nethopper.io

What Happened to Platform Engineering?

eBook

By Daniel P. Donahue
@ddonahuex | 𝕏

Author

Daniel P. Donahue

Dan is Nethopper's Principal Solutions Architect. He defines himself as a startup veteran, contributing at various levels, including architecture, leadership, and software development. Dan received his first patent in 2022. Prior to Nethopper, Dan held leadership and technical roles at Parallel Wireless, Juniper Networks, and RiverDelta Networks, and other early stage startup companies.

nethopper.io

Table of Contents

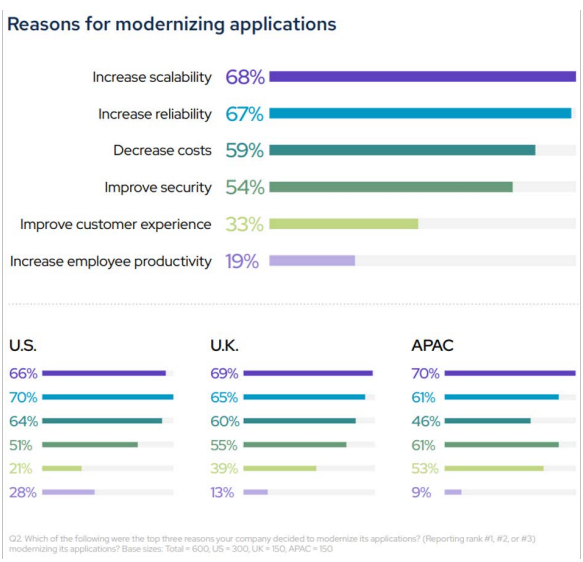
1. EXECUTIVE SUMMARY

The Rush to App Modernization

What Happened to Platform Engineering?

The *purpose* of this ebook is to define what happened to Platform Engineering (PE) in the frenzied push for application modernization that has left many organizations mired in their cloud native initiatives.

The *goal* of this ebook is to provide a solution that gives organizations the ability to jumpstart their platform engineering efforts, and enable their business applications.



2. INTRODUCTION

Platform Engineering

Bits, Bytes, and Nuances

I understand most technical white papers are supposed to be lots of bits and bytes and devoid of any personal nuances, but this one is going to be different.

While the bits and bytes will certainly be present, they will be mixed with personal experiences, because, by nature, I am a Platform Software Engineer. It's what I've done for almost 25 years. I am very passionate about it.

Much of what I've read recently in blogs, social media posts, and books compelled me to write this ebook, because Platform Engineering (PE) seems to be a new concept to many. It is not. Platform engineering simply moved from Software Engineering to DevOps/IT.

Therein lies the awakening of DevOps/IT teams to platform engineering and the challenge it presents for them.

In his book entitled **Platform Engineering on Kubernetes,** which I highly recommend, author Mauricio Salatino makes a similar statement regarding the platform engineering term. Mauricio says:

> ***Platform engineering is not a new term in the tech industry. But it is quite new in the cloud native space and the context of Kubernetes. We were not using the term in the cloud native communities when I started writing this book in back in 2020.[1]***

2. INTRODUCTION

In 2013, I was hired by a startup to architect, develop, and deliver platform software for 4G and 5G mobile network solutions. In that role, like many of my startup roles before it, I was expected to, and did develop platform software from scratch to support the core application software (mobile networking protocol stacks) running in remote radio heads (RRH) and COTS server running in the network core.

In late 2018, the same startup company mandated both the RRH and COTS servers move to cloud native, or in other words, Kubernetes. The deliverables of my role changed from architecting proprietary platform software to architecting solutions based on lots of open-source (OSS) tools and applications.

Rearchitecting the core software application out of the monolithic images and into containerized microservices-based images was the easy part. Replacing the infrastructure support provided by the proprietary platform software proved to be the most difficult.

I was dragged kicking and screaming into that effort and subsequently into the Kubernetes world. I was required to develop a platform solution that supported the new microservices based architecture. I didn't like it.

At first, I was frustrated sifting through the plethora of information available for learning Kubernetes and other CNCF (Cloud Native Computing Foundation) projects. It was initially overwhelming, but in time I found the engineering principles required to develop cloud native platforms were not so different from developing proprietary ones and, so, I made my peace with Kubernetes.

This ebook is an attempt at articulating my journey in such a way that helps others on their cloud native platform engineering journey.

3. THE APP MODERNIZATION SHIFT

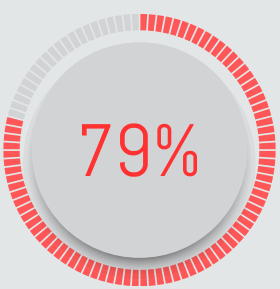
Shift to Cloud Native

From Monolithic to Containerized Microservices

Call it app modernization, digital transformation, cloud native, or any other catchy industry phase for the shift to Kubernetes.

In this shift to cloud native development, the industry has moved from delivering monolithic application images to delivering containerized microservices based images.

It has been estimated 79% of application modernization efforts fail.[2] The failure can be attributed to several reasons.



