Open in app ↗ Sign up Sign In

Search Medium

Published in Bluecore Engineering

Shray Kumar Follow
Oct 7, 2020 · 7 min read · Listen

Save

The Infrastructure Triumvirate: Continuous Service and Infrastructure Delivery with Argo, Kustomize, and Google Config Connector

ArgoCD, Config Connector and Kustomize! (image by Alexa Griffith)

Bluecore Engineering is growing at a frenetic pace — we are scaling our personalization platform to support the ingestion of hundreds of millions of events and emails. To meet this demand, our engineers are creating several containerized microservices on Google Kubernetes Engine (GKE), surgically removing them from our App Engine monolith. The infrastructure team was tasked with creating a delivery pipeline to quickly and reliably deploy new services to production on GKE to support the transition to a service orientated architecture, while ensuring that development teams could easily manage and scale their services on GKE as easily as they could on Google App Engine (GAE).

Background

The team began by identifying the foundational elements of continuous delivery. In Accelerate: The Science of DevOps, continuous delivery is described as "comprehensive configuration management" and states, "*It should be possible to provision our environments and build, test and deploy our software in a fully automated fashion purely from information stored in version control. Any change to environments or the software that runs on them should be applied using an automated process from version control.*" Under this mantra, we sought to implement tools that promote **first-class automation with best practices based on declarative configuration.**

Before our journey to continuous delivery began, Bluecore already made significant investments into GKE. Kubernetes is a powerful deployment platform but leaves many aspects of deployment and configuration up to the end-user. We had a few deployment solutions in place which followed our engineering principle, *as simple as possible, as powerful as necessary* which helped teams initially adopt Kubernetes. However, many questions remained outstanding:

- How can we deploy Kubernetes manifests to a cluster without manual interaction?
- Which templating engine do we select to reduce friction around deploying to multiple environments?
- How do we reduce the steps to create a new service?
- How can an engineer add a database, cache, or PubSub topic to their service without having to learn YAML and Terraform (or worse — clicking around in a console)?

It was time to revisit our initial solutions as we scaled in usage and complexity.

Prior Art and Exploring New Solutions

This investigation led us down a myriad of solutions in the Cloud Native ecosystem. Tools such as Helm and Spinnaker were familiar to our engineering team and showed promise, but did not pan out due to their need for additional automation, operational complexity, and steep learning curve. We also shied away from vendor-based solutions, looking to leverage open-source tools we could invest in.

Our goal was to provide a comprehensive solution that allows for an engineer to make modifications as they need, provide sane prescriptive defaults, reduce copypasta, enforce critical security, and configuration standards and provide '*batteries included'* functionality for those new to the ecosystem. With our foundational requirements in mind, we opted for what we dub to be the happy trifecta: **ArgoCD**, **Kustomize** and **Config Connector**. ArgoCD provides a straightforward GitOps-based approach to deployments; Kustomize provides a powerful way to layer and modify Kubernetes objects without templating, and Google's Config Connector provides a Kubernetes native way to manage Google Cloud resources like BigTable and Redis in the same manner as our application deployments.

Deployments With Argo

We began by provisioning ArgoCD and incorporating the App of Apps pattern. For those unfamiliar with ArgoCD, an Application (CRD) defines a path of manifests to deploy and a destination cluster. We have multiple apps and clusters, and an App of Apps allows you to deploy a specific set of applications to a specific cluster. We derived much of our inspiration from the neco-apps repository. ArgoCD has powerful features we've enabled like Automated Sync and Pruning to ensure your GitHub repository is the main source of truth, and that no engineer has gone in and made manual changes to your Kubernetes cluster. Additionally, some services leverage Argo Rollouts for stage-based deployments, promoting reliability by minimizing the blast radius of errors. Previously these deployments were tedious.

