubeconfig --context target-cluster get nodes
```

Accessing Nodes

Operators can use ssh to access the controller and worker nodes via the OAM IP address. The user id and ssh key can be retrieved using the airshipctl phase render command:

```
airshipctl phase render controlplane-ephemeral
```

The user can also access the ephemeral node via ssh using the OAM IP from the networking catalogue and the user name and password found in the airshipctl phase render command output.

```
airshipctl phase render iso-cloud-init-data
```

Tearing Down Site

To tear down a deployed bare metal site, the user can simply power off all the nodes and clean up the deployment artifacts on the build node as follows:

```
airshipctl baremetal poweroff --name <server-name> # alternatively, use iDrac or iLO
rm -rf ~/.airship/ /srv/images/*
docker rm -f -v $(sudo docker ps --all -q | xargs -I{} sudo bash -c 'if docker␣
↪inspect {} | grep -q airship; then echo {} ; fi')
docker rmi -f $(sudo docker images --all -q | xargs -I{} sudo bash -c 'if docker␣
↪image inspect {} | grep -q airship; then echo {} ; fi')
```

1.6.4 Support

Bugs may be viewed and reported using GitHub issues for specific projects in Airship group:

- Airship airshipctl
- Airship charts
- Airship hostconfig-operator
- Airship images
- Airship sip
- Airship treasuremap

- Airship vino

1.6.5 Terminology

Please refer to *Airship Glossary of Terms* for the terminology used in this document.

1.6.6 Versioning

This document requires Airship Treasuremap and Airshipctl release version v2.0.0 or newer.

Airship Treasuremap reference manifests are delivered periodically as release tags in the Treasuremap Releases.

Airshipctl manifests can be found as release tags in the Airshipctl Releases.

Note: The releases are verified by Airship in Baremetal Environment, and Airship in Virtualized Environment pipelines before delivery and are recommended for deployments instead of using the master branch directly.

1.7 Integration with Cluster API Providers

The Cluster-API (CAPI) is a Kubernetes project to bring declarative, Kubernetes-style APIs to cluster creation, configuration, and management. By leveraging the Cluster-API for cloud provisioning, Airship takes advantage of upstream efforts to build Kubernetes clusters and manage their lifecycle. Most importantly, we can leverage a number of `CAPI` providers that already exist. This allows Airship deployments to target both public and private clouds, such as Azure, AWS, and OpenStack.

The Site Authoring Guide and Deployment Guide in this document focus on deployment on the bare metal infrastructure, where Airship utilizes the `Metal3-IO` Cluster API Provider for Managed Bare Metal Hardware (CAPM3) and Cluster API Bootstrap Provider Kubeadmin (CABPK).

There are Cluster-API Providers that support Kubernetes deployments on top of already provisioned infrastructure, enabling Bring Your Own bare metal use cases as well. Here is a list of references on how to use Airshipctl to create Cluster API management cluster and workload clusters on various different infrastructure providers:

- Airshipctl and Cluster API Docker Integration
- Airshipctl and Cluster API Openstack Integration
- Airshipctl and Cluster API GCP Provider Integration
- Airshipctl and Azure Cloud Platform Integration

1.8 Getting Started

Thank you for your interest in Airship. Our community is eager to help you contribute to the success of our project and welcome you as a member of our community!

We invite you to reach out to us at any time via the Airship mailing list or on our Slack workspace.

Welcome aboard!

1.9 Airship 2 Development

Development is underway on Airship 2: the educated evolution of Airship 1, designed with our experience using Airship in production. Airship 2 makes the Airship control plane ephemeral, leverages entrenched upstream projects such as the Cluster API, Metal Kubed, Kustomize, and kubeadm, and embraces Kubernetes Custom Resource Definitions (CRDs). To learn more about the Airship 2.0 evolution, see the Airship 2 evolution blog series.

Each Airship 2 project has its own development guidelines. Join the ongoing Airship 2 development by referencing the airshipctl or the airshipui documentation.

1.9.1 Testing Changes

Testing of Airship changes can be accomplished several ways:

1. Standalone, single component testing
2. Integration testing
3. Linting, unit, and functional tests/linting

Note: Testing changes to charts in Airship repositories is best accomplished using the integration method describe below.

1.9.2 Final Checks

Airship projects provide Makefiles to run unit, integration, and functional tests as well as lint Python code for PEP8 compliance and Helm charts for successful template rendering. All checks are gated by Zuul before a change can be merged. For more information on executing these checks, refer to project-specific documentation.

Third party CI tools, such as Jenkins, report results on Airship-in-a-Bottle patches. These can be exposed using the "Toggle CI" button in the bottom left-hand page of any gerrit change.

1.9.3 Pushing code

Airship uses the OpenDev gerrit for code review. Refer to the OpenStack Contributing Guide for a tutorial on submitting changes to Gerrit code review.

1.9.4 Next steps

Upon pushing a change to gerrit, Zuul continuous integration will post job results on your patch. Refer to the job output by clicking on the job itself to determine if further action is required. If it's not clear why a job failed, please reach out to a team member in IRC. We are happy to assist!

Assuming all continuous integration jobs succeed, Airship community members and core developers will review your patch and provide feedback. Many patches are submitted to Airship projects each day. If your patch does not receive feedback for several days, please reach out using IRC or the Airship mailing list.

1.9.5 Merging code

Like most OpenDev projects, Airship patches require two +2 code review votes from core members to merge. Once you have addressed all outstanding feedback, your change will be merged.

1.9.6 Beyond

Congratulations! After your first change merges, please keep up-to-date with the team. We hold two weekly meetings for project and design discussion:

Our weekly #airshipit IRC meeting provides an opportunity to discuss project operations.

Our weekly design call provides an opportunity for in-depth discussion of new and existing Airship features.

For more information on the times of each meeting, refer to the Airship wiki.

1.10 Getting Started

Thank you for your interest in Airship. Our community is eager to help you contribute to the success of our project and welcome you as a member of our community!

We invite you to reach out to us at any time via the Airship mailing list or on our Slack workspace.

Welcome aboard!

1.11 Airship 1 Development

Airship 1 is a collection of open source tools that automate cloud provisioning and management. Airship provides a declarative framework for defining and managing the life cycle of open infrastructure tools and the underlying hardware. These tools include OpenStack for virtual machines, Kubernetes for container orchestration, and MaaS for bare metal, with planned support for OpenStack Ironic.