OF APP MODERNIZATION PROJECTS FAIL

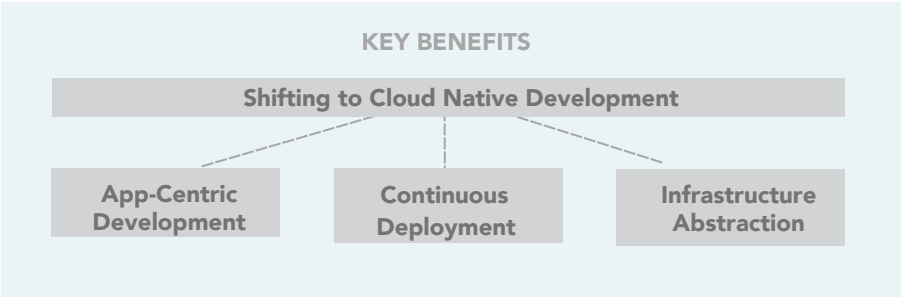
- Inadequate skills due to the global lack of cloud native talent.
- Lack of intelligent tools.
- Lots of open source projects, but they come with no support.
- The effort is often underestimated from a complexity, timeline, and budget perspective.

3. THE APP MODERNIZATION SHIFT

Monolithic images contain everything needed to support the core application/s including, the OS (sometimes) proprietary platform infrastructure software, OSS and proprietary libraries required by the OS, platform software, and core applications. By contrast, containerized images contain the core application and only the OSS tools and packages needed by the application.

A monolithic image can sometimes be refactored into several containerized images as was the case of my previous startup as mentioned in the Introduction. Refactoring monolithic applications into containerized microservices enables software developers to deliver their containerized applications with no dependencies on other domains co-resident in a monolithic image. This means in the cases where application specific teams, who have no external dependencies on features from other teams, can deliver features at their own cadence. When the image is delivered to production, Kubernetes will update only the pod/s associated with the image.

There are several benefits to shifting to cloud native development, but we will focus on three of them: app-centric development, continuous deployment, and infrastructure abstraction.



3.1 APP CENTRIC DEVELOPMENT

App-Centric Development Requires a Framework Like Kubernetes to Succeed

Hyper Focus On the Core Application(s)

The app-centric development model enables organizations to hyper focus on the core application/s that drive their business. For example, in the *Introduction* I mentioned my previous role of delivering proprietary platform software for a mobile network solution. That company was not in the platform software business. Their core applications revolved around mobile networking protocol stacks, physical layer interfaces, and core network functionality.

In their shift to cloud native, the core applications were re-factored into containerized image artifacts. My team's and my proprietary platform software were not.

Kubernetes would now be handling the launching, monitoring, security, and upgrading of the applications. OSS projects like fluentd, Prometheus, and others would handle FCAPS (Fault, Configuration, Accounting, Performance, Security) previously handled by the proprietary platform software.

> ***What does the shift to app-centric development mean for organizations delivering applications?***

Product led organizations no longer must concern themselves (or their software engineers) with proprietary platform software and software requiring intimate knowledge of system hardware requirements.

The Kubernetes framework handles many of the functions of proprietary platform software. Kubernetes provides an abstraction layer for the management of compute, network, and storage. Kubernetes also manages the application (pod) lifecycle. No need to write Linux daemons for your applications anymore. Write an application manifest and give it to the Kubernetes API using kubectl and your application pod is deployed and monitored. If the pod encounters an error or terminates, then Kubernetes will restart it.

3.2 CONTINUOUS DEPLOYMENT

A Full-Featured Platform Is Required to Do Continuous Delivery

CI/CD Pipelines

As a software engineer and Platform team lead, I've been part of many CI/CD initiatives. In all of them, we got the CI and CD (Continuous Delivery) part right, but somehow the Continuous Deployment got left behind entirely.

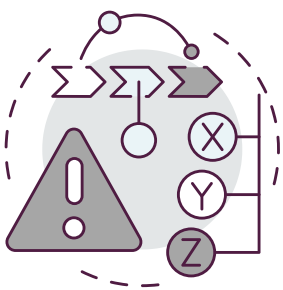
Before moving on, I should explain the difference between Continuous Delivery and Continuous Deployment.

- Continuous Delivery essentially automates the software development process up to the point of deployment to Production, which is done manually.
- Continuous Deployment can automatically deploy to Production.

I know the term "automatically deploying to Production" immediately gives agita to everyone from the C-suite to the engineer committing the code. As stated previously Continuous Deployment can automatically deploy to Production.

A well architected platform provides that ability but also makes it optional (enabling manual update) or elegant using various rollout techniques (canary, blue/green, etc.).

The Continuous Delivery model is typically handled by an automation server like Jenkins. I know the Jenkins users will write telling me I am wrong, that Jenkins can do Continuous Deployment as well. Maybe, but it is not elegant, it is certainly not production grade because Jenkins is a software development tool, an automation server – not a platform. The result is Continuous Delivery is left behind in the push to create CI/CD pipelines.



3.2 CONTINUOUS DEPLOYMENT (Cont.)

At Nethopper, I've met with many organizations and pro/managed services providers and realized my experience is not unique. Many organizations are struggling with Continuous Deployment.

What is the common thread?

The platform required to manage the cloud native ecosystem application deployment and monitoring, infrastructure automation, observability, etc., does not exist.

They are left with a platform choice: build, manual operation, or buy:

- ***BUILD:*** Building it themselves is risky. The time and expense put forth is not a guarantee of success.
- ***MANUAL OPERATIONS***: If the choice is manual management, then Ops teams must use kubectl commands on every cluster to manage applications. That does not scale.
- ***BUY:*** The best option is to buy, but what should be the criteria for choosing a platform?

We'll get to that in a bit.