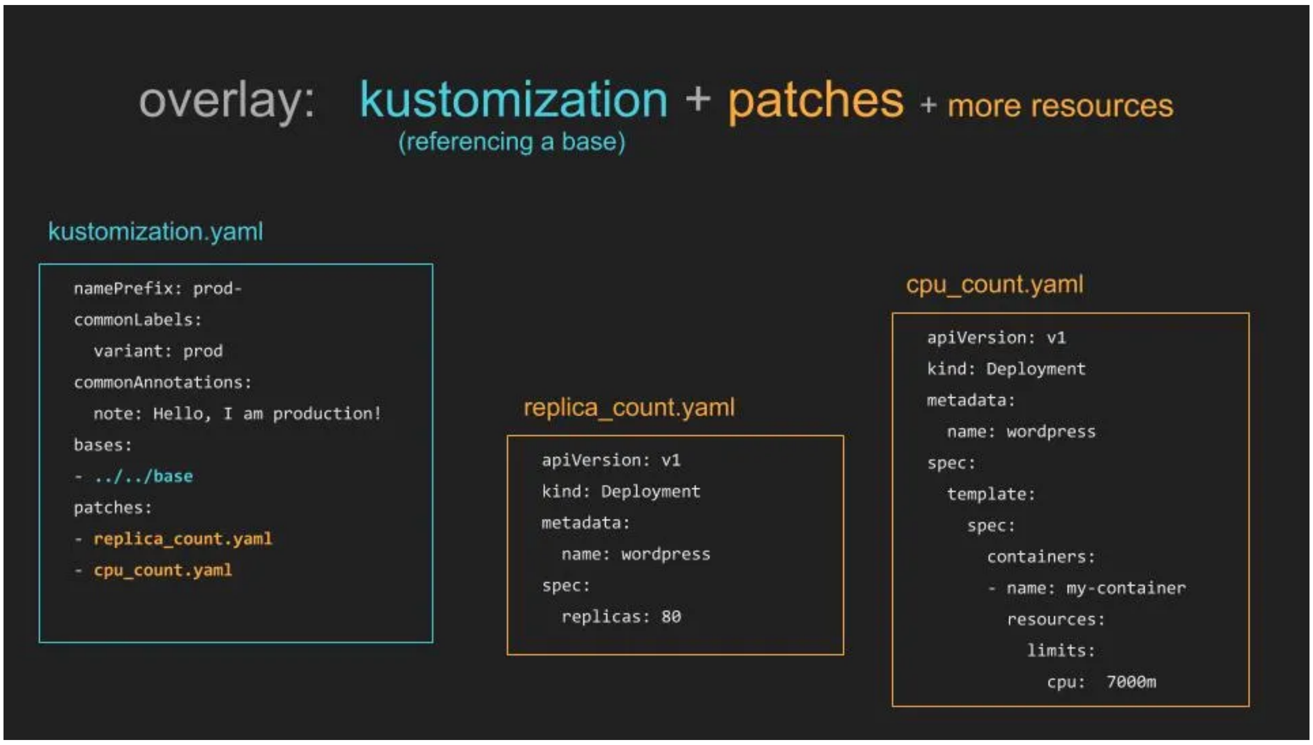
Deployment audits are trivial, as you can simply look back at your GitHub commit history to determine when a change was made and under what premise. This provides key visibility into who changed what and lets us tap into familiar Git based automation. Finally, ArgoCD provides a dashboard to allow insight into your cluster's current state to determine what may have diverged by manual intervention and whether a sync may be required. There are no longer phantom workloads running, making cluster migrations easier and increasing our security posture.

```
# App of Apps
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: istio-operator
  namespace: argocd
spec:
  project: infrastructure
  source:
    repoURL: git@github.com:bluecore/k8s-manifests.git
    targetRevision: master
```

```
      path: kustomize/istio-operator/base
      destination:
        server: 'https://kubernetes.default.svc'
        namespace: istio-operator
    syncPolicy:
      syncOptions:
      - CreateNamespace=true
    ignoreDifferences:
    - group: rbac.authorization.k8s.io
      kind: ClusterRole
      name: istio-operator
      jsonPointers:
      - /metadata/creationTimestamp
```

Un-Templating with Kustomize

Now that we had found our means for deploying Kubernetes manifests, it was time to choose a templating engine. Kustomize is a template-free customization tool for Kubernetes and was a suitable choice for many reasons. Adding another environment, or perhaps a secondary variant to an application, is as easy as creating another overlay. Kustomize will then handle prefixing components for you, which simplifies operations. Additionally, its base, stitch, and overlay approach means we can have sane defaults but provide extensions as needed. Previously, this would require bespoke helm charts or other complex templating support.