Improvements are still made Airship 1 daily, and we welcome additional contributors. We recommend that new contributors begin by reading the high-level architecture overview included in our treasuremap documentation. The architectural overview introduces each Airship component, their core responsibilities, and their integration points.

1.11.1 Deep Dive

Each Airship component is accompanied by its own documentation that provides an extensive overview of the component. With so many components, it can be challenging to find a starting point.

We recommend the following:

1.11.2 Try an Airship environment

Airship provides two single-node environments for demo and development purpose.

Airship-in-a-Bottle is a set of reference documents and shell scripts that stand up a full Airship environment with the execution of a script.

Airskiff is a light-weight development environment bundled with a set of deployment scripts that provides a single-node Airship environment. Airskiff uses minikube to bootstrap Kubernetes, so it does not include Drydock, MaaS, or Promenade.

Additionally, we provide a reference architecture for easily deploying a smaller, demo site.

Airsloop is a fully-authored Airship site that can be quickly deployed as a bare metal, demo lab.

1.11.3 Focus on a component

When starting out, focusing on one Airship component allows you to become intricately familiar with the responsibilities of that component and understand its function in the Airship integration. Because the components are modeled after each other, you will also become familiar with the same patterns and conventions that all Airship components use.

Airship source code lives in the OpenDev Airship namespace. To clone an Airship project, execute the following, replacing <*component*> with the name of the Airship component you want to clone.

Refer to the component's documentation to get started. A list of each component's documentation is listed below for reference:

- Armada
- Deckhand
- Divingbell
- Drydock
- Pegleg
- Promenade
- Shipyard

1.11.4 Testing Changes

Testing of Airship changes can be accomplished several ways:

1. Standalone, single component testing
2. Integration testing
3. Linting, unit, and functional tests/linting

Note: Testing changes to charts in Airship repositories is best accomplished using the integration method describe below.

1.11.5 Standalone Testing

Standalone testing of Airship components, i.e. using an Airship component as a Python project, provides the quickest feedback loop of the three methods and allows developers to make changes on the fly. We recommend testing initial code changes using this method to see results in real-time.

Each Airship component written in Python has pre-requisites and guides for running the project in a standalone capacity. Refer to the documentation listed below.

- Armada
- Deckhand
- Drydock
- Pegleg
- Promenade
- Shipyard

1.11.6 Integration Testing

While each Airship component supports individual usage, Airship components have several integration points that should be exercised after modifying functionality.

We maintain several environments that encompass these integration points:

1. Airskiff: Integration of Armada, Deckhand, Shipyard, and Pegleg
2. *Airship-in-a-Bottle Multinode*: Full Airship integration

For changes that merely impact software delivery components, exercising a full Airskiff deployment is often sufficient. Otherwise, we recommend using the Airship-in-a-Bottle Multinode environment.

Each environment's documentation covers the process required to build and test component images.

1.11.7 Final Checks

Airship projects provide Makefiles to run unit, integration, and functional tests as well as lint Python code for PEP8 compliance and Helm charts for successful template rendering. All checks are gated by Zuul before a change can be merged. For more information on executing these checks, refer to project-specific documentation.

Third party CI tools, such as Jenkins, report results on Airship-in-a-Bottle patches. These can be exposed using the "Toggle CI" button in the bottom left-hand page of any gerrit change.

1.11.8 Pushing code

Airship uses the OpenDev gerrit for code review. Refer to the OpenStack Contributing Guide for a tutorial on submitting changes to Gerrit code review.

1.11.9 Next steps

Upon pushing a change to gerrit, Zuul continuous integration will post job results on your patch. Refer to the job output by clicking on the job itself to determine if further action is required. If it's not clear why a job failed, please reach out to a team member in IRC. We are happy to assist!

Assuming all continuous integration jobs succeed, Airship community members and core developers will review your patch and provide feedback. Many patches are submitted to Airship projects each day. If your patch does not receive feedback for several days, please reach out using IRC or the Airship mailing list.

1.11.10 Merging code

Like most OpenDev projects, Airship patches require two +2 code review votes from core members to merge. Once you have addressed all outstanding feedback, your change will be merged.

1.11.11 Beyond

Congratulations! After your first change merges, please keep up-to-date with the team. We hold two weekly meetings for project and design discussion:

Our weekly #airshipit IRC meeting provides an opportunity to discuss project operations.

Our weekly design call provides an opportunity for in-depth discussion of new and existing Airship features.

For more information on the times of each meeting, refer to the Airship wiki.

1.12 Airship Conventions

Airship components conform to a minimal set of conventions to provide for reasonable levels of consistency.

1.12.1 Language

While these documents are not an IETF RFC, RFC 2119 provides for useful language definitions. In this spirit:

- 'must', 'shall', 'will', and 'required' language indicates inflexible rules.
- 'should' and 'recommended' language is expected to be followed but reasonable exceptions may exist.
- 'may' and 'can' language is intended to be optional, but will provide a recommended approach if used.

1.12.2 Conventions and Standards

API Conventions

A collection of conventions that components of Airship utilize for their REST APIs

Resource path naming

- Resource paths nodes follow an all lower case naming scheme, and pluralize the resource names. Nodes that refer to keys, ids or names that are externally controlled, the external naming will be honored.
- The version of the API resource path will be prefixed before the first node of the path for that resource using v#.# format.
- By default and unless otherwise noted, the API will be namespaced by /api before the version. For the purposes of documentation, this will not be specified in each of the resource paths below. In more complex APIs, Airship components may use values other than /api to be more specific to point to a particular service.