> ***The platform required to manage the cloud native ecosystem application deployment and monitoring, infrastructure automation, observability, etc. does not exist.***

3.3 INFRASTRUCTURE ABSTRACTION

Solving Infrastructure Abstraction for Cloud Native Architectures and Enabling Devs to Develop and Ops to Operate

Migration Path

The migration path of software platforms over the last couple of decades (post-mainframe) went from embedded, to COTS servers, VMs, and now cloud native. One of the main benefits in moving to a cloud native software architecture using Kubernetes is the compute, network, and storage are abstracted from the application.

The software architect, not the application developers, needs only to define the compute, network, and storage requirements for their application and the public cloud providers ostensibly handle the rest. Therein lies the problem.

The former platforms, embedded/ COTS servers, and many times VMs, were defined, pre-configured, pre-installed with app software by either engineering or operations teams. The burden of building clusters has now moved to DevOps, which in many companies, from a skills perspective, is ill-equipped to handle it. DevOps, Central IT, now has choices to make.

Do we upskill our team? Do we hire new talent? Upskilling takes time with no guaranteed outcome. Hiring the right talent is elusive and expensive when found. One CIO told me finding such a person is nearly impossible and when you do they want more money than the CEO.

Building and provisioning infrastructure results in one of two scenarios:

- The first scenario is the infrastructure burden falls back to engineering as opposed to central IT. The development experience promised by a cloud native architecture of being app-centric turns out to be false. Developers now must learn Kubernetes, Infrastructure-as-Code (IaC), etc.

- The other scenario is the DevOps teams must now develop a platform for building infrastructure, deploying applications, observability for the infrastructure and applications.

3.3 INFRASTRUCTURE ABSTRACTION (Cont.)

In either case, the net result is either Devs doing Ops work or Ops folks doing Dev work. In both cases, nobody is happy, and the organization is not functioning well.

Is there a solution?

The answer to solving the infrastructure abstraction reality for cloud native architectures is an IDP (Internal Developer Platform) that enables devs to dev and ops to op.

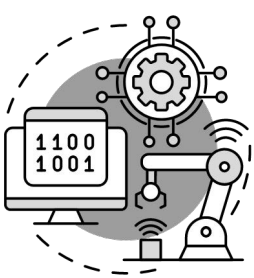
An IDP provides a framework the Ops team use to create an operational cloud native ecosystem that serves all contexts for an organization's applications.

A typical ecosystem has the following deployment contexts: Dev, QA, Staging, and Prod.

An IDP should be able to build infrastructure, deploy and observe the applications using existing CI/CD pipelines to achieve true Continuous Deployment for each of the contexts in a consistent manner.

A recent article on thenewstack.io cited a poll that said:

> ***"We found that virtually all (99%) of the engineering leaders we surveyed had begun using platform engineering in their organizations, with 53% reporting that they've begun [2023]. Meanwhile, the vast majority (85%) of respondents said they'd either started implementing internal developer portals or were planning to do so in the next year."***



4. IDP ARCHITECTURE

Internal Developer Platform (IDP)

Cloud Native IDP

An IDP is required to build and manage infrastructure (both private and public clouds) using IaC (Infrastructure-as-Code), deploy applications, automate CI/CD pipelines, while providing observability for all those domains.

There are two architectural components required for a cloud native IDP: a DevOps framework and secure multi-cluster networking.

- A DevOps framework that supports a cloud native IDP is the first architectural decision platform engineers need to make. GitOps is widely considered the foundational element of an IDP architecture.

- An IDP that manages an ecosystem of clusters requires communication between clusters. This requires multi-cluster networking, which must be secure and not needlessly complex.

Kubernetes provides simple intra-cluster communication but does not natively support inter-cluster communication.

Notice I didn't say multi-cloud. Multi-cloud is an industry term that typically refers to support for the big cloud providers (AWS, GCP, and Azure).

Multi-cluster networking is a technical term used in this document referring to communication between any variation of clusters, whether private (on-prem) or public. Multi-cluster can also be referred to as hybrid clouds.

These two architectural components are detailed in the following subsections.

4.1 GITOPS

Git Repositories Are the Declarative Source of Truth for the IDP

DevOps Is a philosophy, Not a Framework

GitOps is a framework applicable to DevOps philosophies. GitOps is defined by GitLab as:

> ***GitOps is an operational framework that takes DevOps best practices used for application development such as version control, collaboration, compliance, and CI/CD, and applies them to infrastructure automation.[3]***

GitOps by nature is both declarative and auditable. Declarative in the sense that what is defined in git is what is instantiated. GitOps is auditable because all git actions in repositories leave an immutable audit trail of who changed what and when.

In GitOps, git repositories are the declarative source of truth for the IDP.

An IDP that utilizes GitOps builds (infra), deploys (apps), and manages (CI/CD pipelines) what is declared in git.

When used for IaC, for example, GitOps can control cloud spend by mitigating infrastructure drift. The next section details this further.

4.1.1 BUILDING INFRASTRUCTURE WITH GITOPS

Infrastructure built using GitOps mitigates infrastructure drift.

Having a declarative source of truth becomes particularly important when using IaC to build public and private (on-prem) clusters. Having your infrastructure declared in git completely mitigates infrastructure drift. Nothing changes unless something changes in git. Controlling infrastructure drift using GitOps controlled IaC enables organizations to control infrastructure costs. A GitOps-based IDP provides organizations with the ability to definitively control cloud spend. Consider the following example.

NOTE: This following example will become clearer after reading the KAOPS section below, but for now, the important point is the infrastructure built using IaC and GitOps will not change unless something in the source of truth changes.

