The Nephio Approach

In Episode 2, we closed with the statement that we would follow three basic principles to address the complexity of managing infrastructure and workloads for distributed clouds:

- Consolidate on a single, unified platform for automation
- Declarative configuration with active reconciliation to support day one and day two.
- Configuration that can be cooperatively managed by machines and humans.

The first of these - a uniform platform for automation - enables the same tooling to work across the three "swimlanes" that are in scope for Nephio. These are infrastructure, workloads (function modules), and workload configurations. All three of these tend to be interrelated, because the end goal is to meet some user requirement. Those requirements dictate the specific configurations of the workloads (for example, the capacity requirements), as well as the specific functionality needed (and therefore which workloads we deploy). The workloads in turn make demands on the infrastructure for specific networking, compute, and storage capacity and connectivity.

It is prohibitively expensive to build automation that can respond to changes in user requirements or the deployment context when different systems are used to manage each of these areas, each with their own data format and set of APIs, and each of which may vary from customer to customer. Of course, it is also prohibitively expensive to rebuild all of the existing tooling to manage each layer of the stack, and that is not what Nephio proposes. Instead, we are proposing to use Kubernetes as this automation layer, and use its flexible architecture to integrate with existing management platforms.

While container orchestration is the use case that birthed Kubernetes, the architecture and API model in Kubernetes is general purpose. This has been proven out over the last several years by the development of hundreds of different "operators", or Kubernetes-based controllers for managing specific types of workloads. "Wrapping" underlying management APIs in well designed, declarative Kubernetes APIs can achieve the uniformity we need without requiring wholesale replacement of existing platforms.

Kubernetes also provides a proven basis for our second pillar - declarative management. Declarative management is especially important when we move past the initial deployment and on to "day 2" operations. Imperative systems such as workloads start from a well-known state - typically the "empty" state before anything has been done. In the cloud world, where we have on-demand, API-driven access to resources, we have an opportunity to build systems that instead respond to changes in the state of the world around them. Rather than starting from one of a few simple, predefined states, declarative systems continuously evaluate the current state, determine the steps to realize the intended state, and then reconcile those states by modifying the current state. Kubernetes does this today at the individual workload or cluster level; Nephio intends to bring this same self-healing, autoscaling, intent-driven active reconciliation for the overall topology of edge workloads. If well executed, these techniques obviate the need for imperative workflows, instead building the intelligence to map user-level intent into resource-level configuration, and continuously assure the realization of that intent, into software running at different layers of the control plane.

Our last pillar comes from the realization that these high-scale edge systems are too large and complex for any single person or team to manage. Instead, they are born of the cooperation of many teams. Unfortunately, our existing tooling, such as Helm and other templating engines, assume a single "starting point", where all inputs are known. This leaves the problem of figuring out all those inputs - and coordinating among the many teams that provide them - as "an exercise for the reader".

GitOps begins to address this problem by providing a common system (git), where teams can operate on "draft" versions of their infrastructure and workload specifications, prior to publishing them to the Kubernetes API server. However, it doesn't go quite far enough, because the contents of the git repositories is still a complete free-for-all; it provides an unstructured data store. It is still left up to the organization to impose any structure on it.

Nephio embraces an approach called "Configuration-as-Data" (CaD), which extends standard gitOps with a few additional principles. CaD:

1. Makes configuration data in versioned storage (e.g., git) the source of truth (this is standard gitOps)
2. Uses a uniform, serializable data model (KRM) to represent configuration (infra and workload specifications)
3. Separates code that acts on the configuration from the data
4. Provides APIs so that clients manipulating configuration data do not directly interact with storage

By separating code from data, we enable building reusable tools that can operate on any workload specification. In our existing templating systems, a workload package such as a Helm chart is an abstraction. It is intended to be an opaque box that the user cannot peer into and modify; you simply provide the inputs, and it spits out the workload specification. This means that every single package has its own, bespoke API - in Helm this is the values.yaml file. If we want to build automation, we need to build them on top of each of those specific package APIs. With CaD, the package is instead an open package, populated by well-structured KRM entities. Every package consists of the same resources, simply arranged differently. Thus, rather than building automation that needs to understand many thousands of individual, bespoke package APIs, we can build them to understand the orders-of-magnitude fewer types of resources. As an added benefit, those resource APIs include validation schemas, versioning, and planned deprecation policies, bringing stability to the automation built on top of them.

Some of computer science's biggest success stories rely on a similar separation of data and code. Consider the transition from mainframe-style, bespoke fixed file formats to simpler, Unix stream-based formats. Modeling files as simple streams enabled the creation of universally useful tools such as sed and Unix pipelines. Adding a little additional structure in the form of line breaks expanded tools to functions such as vi, wc and awk. Adding column breaks suddenly enabled Structured Query Language (SQL), inducing rapid advancement of database management systems and unlocking trillions of dollars in value. The CaD approach takes lessons from these previous successes.
Nephio believes these approaches enable our mission - to "materially simplify the deployment and management of multi-vendor cloud infrastructure and network functions across large scale edge deployments". We have many more challenges and lessons to learn as we attempt to achieve that mission, and careful and competent execution will be key to our success. We hope that you will join us and contribute to tackling these challenges.

About the Nephio Community

Nephio is an open source project of the Linux Foundation. We do our best to be a friendly, welcoming community. We have several different "special interest groups" (SIGs) that meet regularly, as well as mailing lists and a Slack instance. Please join us!

- Learning page - https://nephio.org/learn
- Site - https://nephio.org/, https://nephio.org/about/
- Wiki - https://wiki.nephio.org/
- Docs - https://github.com/nephio-project/docs
- Blog Postings - https://nephio.org/blog/
- Slack - https://nephio.slack.com/ (join via this invitation)
- Project Github - https://github.com/nephio-project
- Project email distro - https://lists.nephio.org
  - nephio-tsc (for TSC members and interested parties)
  - nephio-dev (for all)
- SIG lists: sig-netarch, sig-automation, sig-release