```
/api/v1.0/sampleresources/ExTeRnAlNAME-1234
      ^         ^       ^       ^
      |         |       |      defer to external naming
      |         |      plural
      |        lower case
     version here
```

Status responses

Status responses, and more specifically error responses (HTTP response body accompanying 4xx and 5xx series responses where possible) are a customized version of the Kubernetes standard for error representation. Airship utilizes the details field in a more formalized way to represent multiple messages related to a status response, as follows:

```
{
  "kind": "Status",
  "apiVersion": "v{{#.#}}",
  "metadata": {},
  "status": "{{Success | Failure}}",
  "message": "{{message phrase}}",
```

(continues on next page)

(continued from previous page)

```
  "reason": "{{reason name}}",
  "details": {
    "errorCount": {{n}},
    "messageList": [
       { "message" : "{{message contents}}",
         "error": true|false,
         "kind": "SimpleMessage" }
       ...
    ]
  },
  "code": {{http status code}}
}
```

such that:

- The metadata field is optionally present, as an empty object. Clients should be ready to receive this field, but services are not required to produce it.
- The message phrase is a terse but descriptive message indicating what has happened.
- The reason name is the short name indicating the cause of the status. It should be a camel cased phrase-as-a-word, to mimic the Kubernetes status usage.
- The details field is optional.
- If used, the details follow the shown format, with an errorCount and messageList field present.
- The repeating entity inside the messageList can be decorated with as many other fields as are useful, but at least have a message field and error field.
  - A kind field is optional, but if used will indicate the presence of other fields. By default, the kind field is assumed to be "SimpleMessage", which requires only the aforementioned message and error fields.
- The errorCount field is an integer representing the count of messageList entities that have `error: true`
- When using this document as the body of a HTTP response, `code` is populated with a valid HTTP status code

Required Headers

X-Auth-Token The auth token to identify the invoking user. Required unless the resource is explicitly unauthenticated.

Optional Headers

X-Context-Marker A context id that will be carried on all logs for this client-provided marker. This marker may only be a 36-character canonical representation of an UUID (8-4-4-4-12)

X-End-User The user name of the initial invoker that will be carried on all logs for user tracing cross components. Shipyard doesn't support this header and when passed, it will be ignored.

Validation API

All Airship components that participate in validation of the design supplied to a site implement a common resource to perform document validations. Document validations are synchronous. Because of the different sources of documents that should be supported, a flexible input descriptor is used to indicate from where an Airship component will retrieve the documents to be validated.

POST /v1.0/validatedesign

Invokes an Airship component to perform validations against the documents specified by the input structure. Synchronous.

Input structure

```
{
  rel : "design",
  href: "deckhand+https://{{deckhand_url}}/revisions/{{revision_id}}/rendered-
↪documents",
  type: "application/x-yaml"
}
```

Output structure

The output structure reuses the Kubernetes Status kind to represent the result of validations. The Status kind will be returned for both successful and failed validation to maintain a consistent of interface. If there are additional diagnostics that associate to a particular validation, the entries in the messageList should be of kind "ValidationMessage" (preferred), or "SimpleMessage" (assumed default base message kind).

Failure message example using a ValidationMessage kind for the messageList:

```
{
  "kind": "Status",
  "apiVersion": "v1.0",
  "metadata": {},
  "status": "Failure",
  "message": "{{Component Name}} validations failed",
  "reason": "Validation",
  "details": {
    "errorCount": {{n}},
    "messageList": [
       { "message" : "{{validation failure message}}",
         "error": true,
         "name": "{{identifying name of the validation}}",
         "documents": [
             { "schema": "{{schema and name of the document being validated}}",
               "name": "{{name of the document being validated}}"
             },
             ...
         ]
         "level": "Error",
         "diagnostic": "{{information about what lead to the message}}",
         "kind": "ValidationMessage" },
       ...
    ]
  },
  "code": 400
}
```

Success message example:

```
{
  "kind": "Status",
  "apiVersion": "v1.0",
  "metadata": {},
  "status": "Success",
  "message": "{{Component Name}} validations succeeded",
  "reason": "Validation",
  "details": {
    "errorCount": 0,
    "messageList": []
  },
  "code": 200
}
```

ValidationMessage Message Type

The ValidationMessage message type is used to provide more information about validation results than a SimpleMessage provides. These are the fields of a ValidationMessage:

- documents (optional): If applicable to configuration documents, specifies the design documents by schema and name that were involved in the specific validation. If the documents element is not provided, or is an empty list, the assumption is that the validation is not traced to a document, and may be a validation of environmental or process needs.
  - schema (required): The schema of the document. E.g. drydock/NetworkLink/v1
  - name (required): The name of the document. E.g. pxe-rack1
- error (required): true if the message indicates an error, false if the message indicates a non-error.
- kind (required): ValidationMessage
- level (required): The severity of the validation result. This should align with the error field value. Valid values are "Error", "Warning", and "Info".
- message (required): The more complete message indicating the result of the validation. E.g.: MTU 8972 for pxe-rack1 is invalid for standard (non-jumbo) frames
- name (required): The name of the validation being performed. This is a short name that identifies the validation among a full set of validations. It is preferred to use non-action words to identify the validation. E.g. "MTU in bounds" is preferred instead of "Check MTU in bounds"
- diagnostic (optional): Provides further contextual information that may help with determining the source of the validation or provide further details.

Health Check API

Each Airship component shall expose an endpoint that allows other components to access and validate its health status. Clients of the health check should wait up to 30 seconds for a health check response from each component.

GET /v1.0/health

Invokes an Airship component to return its health status. This endpoint is intended to be unauthenticated, and must not return any information beyond the noted 204 or 503 status response. The component invoked is expected to return a response in less than 30 seconds.

Health Check Output

The current design will be for the component to return an empty response to show that it is alive and healthy. This means that the component that is performing the query will receive HTTP response code 204.

HTTP response code 503 with a generic response status or an empty message body will be returned if the component determines it is in a non-healthy state, or is unable to reach another component it is dependent upon.

GET /v1.0/health/extended

Airship components may provide an extended health check. This request invokes a component to return its detailed health status. Authentication is required to invoke this API call.

Extended Health Check Output

The output structure reuses the Kubernetes Status kind to represent the health check results. The Status kind will be returned for both successful and failed health checks to ensure consistencies. The message field will contain summary information related to the results of the health check. Detailed information of the health check will be provided as well.

Failure message example:

```
{
  "kind": "Status",
  "apiVersion": "v1.0",
  "metadata": {},
  "status": "Failure",
  "message": "{{Component Name}} failed to respond",
  "reason": "HealthCheck",
  "details": {
    "errorCount": {{n}},
    "messageList": [
       { "message" : "{{Detailed Health Check failure information}}",
         "error": true,
         "kind": "SimpleMessage" },
       ...
    ]
  },
  "code": 503
}
```

Success message example:

```
{
  "kind": "Status",
  "apiVersion": "v1.0",
  "metadata": {},
  "status": "Success",
  "message": "",
  "reason": "HealthCheck",
  "details": {
    "errorCount": 0,
    "messageList": []
  },
```

(continues on next page)

(continued from previous page)

```
  "code": 200
}
```

Versions API

Each Airship component shall expose an endpoint that allows other components to discover its different API versions. This endpoint is not prefixed by /api or a version.

GET /versions

Invokes an Airship component to return its list of API versions. This endpoint is intended to be unauthenticated, and must not return any information beyond the output noted below.

Versions output

Each Airship component shall return a list of its different API versions. The response body shall be keyed with the name of each API version, with accompanying information pertaining to the version's *path* and *status*. The *status* field shall be an enum which accepts the values *stable* and *beta*, where *stable* implies a stable API and *beta* implies an under-development API.

Success message example:

```
{
  "v1.0": {
    "path": "/api/v1.0",
    "status": "stable"
  },
  "v1.1": {
    "path": "/api/v1.1",
    "status": "beta"
  },
  "code": 200
}
```

Code and Project Conventions

Conventions and standards that guide the development and arrangement of Airship component projects.

Project Structure

Charts

Each project that maintains helm charts will keep those charts in a directory `charts` located at the root of the project. The charts directory will contain subdirectories for each of the charts maintained as part of that project. These subdirectories should be named for the component represented by that chart.

E.g.: For project `foo`, which also maintains the charts for `bar` and `baz`:

- foo/charts/foo contains the chart for `foo`

- foo/charts/bar contains the chart for `bar`
- foo/charts/baz contains the chart for `baz`

Helm charts utilize the helm-toolkit supported by the Openstack-Helm team and follow the standards documented there.

Images

Each project that creates Docker images will keep Dockerfiles in a directory `images` located at the root of the project. The images directory will contain subdirectories for each of the images created as part of that project. The subdirectory will contain Dockerfiles that can be used to generate images.

E.g.: For project `foo`, which also produces a Docker image for `bar`:

- foo/images/foo contains Dockerfiles for `foo`
- foo/images/bar contains Dockerfiles for `bar`

Each image must include the following set of labels conforming to the OCI image annotations standard as the minimum:

```
org.opencontainers.image.authors='airship-discuss@lists.airshipit.org, irc://
↪#airshipit@freenode'
org.opencontainers.image.url='https://airshipit.org'
org.opencontainers.image.documentation='<documentation on readthedocs or in␣
↪repository URL>'
org.opencontainers.image.source='<repository URL>'
org.opencontainers.image.vendor='The Airship Authors'
org.opencontainers.image.licenses='Apache-2.0'
org.opencontainers.image.revision='<Git commit ID>'
org.opencontainers.image.created='UTC date and time in RFC3339 format with seconds'
org.opencontainers.image.title='<image name, e.g. "armada">'
```

Last three annotations (revision, created and title), being dynamic, are added on a container build stage. Others are statically defined in Dockerfiles. Optional custom `org.airshipit.build=community` annotation is added to the Airship images published to the community.

Image tags must follow format:

- `:<full Git commit ID>_<distro_suffix>`
- `:<branch>_<distro_suffix>` - latest image built from specific branch
- `:latest_<distro_suffix>` - latest image built from master

The `_<distro suffix>` (e.g. `_ubuntu_xenial`) could be omitted. See Airship Multiple Linux Distribution Support specification for details.

Images should follow best practices for the container images. Be slim and secure in particular.

Dockerfile

Dockerfile file names must follow format: `Dockerfile.<distro_suffix>`, where `<distro_suffix>` matches corresponding image tag suffix. The `.<distro_suffix>` could be omitted where not relevant.

Lines should be indented by a space character next to the Dockerfile instruction block they correspond to.

Dockerfile must allow base image substitution via `FROM` argument. This is to allow the use of base images stored in third-party or internal repositories.

Dockerfile should follow best practices. Use multistage container builds where possible to reduce image size and attack surface. `RUN` statements should pass linting via `shellcheck`. You may use available Dockerfile linters, if you wish to do so.

See example Dockerfile file for reference.

Makefile

Each project must provide a Makefile at the root of the project. The Makefile should implement each of the following Makefile targets:

- `images` will produce the docker images for the component and each other component it is responsible for building.
- `charts` will helm package all of the charts maintained as part of the project.
- `lint` will perform code linting for the code and chart linting for the charts maintained as part of the project, as well as any other reasonable linting activity.
- `dry-run` will produce a helm template for the charts maintained as part of the project.
- `all` will run the lint, charts, and images targets.
- `docs` should render any documentation that has build steps.
- `run_{component_name}` should build the image and do a rudimentary (at least) test of the image's functionality.
- `run_images` performs the individual run_{component_name} targets for projects that produce more than one image.
- `tests` to invoke linting tests (e.g. PEP-8) and unit tests for the components in the project
- `format` to invoke automated code formatting specific to the project's code language (e.g. Python for armada and Go for airshipctl) as listed in the Linting and Formatting Standards.

For projects that are Python based, the Makefile targets typically reference tox commands, and those projects will include a tox.ini defining the tox targets. Note that tox.ini files will reside inside the source directories for modules within the project, but a top-level tox.ini may exist at the root of the repository that includes the necessary targets to build documentation.

Documentation

Also see *Documentation*.

Documentation source for the component should reside in a 'docs' directory at the root of the project.

Linting and Formatting Standards

Code in the Airship components should follow the prevalent linting and formatting standards for the language being implemented. In lieu of industry accepted code formatting standards for a target language, strive for readability and maintainability.

Known Standards	
Language	Tools Used
Ansible	ansible-lint
Bash	Shellcheck
Go	gofmt
Markdown	markdownlint
Python	YAPF, Flake8

Ansible formatting

Ansible code should be linted to be conformant to the standards checked by ansible-lint project.

Bash Formatting

Bash shell scripts code should be linted to be conformant to the standards checked by Shellcheck project.

Bash shell scripts code in Helm templates should ideally be linted as well, however gating of it is a noble goal and is only desired.

Go Formatting

Go code should be formatted using gofmt. When using gofmt be sure to use the -s flag to include simplification of code for example:

```
gofmt -s /path/to/file.go
```

Markdown Formatting

Markdown code (documentation) should be linted to be conformant to the standards checked by markdownlint project.

Python PEP-8 Formatting

Python should be formatted via YAPF. The knobs for YAPF can be specified in the project's root directory in '.style.yapf'. The contents of this file should be:

```
[style]
based_on_style = pep8
spaces_before_comment = 2
column_limit = 79
blank_line_before_nested_class_or_def = false
blank_line_before_module_docstring = true
split_before_logical_operator = true
split_before_first_argument = true
allow_split_before_dict_value = false
split_before_arithmetic_operator = true
```

A sample Flake8 section is below, for use in tox.ini, and is the method of enforcing import orders via Flake8 extension flake8-import-order:

```
[flake8]
filename = *.py
show-source = true
# [H106] Don't put vim configuration in source files.
# [H201] No 'except:' at least use 'except Exception:'
# [H904] Delay string interpolations at logging calls.
enable-extensions = H106,H201,H904
# [W503] line break before binary operator
ignore = W503
exclude=.venv,.git,.tox,build,dist,*lib/python*,*egg,tools,*.ini,*.po,*.pot
max-complexity = 24
```

Airship components must provide for automated checking of their formatting standards, such as the lint step noted above in the Makefile, and in the future via CI jobs. Components may provide automated reformatting.

YAML Schema

YAML schema defined by Airship should have key names that follow camelCase naming conventions.

Note that Airship also integrates and consumes a number of projects from other open source communities, which may have their own style conventions, and which will therefore be reflected in Airship deployment manifests. Those fall outside the scope of these Airship guidelines.

Any YAML schema that violate this convention at the time of this writing (e.g. with snake_case keys) may be either grandfathered in, or converted, at the development team's discretion.

Tests Location

Tests should be in parallel structures to the related code, unless dictated by target language ecosystem.

For Python projects, the preferred location for tests is a `tests` directory under the directory for the module. E.g. Tests for module foo: {root}/src/bin/foo/foo/tests. An alternative location is `tests` at the root of the project, although this should only be used if there are not multiple components represented in the same repository, or if the tests cross the components in the repository.

Each type of test should be in its own subdirectory of tests, to allow for easy separation. E.g. tests/unit, tests/functional, tests/integration.

Source Code Location

A standard structure for the source code places the source for each module in a module-named directory under either /src/bin or /src/lib, for executable modules and shared library modules respectively. Since each module needs its own setup.py and setup.cfg (python) that lives parallel to the top-level module (i.e. the package), the directory for the module will contain another directory named the same.

For example, Project foo, with module foo_service would have a source structure that is /src/bin/foo_service/foo_service, wherein the __init__.py for the package resides.

Sample Project Structure (Python)

Project `foo`, supporting multiple executable modules `foo_service`, `foo_cli`, and a shared module `foo_client`