4.1.1.1 BUILDING INFRASTRUCTURE WITH GITOPS

Example: Mitigating Infrastructure Drift

Let's say I built a 2 node EKS cluster in AWS using GitOps and then co-worker with IAM rights logs into the AWS console and changes the node count to 4. My compute costs for that cluster essentially doubled. The AWS console would allow the change (as it should) and begin to build the additional two nodes. The cluster configuration in AWS changed, but it did not change in the source repository where the node count is still 2. The IaC feature of the IDP controlled by GitOps will revert that EKS cluster back to 2 nodes because the source of truth did not change, and costs do not increase. Drift mitigated, but the application, now expecting 4 nodes, will likely break. Building infrastructure with GitOps avoids app disruption.

4.1.2 DEPLOYING APPS WITH GITOPS

A GitOps-based IDP gives you a declarative method for upgrading and rolling back application versions.

Cloud native applications are defined and deployed using Kubernetes manifest files. A manifest file is a text-based schema, typically written in YAML, to define the desired state of a Kubernetes object. An application is a Kubernetes object.

A manifest file contains an image digest. An image digest contains the version of the application image to be deployed. Using a GitOps approach an application's manifest file would be stored in a git repository. The GitOps based IDP will monitor the manifest in the repository and deploy whatever is declared in git. This GitOps approach means an application can be upgraded or rolled back simply by changing the version in the image digest of the manifest file stored in git.

To upgrade or rollback a deployed application the image version is changed in the application's source manifest file and committed and merged to the repository branch being monitored by GitOps.

For example, the manifest snippet below would create a pod running MongoDB version 5.0. If the manifest is changed to 5.1, then a new pod running 5.1 would be created and the 5.0 one terminated. If unhappy with the change, then simply change the manifest back to 5.0 and the upgrade would be rolled back.

Current:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-prim
spec:
    containers:
      image: mongo:5.0
...
```

Upgrade:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-prim
spec:
    containers:
      image: mongo:5.1
...
```

4.1.3 GITOPS AND BUSINESS MANAGEMENT TOOLS

The entire sequence from developer commit to production change would have a complete audit trail of changes and approvals.

GitOps functions (e.g. git merge) are easily integrated into workflow features provided by business management tools like JIRA and ServiceNow.

Let's consider a JIRA workflow example that controls IaC changes in production.

An IaC developer wants to change a production AWS EKS cluster node count from 2 to 4 because more compute is required. The developer commits the change to his/her branch and issues a pull request (merge developer branch to production branch). A pull request must be approved before being merged to a target branch (e.g. production).

A JIRA workflow can be easily developed to do an "under the hood" pull request approval and merge (or not) so the change can be realized in production (or not).

The JIRA approver need not know what a pull request is or how to perform one to affect the production change. Once approved, the JIRA workflow approves the pull request, merges the change to the production branch, and the node count would be changed from 2 to 4. The entire sequence from developer commit to production change would have a complete audit trail of changes and approvals.

The approach used in the infrastructure example above applies to application upgrades and rollbacks as well. The same JIRA workflows could be applied to repositories containing the production application manifests. Only approved approvers can sign off on production changes.

4.1.4 NOT JUST FOR PRODUCTION

Improving the DevEx greatly reduces organizational friction because it enables devs to dev and ops to op.

The examples in the sections above focus on a production environment. The great thing about a GitOps-based IDP is that it applies to any context: Dev, QA, and Staging. A production environment can easily be reproduced in-house for blue-green testing, customer debug scenarios, etc.

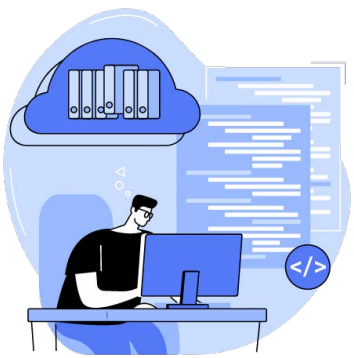
Platform Engineers and DevOps teams that choose a GitOps-based IDP can create and manage consistent environments across an organization that fosters a developer friendly solution by removing the requirement for developers to manage cloud native infrastructure to deploy and test their changes.

4.2 MULTI-CLUSTER NETWORKING

Abstracting the Complexities of an Overlay Network, Connecting Clusters, and Forming Virtual Application Networks

Multi-Cluster Networking

Multi-Cluster networking is a foundation requirement of a cloud native IDP because things like GitOps and Observability require connectivity between clusters.



For managing a network of clusters, a Layer 7 overlay network should be considered. An overlay network is a virtual (or logical) network that exists on top of a physical network. The Internet is an example of an overlay network. Building Layer 3/4 network connectivity between clusters is complex and costly because it typically requires multiple domain experts (network, firewall, IP gateways, etc.) at ISO layers 3 and 4.

An IDP that uses a secure overlay network can quickly connect any cluster and effectively build a Kubernetes control plane that can be called a Virtual Application Network.

5. KAOPS - A MODERN IDP FRAMEWORK

Nethopper KAOPS

GitOps and Multi-Clustering Networking

In the Introduction, I mentioned how fragmenting the monolithic image into microservices was the easy part. Deployment and orchestration of our applications proved difficult.

In the search for a managed Kubernetes platform, I found and was introduced to Nethopper and KAOPS. I liked KAOPS so much that I joined the company.

As mentioned previously a cloud native IDP requires GitOps and Multi-Clustering Networking.

Before diving into the architecture details, we'll highlight the OSS projects KAOPS integrates to satisfy those requirements.

KAOPS stands for Kubernetes Application Operations Platform as a Service.

KAOPS provides value in two key areas:

- Simplification of app
- Infrastructure management and extensibility.

The simplicity of KAOPS enables organizations to rapidly come up to speed on managing and observing applications and infrastructure. No need to look for a unicorn cloud native expert that may create an esoteric solution that is unusable once he/she leaves the company. KAOPS is extensible via git.

More on that in a bit.

5. KAOPS - A MODERN IDP FRAMEWORK (Cont.)

Consider the following from InfoQ's July 2023 DevOps and Cloud Trends Report:

> ***Platform engineering is evolving toward simplification and value delivery, adopting a platform-as-a-service mindset. The role of platform engineering teams is shifting from complex infrastructure management to becoming service providers focused on user satisfaction and value creation. Observability, financial aspects, and sustainability considerations are becoming integral to platform engineering.***

KAOPS is a cloud provider and Kubernetes agnostic that provides enterprise level support for all integrated OSS projects and greatly abstracts their complexities.

KAOPS users need not take the time to upskill themselves by learning the intimate details of the projects like ArgoCD or concern themselves with their release trains. KAOPS integrates stable and production tested versions of each OSS project.

The first step in architecting a cloud native environment is selecting the infrastructure. Infrastructure choices include public cloud providers, VMWare, on-prem options like Openstack and others. Once the infrastructure is chosen the next choice becomes which Kubernetes distribution to use. In most cases, that decision is made for you by the cloud providers, EKS for AWS, AKS for Azure, GKE for Google, Openshift for Openstack, etc.

KAOPS is the only IDP that does not force a Kubernetes distribution upon you. This is intentional. Providing an IDP that is cloud and Kubernetes agnostic makes your IDP future-proof. Application migration across cloud providers and on-prem clusters is seamless. Beware of any IDP that comes with a Kubernetes distribution.

The following illustration depicts how KAOPS sits on top of any Kubernetes distribution and the currently integrated OSS projects and their domain.

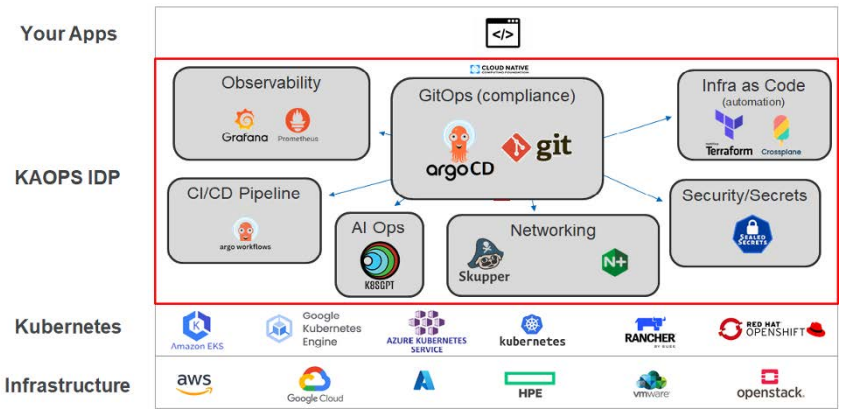


Figure 1 - KAOPS Ecosystem

5.1 THE OPINIONATED PATH

KAOPS: A Foundation for Platform Engineers to Build Their IDP

GitOps-based IDP Framework

KAOPS is Nethopper's GitOps-based IDP framework that provides a foundation for platform engineers to build their internal developer platform.

It takes an opinionated approach to GitOps and provides an extensible and configurable way for platform engineers to build and manage infrastructure, manage secrets, deploy applications, AIOps, and observe it all.

KAOPS integrates:

- ArgoCD for GitOps management
- Skupper for multi-cluster networking
- Other OSS projects for IaC, Observability, AIOps, etc.

These projects are detailed later in this section.

KAOPS is a modern IDP that enables platform engineers to jumpstart their own IDP by providing an extensible GitOps-based multi-cluster networking platform.

5.1.1 ARGO CD FOR GITOPS

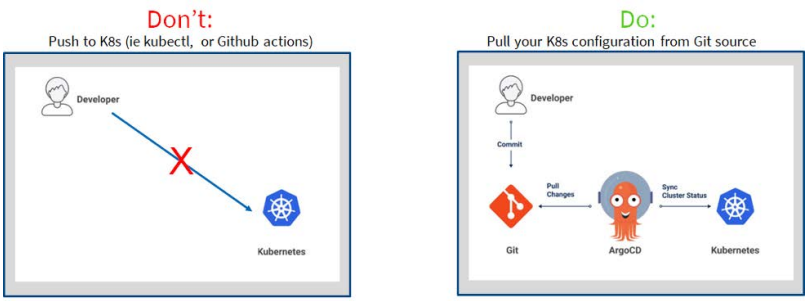
ArgoCD is a declarative, GitOps continuous delivery tool for Kubernetes.

ArgoCD is a CNCF project that achieved Graduated status in December 2022. Graduated simply means the project is considered mature, proven, and widely adopted. What is ArgoCD? ArgoCD is a declarative, GitOps continuous delivery tool for Kubernetes[4].

Most Kubernetes deployments are managed using a push model. An Ops person uses kubectl to deploy or modify applications. This manual process is labor intensive and error prone, especially in a production environment. ArgoCD uses an automated pull model.