```
{root of foo}
 |- /doc
 |    |- /source
 |    |- requirements.txt
 |- /etc
 |    |- /foo
 |         |- {sample files}
 |- /charts
 |    |- /foo
 |    |- /bar
 |- /images
 |    |- /foo
 |    |    |- Dockerfile
 |    |- /bar
 |         |- Dockerfile
 |- /tools
 |    |- {scripts/utilities supporting build and test}
 |- /src
 |    |- /bin
 |    |    |- /foo_service
 |    |    |    |- /foo_service
 |    |    |    |    |- __init__.py
 |    |    |    |    |- {source directories and files}
 |    |    |    |- /tests
 |    |    |    |    |- unit
 |    |    |    |    |- functional
 |    |    |    |- setup.py
 |    |    |    |- setup.cfg
 |    |    |    |- requirements.txt (and related files)
 |    |    |    |- tox.ini
 |    |    |- /foo_cli
 |    |         |- /foo_cli
 |    |         |    |- __init__.py
 |    |         |    |- {source directories and files}
 |    |         |- /tests
 |    |         |    |- unit
 |    |         |    |- functional
 |    |         |- setup.py
 |    |         |- setup.cfg
 |    |         |- requirements.txt (and related files)
 |    |         |- tox.ini
 |    |- /lib
 |         |- /foo_client
 |              |- /foo_client
 |              |    |- __init__.py
 |              |    |- {source directories and files}
 |              |- /tests
 |              |    |- unit
 |              |    |- functional
 |              |- setup.py
 |              |- setup.cfg
 |              |- requirements.txt (and related files)
 |              |- tox.ini
 |- Makefile
 |- README  (suitable for github consumption)
 |- tox.ini (primarily for the build of repository-level docs)
```

Note that this is a sample structure, and that target languages may preclude the location of some items (e.g. tests).

For those components with language or ecosystem standards contrary to this structure, ecosystem convention should prevail.

CRD Conventions

Airship will use CRDs to enrich the Kubernetes API with Airship-specific document schema. Airship projects will follow the following conventions when defining Custom Resource Definitions (CRDs).

Note that Airship integrates and consumes a number of projects from other open source communities, which may have their own style conventions, and which will therefore be reflected in Airship deployment manifests. Those fall outside the scope of these Airship guidelines.

In general, Airship will follow the Kubernetes API Conventions when defining CRDs. These cover naming conventions (such as using camelCase for key names), expected document structure, HTTP request verbs and status codes, and behavioral norms.

Exceptions or restrictions from the Kubernetes conventions are specified below; this list may grow in the future.

- The `apiGroup` (and `apiVersion`) for Airship CRDs will have values following the convention `<function>.airshipit.org`.

Documentation

Each Airship component will maintain documentation addressing two audiences:

1. Consumer documentation
2. Developer documentation

Consumer Documentation

Consumer documentation is that which is intended to be referenced by users of the component. This includes information about each of the following:

- Introduction - the purpose and charter of the software
- Features - capabilities the software has
- Usage - interaction with the software - e.g. API and CLI documentation
- Setup/Installation - how an end user would set up and run the software including system requirements
- Support - where and how a user engages support or makes change requests for the software

Developer Documentation

Developer documentation is used by developers of the software, and addresses the following topics:

- Architecture and Design - features and structure of the software
- Inline, Code, Method - documentation specific to the functions and procedures in the code
- Development Environment - explaining how a developer would need to configure a working environment for the software
- Contribution - how a developer can contribute to the software

Format

There are multiple means by which consumers and developers will read the documentation for Airship components. The two common places for Airship components are Github in the form of README and code-based documentation, and Readthedocs for more complete/formatted documentation.

Documentation that is expected to be read in Github must exist and may use either reStructuredText or Markdown. This generally would be limited to the README file at the root of the project and/or a documentation directory. The README should direct users to the published documentation location.

Documentation intended for Readthedocs will use reStructuredText, and should provide a Sphinx build of the documentation.

Finding Treasuremap

Treasuremap is a project that serves as a starting point for the larger Containerized Cloud Platform, and provides context for the Airship component projects.

Airship component projects should include the following at the top of the main/index page of their Readthedocs documentation:

Tip: {{component name}} is part of Airship, a collection of components that coordinate to form a means of configuring, deploying and maintaining a Kubernetes environment using a declarative set of yaml documents. More details on using Airship may be found by using the Treasuremap

---

Issue Tracking

Issues for the Airship Project are tracked on a per-project basis using Github Issues. All feature requests and bugs should be submitted to the respective project's Github Issues board for community evaluation and action. For additional details on a project's issue tracking workflow, see its CONTRIBUTING.md file.

Project CONTRIBUTING.md Files:

- airshipctl CONTRIBUTING.md

Submitting Issues

Issues can be submitted by navigating to a project's Github Issue page and selecting "New issue".

Depending on the project, you may be prompted to select an issue type. These selections are associated with templates to aid in the issue creation process. If there are no templates associated with the project, utilize the following templates:

- Feature Request Template
- Bug Report Template

When submitting issues, be descriptive and concise. If the issue is complex consider breaking it down into smaller issues so it can be more easily addressed by the community.

Grooming Issues

Issues are groomed by each project's core reviewers. Depending on how active the project is and their workflow, it can take up to two weeks for issues to be groomed. Issues will be marked with the "triage" label until they have been

groomed and prioritized. We encourage developers to not work on issues marked as "triage" as these issues are not guaranteed to be accepted by the project.

Each project will have a multitude of labels used to help categorize issues and improve organization. Labels will differ from project to project, but some common labels may include:

- Process labels ("ready for review", "wip", "blocked", "triage")
- Priority labels ("priority/low", "priority/medium", "priority/high")
- Component labels ("component/engine", "component/cli", etc. . .)
- Type labels ("feature", "bug", "epic")

As many applicable labels as possible should be applied to issues. These labels should be consistently reevaluated and updated as the issue evolves.

Issue Lifecycle

The general lifecycle for issues is as follows:

1. The issue is created by a member of the community on Github Issues. By default, the issues should have a type label and be labeled as "triage". Neither the submitter nor any other developer should work on the issue at this time, but they may discuss it on the Github Issue board to further clarify the issue.
2. The issue is groomed by members of the core reviewer team for the project, either on a scheduled grooming call or asynchronously on the Github Issues board. During the grooming process the core reviewer team will establish whether the issue is part of the project's scope and what the issue's priority level is. They will also apply a multitude of labels to help improve the organization of the issue board and the visibility of the issue. At the end of the process if the issue is accepted, the "triage" label will be removed and the issue will be available to be assigned and worked on. It is possible that the core reviewer team will need additional details. Please be attentive to updates on created issues to ensure they are addressed in a timely manner.
3. The issue is assigned to a member of the community to be addressed. Only core reviewers have the ability to assign issues. If a member of the community wishes to be assigned to an issue, they should make a comment on the issue requesting to have it assigned to them. Please only start work on an issue after it is assigned to you to decrease the risk of duplicate changes. Issue assignees are encouraged to use process labels to indicate where they are in the development workflow (i.e. "wip" or "ready for review"). Consistent updates let the core reviewer team know that the issue is still active. If no measurable work has been performed on an assigned issue within two weeks, the issue may be considered "stale" and the assignee may be removed from the issue.
4. Once a developer completes work on an issue's associated change (or research), they should indicate that work is complete and "ready for review". Include a link to the associated change along with any observations of worth in the issue's comments. Once the change is merged, the core reviewer team will mark the issue as completed.

Github Issues Bot

To help improve Gerrit and Github integration, Airship utilizes a python daemon to update Github Issues with Gerrit change details. The utility is hosted on Github under the name Gerrit-to-Github-Issues. Currently the bot only runs against the airshipctl project, but it may be implemented later into other projects.

The bot performs the following functions:

- Comments on issues with the information of active Gerrit changes including links, review statuses, and author information.
- Updates issue process labels with the "wip" label if "DNM" or "WIP" are present in the change's commit message and "ready for review" in all other cases.

To leverage the bot's full functionality, be sure to include a reference for the issue you are addressing from GitHub Issues inside the change commit message. There are three ways of doing this:

#. Add a statement in your commit message in the format of `Relates-To: #X`. This will add a link on issue "#X" to your change. #. Add a statement in your commit message in the format of `Closes: #X`. This will add a link on issue "#X" to your change and will close the issue when your change merges. #. Add a bracketed tag at the beginning of your commit message in the format of `[#X] <begin commit message>`. This will add a link on issue "#X" to your change. This method is considered a fallback in lieu of the other two methods.

Any issue references should be evaluated within 15 minutes of being uploaded.

For any questions, comments, or requests please reach out to Ian Pittwood either at ian-pittwood on the Airship IRC/Slack or at pittwoodian@gmail.com.

Service Logging Conventions

Airship services must provide logging, should conform to a standard logging format, and may utilize shared code to do so.

Standard Logging Format

The following is the intended format to be used when logging from Airship services. When logging from those parts that are no services, a close reasonable approximation is desired.