The main purpose of ArgoCD is to ensure the live state of an application matches the desired state of an application. The live state is the status of the application running in a pod. The desired state is the application resource manifest declared in a git repository. ArgoCD accomplishes this task by continually comparing the live state against the desired state. If ArgoCD detects any difference between the two, then the application is redeployed according to the resource manifest in git.

In summary, pull, don't push.



5.1.1 ARGOCD FOR GITOPS (Cont.)

The following diagram depicts the ArgoCD architecture.

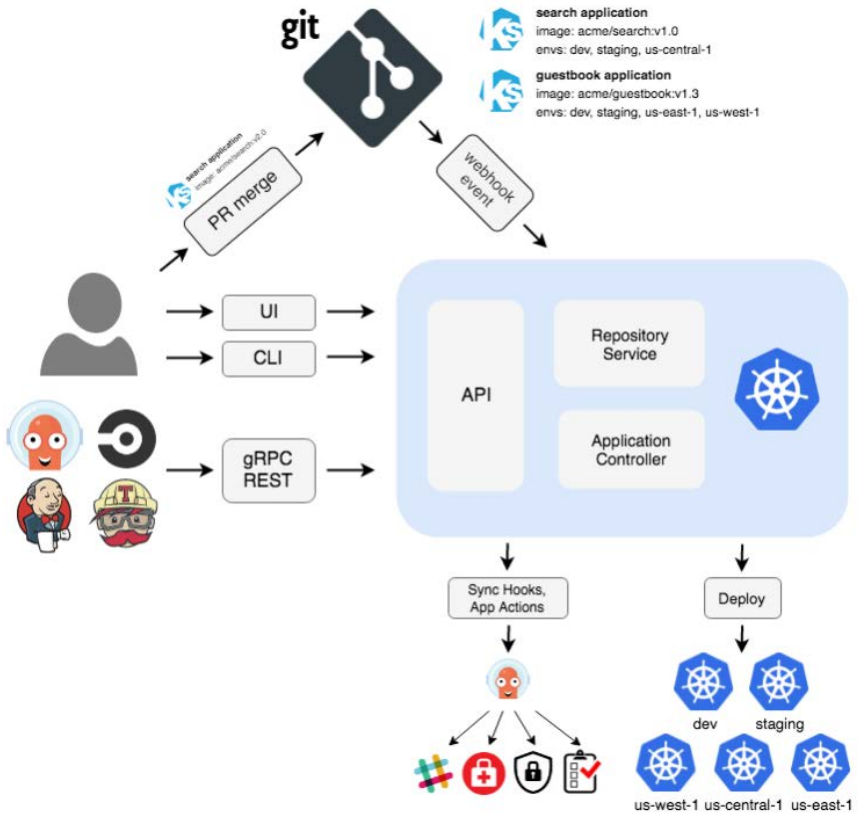


Figure 2 - ArgoCD Architecture

4.1.1.1 COMPLEXITIES ABSTRACTED

KAOPS abstracts a lot of the complexities of ArgoCD (e.g. ApplicationSet) so platform engineers need not spend the time to learn its intimate details. KAOPS also uses multi-cluster networking in a way that enables a single ArgoCD server to manage multiple clusters in a virtual application network.

ArgoCD complexities like ApplicationSets are abstracted by KAOPS using simple Distribution Rules. KAOPS users apply tags to clusters and applications and then create simple rules that KAOPS uses to determine which application should go to which clusters.

Any application tagged “foo” should go to any cluster tagged “bar”.

KAOPS will create ApplicationSets in the cluster running the instance of the ArgoCD server and ArgoCD will distribute the application accordingly. An application can be any Kubernetes object, including a core application manifest, tool manifest, helm chart, etc.

5.1.2 SKUPPER FOR MULTI-CLUSTER NETWORKING

Skupper enables cluster-to-cluster communication.

Skupper is the best project you've probably haven't heard about.

It's an OSS project from Red Hat used by KAOPS that enables secure cluster-to-cluster communication by creating a Virtual Application Network (VAN).

A VAN consists of a Hub cluster and Edge clusters. The Hub cluster is the one cluster in a VAN that is reachable to Edge Clusters. To enable reachability, a Hub cluster is configured using either Ingress, Load Balancer, or NodePort. Edge clusters only require Port 443 to be opened in the outbound direction to attach to the hub.

> ***KAOPS creates and uses a VAN to create an IDP control plane used by ArgoCD and other OSS projects to manage a group of clusters.***

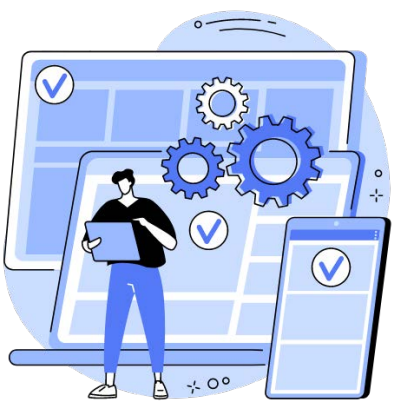
5.2 THE GOLDEN PATH: INTEGRATED OSS PROJECTS

KAOPS: An Extensible, Yet Opinionated Approach to the Golden Path

GitOps and Multi-Cluster Networking

As mentioned previously, KAOPS is only opinionated when it comes to GitOps and Multi-Cluster Networking.

KAOPS remains somewhat opinionated about other OSS projects used for IaC, Observability, Secrets, CI/CD workflow automation, and AIOps. We call this extensible yet opinionated approach: The Golden Path.



5.2.1 INTEGRATED PROJECTS

Golden path comprised of chosen Open Source Software as best-practices.

The golden path in KAOPS is comprised of OSS tools that were chosen as best practices in collaboration with Nethopper customers and partners. They are fully integrated into the platform and come with enterprise-grade support.

The following subsections briefly describe each of the OSS projects shown in the KAOPS Ecosystem figure above. The OSS projects below are enabled by default when creating a VAN. They are all optional and can be deselected when creating a VAN.

5.2.1.1 CROSSPLANE

Crossplane is used by KAOPS for IaC. It has native support for Terraform, AWS, Azure, GCP, and Kubernetes. No need to throw away existing Terraform. Write a simple Crossplane manifest, point it to your Terraform repository, configure it in KAOPS, and KAOPS will build it for you. Not only will KAOPS build it for you, but it will ensure the infrastructure declared in git is what is built. No infrastructure drift.

5.2.1.2 PROMETHEUS AND GRAFANA

Grafana and Prometheus are used for Observability. When enabled (default), KAOPS will install a Prometheus server in every cluster in a VAN. A single Grafana instance will be installed in the VAN's hub cluster. The single instance of Grafana can collect and display KPIs from any cluster in the VAN resulting in a single pane of glass for Observability.

KAOPS comes stocked with 28 Grafana dashboards and can easily integrate user dashboards.

SealedSecrets

5.2.1.3 SEALED SECRETS

Sealed Secrets is an OSS project started by Bitnami used to encrypt Kubernetes secrets. KAOPS uses Sealed Secrets for Security giving users the option to encrypt things like cloud provider and private repository credentials in a way that can be shared with users outside an organization (e.g. managed service partner, public git repo, etc.).


K8SGPT

5.2.1.4 K8SGPT

K8sGPT is like having a Kubernetes expert in every cluster. When enabled (default), KAOPS will install K8sGPT in every cluster in a VAN. K8sGPT continually scans each cluster diagnosing and triaging Kubernetes issues in plain English. It anonymizes the data and runs the scans through OpenAI or Local AI (configurable).

When debugging issues K8sGPT makes it fast and simple to either pinpoint the problem to a Kubernetes level or rule it out as a root cause failure.


Argo

5.2.1.5 ARGO EVENTS AND WORKFLOWS

Argo Events and Workflows work together to provide a generic event-driven workflow automation framework for Kubernetes. A typical application for Events and Workflows is to automate CI/CD pipeline. For example, events can be defined (as a manifest) to monitor source code repositories and branches for changes and trigger workflows. Workflows (build, test, security scan, merge, etc.) can be conditionally pipelined in a way that results in a CI/CD framework.

5.2.1.6 OPTIMAL AND EXTENSIBLE

The golden path is not forced on KAOPS users. Installation of the projects is optional and can be deselected with building a VAN. Would you rather use Splunk instead of Grafana and Prometheus for observability? No problem.

> ***KAOPS is extensible by git.***

Using the Splunk example, a platform engineer using KAOPS can deselect Grafana when building the VAN, add the repo containing the Splunk manifests, and then create distribution rules to select target VAN clusters for Splunk deployment. Splunk will be deployed on all appropriate clusters in accordance with the distribution rules created by the user.

What about deploying tools not integrated into KAOPS?

No problem. Tools are deployed no differently, in that applications point KAOPS to the source repository containing the tool manifests, create the distribution rules, and the tool will be deployed.

I did this exact scenario recently for a customer demo. The customer wanted to see KubeVirt deploy in production. KubeVirt is a KAOPS roadmap item, but not yet integrated into the platform, so I deployed it using GitOps.

The following illustration sums up the KAOPS architecture.

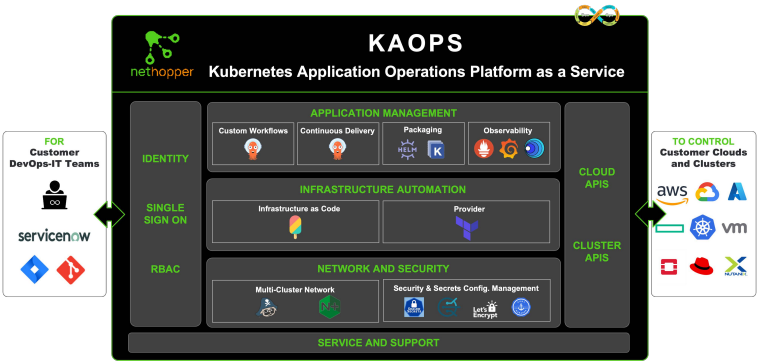


Figure 3 - KAOPS Architecture

5.3 EASE OF USE

The KAOPS UI Has Been Designed to Be Easy to Use

Getting Started with KAOPS

KAOPS can be operated from either the SaaS UI or its API. The KAOPS UI has been designed to be easy to use. The API is not covered in this document.

The following sequence diagram illustrates the typical steps for setting up and using KAOPS.