```
Timestamp Level RequestID ExternalContextID ModuleName(Line) Function - Message
```

Where:

- Timestamp is like `2006-02-08 22:20:02,165`, or the standard output from `%(asctime)s`
- Level is 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL', padded to 8 characters, left aligned.
- RequestID is the UUID assigned to the request in canonical 8-4-4-4-12 format.
- ExternalContextID is the UUID assigned from the external source (or generated for the same purpose), in 8-4-4-4-12 format.
- ModuleName is the name of the module or class from which the logging originates.
- Line is the line number of the logging statement
- Function is the name of the function or method from which the logging originates
- Message is the text of the message to be logged.

Example Python Logging Format

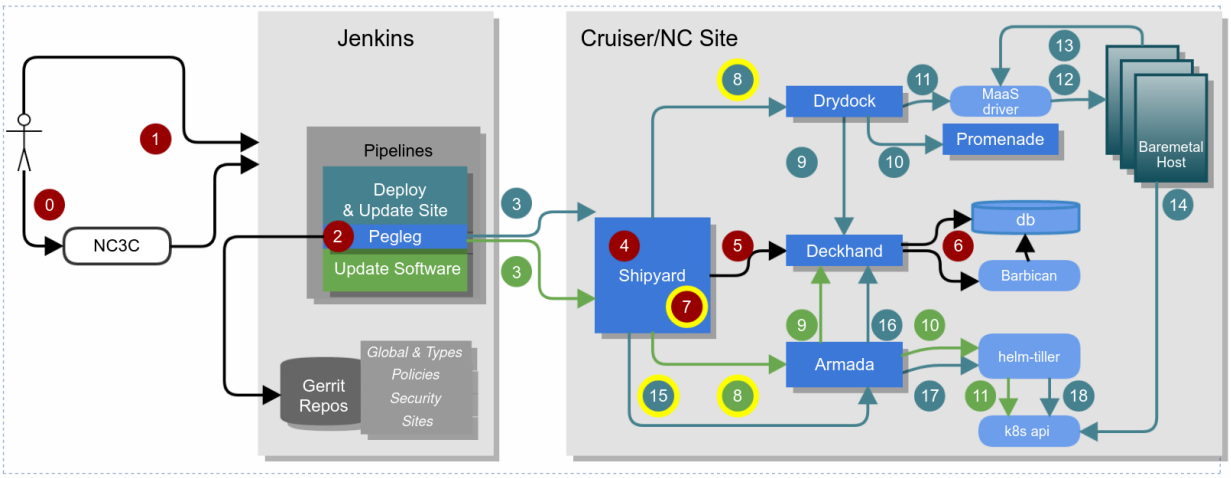
```
%(asctime)s %(levelname)-8s %(req_id)s %(external_ctx)s %(user)s %(module)s(
↪%(lineno)d) %(funcName)s - %(message)s'
```

See Python Logging for explanation of format.

Loggers in Code

Components should prefer loggers that are at the module or class level, allowing for finer grained logging control than a global logger.

1.13 Airship 1.0 Deployment Flows



1.13.1 Airship 1.0 Deploy and Update Site Flow

1. Pegleg facilitates cloning the repositories necessary to interact with a site. Each site has a single site-definition.yaml which contains the repositories that "compose" that site. These may be global repositories, type level repositories (e.g. cruisers or cloud harbor), and finally site-level repositories. These may be entirely different repositories with different permissions. Pegleg facilitates cloning all of these at the correct revisions according to the definition for that site. Pegleg can be driven via a Jenkins pipeline, which can be further abstracted in something like an NC3C dashboard, or it can be driven on the command line directly by imitating the behavior in the pipeline.

2. Pegleg wears several different hats. The CI/CD workflows leverage different pipelines in order to call upon these hats but under the hood, it's really just different command line flags on the pegleg CLI command depending on what type of action is occurring. Pegleg can:

   a. Generate (and re-generate/rotate) new secure secrets for a site according to each secret's requirement (e.g. length, type, and so on). For instance: UUIDs, passwords, keys, and so on.

   b. Encrypt secrets, and Decrypt secrets. When secrets are encrypted, they are wrapped in a YAML envelope containing metadata for each secret. This allows for understanding when secrets are going to expire, when they were last rotated, and so on. All deployment and update pipelines for instance would leverage the decrypt functionality in order to render the documents successfully.

   c. Lint the YAML to ensure it is valid and meets certain basic syntax criteria and deckhand does not have an issue processing rules encountered. For instance, development gating pipelines that validate changes to YAML would invoke pegleg in this way.

   d. Render will actually process the documents through the deckhand library, which will perform substitutions, pull in the secrets that are referenced from the configuration YAML so you can see the target document locally. This is effectively a very in-depth linting process and again would be used in development gates and potentially to fast-fail in deployment and update pipelines if there was an issue.

e. Collect will bundle up all the documents but not actually render them which is appropriate for deployment and update pipelines as it sends the documents through raw (but presumably with decrypted secrets) because each cloud site has its own deckhand instance running maintaining its own revision history capable of rendering the documents in-site. It is used in every deployment and update pipeline as the results of collect are what is sent to shipyard.

3. Once pegleg has decrypted the secrets in the document set within an ephemeral Jenkins pipeline, pegleg collect is called to assemble them all, and finally that is piped to the shipyard client which will publish them via REST API to a Shipyard API service running within the site. There are two scenarios under which Shipyard may be running in the site.

   a. On the genesis host, which is a single node running Kubernetes in a green-field site that will be expanded to a full cluster once more nodes are provisioned.

   b. On the control plane of a greenfield site, receiving a site-update or expansion.

   Simply put, the entire Shipyard workflow can be summarized as follows:

   - Initial region/site data will be passed to Shipyard from either a human operator or Jenkins
   - The data (in YAML format) will be sent to Deckhand for validation and storage
   - Shipyard will make use of the post-processed data from DeckHand to interact with Drydock.
   - Drydock will interact with Promenade to provision and deploy bare metal nodes using Ubuntu MAAS and a resilient Kubernetes cluster will be created at the end of the process
   - Once the Kubernetes clusters are up and validated to be working properly, Shipyard will interact with Armada to deploy OpenStack using OpenStack Helm
   - Once the OpenStack cluster is deployed, Shipyard will trigger a workflow to perform basic sanity health checks on the cluster

4. Shipyard will do a number of pre-validations before delivering the document set to deckhand. Things such as a concurrency check, to ensure we don't try to run updates in parallel unaware of each other. It will also run a number of fail-fast validation checks.

5. Shipyard will leverage the deckhand client library to deliver the documents to deckhand over its REST API, which will again validate them and render them (which again involves performing all layering, substitution, secret interpolation, and so on) and publishes a document revision, so that there is an on-site record of every change that has ever been requested. This document revision that is fully rendered will be available at a deckhand REST API URL that can be retrieved by various Airship sub-components.

6. Deckhand will store secrets within Barbican so that they are not stored in clear text within a database, and the rendered document set revision itself is stored directly in a database. Deckhand will change every secret to a Barbican reference which will be rendered on-demand by Deckhand whenever someone asks for that document revision through the API.