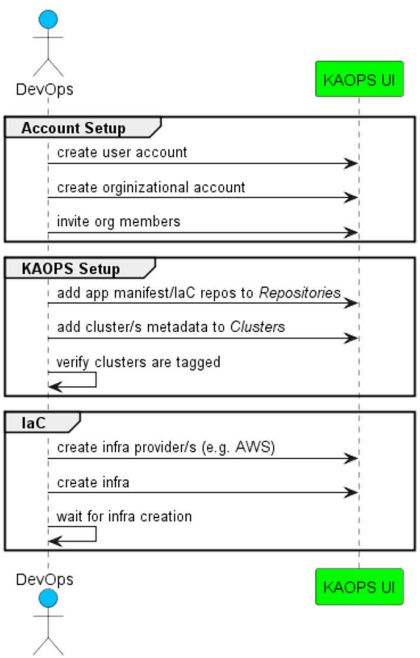


Figure 4 - KAOPS Setup

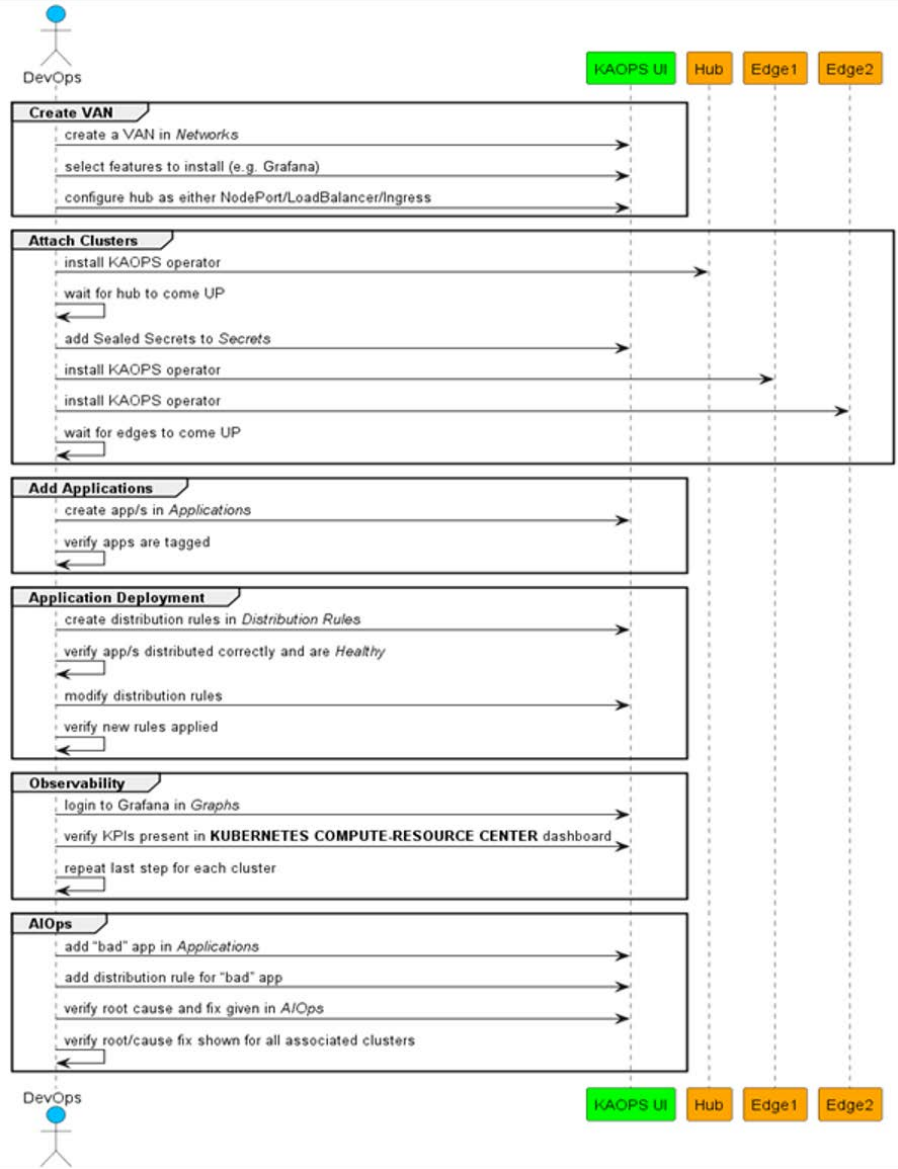


Figure 5 - Using KAOPS

6. CONCLUSION

Build vs. Buy: How to Decide

Key Factors to Consider

All organizations in the cloud native space will require a Platform Engineering initiative. The question for C-suite executives is (or will be), "Buy or build?". Building requires a team of experts in both the cloud native and PE spaces. Building also means time. Building a platform for cloud native environments takes quite a bit of time with no guarantee of success.

Buying a platform like KAOPS that provides an extensible framework for PE saves the time required to architect a solution and build it. KAOPS' framework gives organizations both an architectural and operational framework that comes with enterprise grade support for 8 (eight) OSS projects and counting (more coming!). No need for organizations to sign enterprise agreements with commercial companies built to support a single OSS project.

> ***Organizations can sign a single enterprise agreement with Nethopper and not 8 (eight) different companies.***

Learn More

Reach out to connect or learn more.

- dan@nethopper.io
- Book a 15-minute meeting with me
- LinkedIn

Nethopper's mission is to make it easy for enterprise platform/ DevOps teams to use a platform-engineering framework to build an IDP to operate Kubernetes applications across clouds and clusters. To learn more, visit www.nethopper.io. You can also follow us on LinkedIn, Twitter (X) and YouTube.

References

[1] Platform Engineering on Kubernetes by Mauricio Salatino; Manning Publications ISBN 9781617299322

[2] Common Pitfalls of App Modernization

[3] https://about.gitlab.com/topics/gitops

[4] https://argo-cd.readthedocs.io/en/stable/