7. At this point, with the documents stored in Deckhand, Shipyard will perform another fail-fast step and ask each of the components highlighted in yellow to perform a dry-run no-op validation of the entire document set from their perspective. This means that Drydock for instance, would be validating and acknowledging it would not have any issue processing the document set it sees in Deckhand. This helps ensure we do not encounter updates that fail in the middle of the process. If a component is unhappy with the document set we want to know early and fail before making any changes.

8. Shipyard will now invoke Drydock to provision baremetal hosts that have not already been provisioned and continue to call back or poll for when Drydock has completed this process. Airship has a concept called deployment strategies because the hardware aspect of deployment is not guaranteed or reliable, and we don't always want failures here to block every other process in the stack. In other words, our deployment strategies require that 100% of nodes marked as control plane nodes must be provisioned successfully to continue, but that a certain percentage of each rack of workers could fail and we can still continue past the hardware provisioning steps successfully. In other words, this is where we introduce a threshold of failure.

9. Shipyard will send Drydock the Deckhand URL to obtain the document set for itself for this update. Drydock will retrieve the entire document set from Deckhand but it will only process documents it cares about.

10. Drydock will process any Drydock/BootAction documents that have external references in them to render those upfront before writing an operating system to the physical host. Most importantly, this allows Promenade to construct a host-specific join script. In other words, Drydock calls out to the Promenade REST API to construct a join shell script for each host and this is driven by Drydock/Bootaction documents.

11. Drydock will orchestrate MaaS based on the document set. It does this through several internal tasks, prepare_site, prepare_nodes, and deploy_site. Within prepare_site, upfront orchestration of MaaS occurs setting non-host specific settings via the MaaS API, such as CIDRs, and VLANs. Within prepare_nodes, we identify hosts that haven't already been provisioned and then power cycle hosts, wait for them to be discovered by MaaS, and then aligning and renaming them to hosts in our static inventory. Then the host configuration is orchestrated in MaaS so they have the proper networking and storage configuration as well as receive the correct static overlays, like Kubernetes join scripts, the correct Drivers, and so on, on first-boot. Finally within Drydock's deploy_nodes task we orchestrate several MaaS flows to actually provision the nodes with an operating system where they execute any additional static scripts delivered on first-boot.

12. During the deploy_nodes phase of Drydock, MaaS is effectively writing an operating system to the baremetal nodes.

13. Driven by cloud-init on first boot post provision, the nodes will actually make a rest call back to the MaaS API to inform it that provisioning has completed and they have successfully booted up into functional networking and have booted up successfully. Drydock can use this status within MaaS to understand the nodes were provisioned successfully.

14. The nodes run the Promenade generated shell script to join them to Kubernetes. This host-specific script installs the appropriate dependencies and joins the node as a Kubernetes node, either as a worker, or as a control plane host depending on the hosts profile in the YAML inventory.

15. Shipyard has been polling Drydock for completion of processing the site update. Once the polling for Drydock provisioning completes, Shipyard will move on to performing a similar request to Armada. Armada is asked to update the site and given a Deckhand URL and revision to pull from.

16. Armada pulls the rendered document set from Deckhand.

17. Armada then proceeds to help orchestrate any helm installs or upgrades necessary in the site, and helps do this across a vast number of charts, their ordering, and dependencies. Armada also supports fetching Helm chart source and then building charts from source from various local and remote locations, such as Git endpoints, tarballs or local directories. It will also give the operator some indication of what is about to change by assisting

with diffs for both values, values overrides, and actual template changes. Its functionality extends beyond Helm, assisting in interacting with Kubernetes directly to perform basic pre- and post-steps, such as removing completed or failed jobs, running backup jobs, blocking on chart readiness, or deleting resources that do not support upgrades. However, primarily, it is an interface to support orchestrating Helm.

18. Armada effectively interacts with Tiller for installation (although it may interact with k8s directly to poll, wait, remove jobs, and otherwise help protect helm from failures). Tiller will then interact with k8s to perform helm chart installations or upgrades.

1.13.2 Airship 1.0 Update Software Flow

The Update Software flow (or "action" in Shipyard – depicted with green numbers in the image) is effectively a subset of the above flow. It is used primarily to speed the process up by bypassing the Drydock flow entirely. The reason for this is both speed as interacting with MaaS is slow, as well as times where you want to avoid trying to process hardware requests (e.g. waiting for Drydock to try and provision a piece of failed hardware only to ultimately timeout some time later before moving on to the next step because the deployment strategy allows it).

1.13.3 Further Documentation

- https://airshipit.readthedocs.io/projects/shipyard/en/latest/
- https://airshipit.readthedocs.io/projects/pegleg/en/latest/
- https://airshipit.readthedocs.io/projects/armada/en/latest/
- https://airshipit.readthedocs.io/projects/promenade/en/latest/
- https://airshipit.readthedocs.io/projects/drydock/en/latest/
- https://airshipit.readthedocs.io/projects/deckhand/en/latest/
- https://airshipit.readthedocs.io/en/latest/

1.14 Other Resources

- Airship Blog
- Airship Website
- Airship Wiki

CHAPTER

TWO

GET INVOLVED

2.1 Join our mailing lists

Receive Airship announcements and interact with our community on our mailing lists.

2.2 Join our weekly calls

Airship is constantly evolving. Contribute to the Airship design process and day-to-day community operations in our weekly calls.

2.3 Join our Slack workspace

Get in touch with Airship developers and operators in our Slack workspace.

INDEX