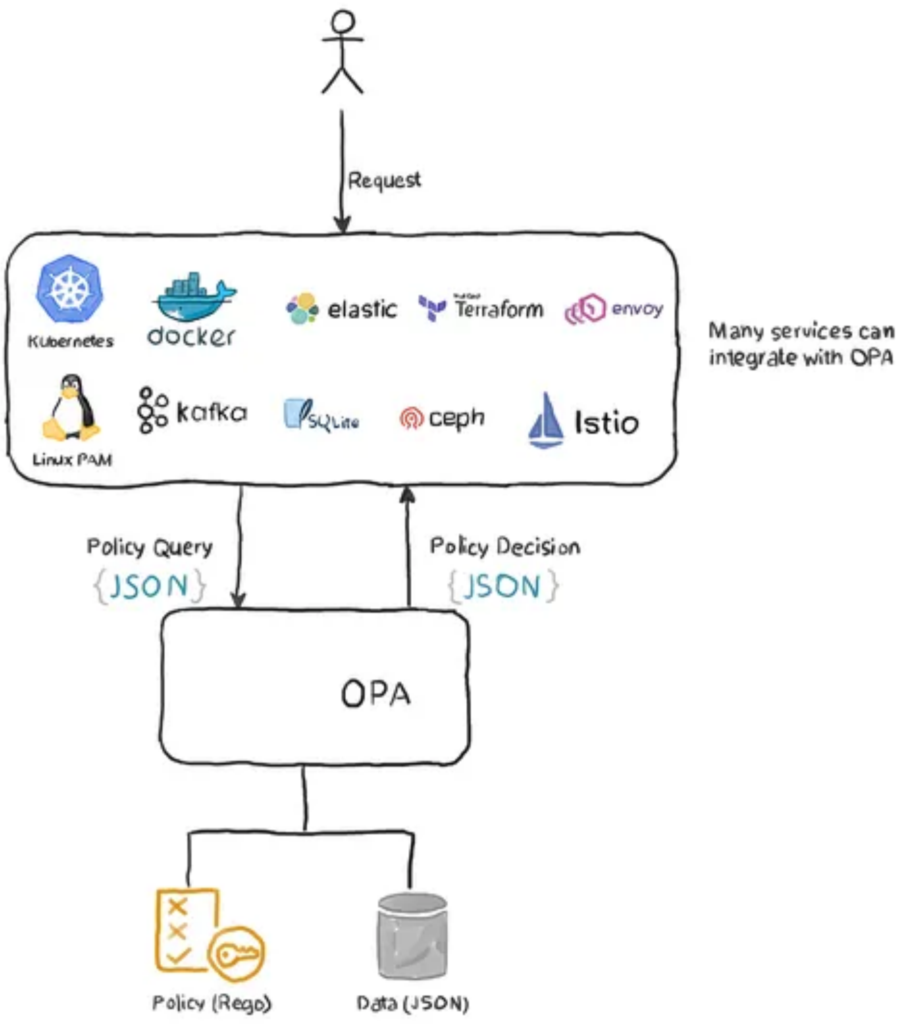
```
namePrefix: prod-
commonLabels:
  variant: prod
commonAnnotations:
  note: Hello, I am production!
bases:
- ../../base
patches:
- replica_count.yaml
- cpu_count.yaml
```

```
apiVersion: v1
kind: Deployment
metadata:
  name: wordpress
spec:
  replicas: 80
```

```
apiVersion: v1
kind: Deployment
metadata:
  name: wordpress
spec:
  template:
    spec:
      containers:
      - name: my-container
        resources:
          limits:
            cpu:  7000m
```

A Kustomize Overlay File, modifying replicas and cpu limits.

Kustomize also provides powerful abstractions to make labeling all Kubernetes objects in a resource straightforward. We use this to define a subset of Recommended Kubernetes Labels and an **k8s.bluecore.com/team** label on all resources. Now, determining what team owns a specific component is no longer a chore, as it's a single field in your Kustomize file that is auto-applied. Enforcing labels is critical to managing the large number of workloads on Kubernetes. For example, we use these labels in our Looker cost instrumentation dashboard. Our dashboard provides insight into how well instances are sized and how much services cost by rolling up label data with GKE Usage Metering in BigQuery. Additionally, these labels are enforced with a Rego Constraint using Open Policy Agent.



Policy as Code, Open Policy Agent — Thanks to Mohamed Ahmed

Kustomize's design makes it easy to distinguish what configurations differ across environments. The clear separation of patches in a Kustomize overlay file versus base manifests make it clear what's environment specific. This allows us to provide best practice defaults and control standard configurations across all our services. For new engineers, the learning curve of Kustomize over tools such as Ksonnet or Helm is much easier due to pattern matching and the familiar YAML based configuration.

Instead of large, messy yaml files and struggling with indentation, we can easily compose small, simple files together. Think of it as a JSON patch, with one file mapping to a field value. Kustomize rarely has the esoteric errors that crop up in other templating engines due to type mismatch, indentation, and runtime errors.

Infrastructure as Code with Config Connector

The final piece is Google's Config Connector. Config Connector is an extremely underrated component of the GCP ecosystem that's just beginning to pick up steam. Being able to deploy infrastructure through the same means as your Kubernetes application code is a powerful interface for developer-focused workflows. Engineers do not have to learn multiple infrastructure as code languages (HCL, YAML, and Go-Template) to build repeatable infrastructure. With Config Connector, it's easy to provision a PubSub topic and subscription for your service, create an IAM user, and bind your service user to your Kubernetes workload with Workload Identity. There is no need for GOOGLE_CREDENTIALS environment variables or key exports! And all without ever opening your GCP Console. Combined with Kustomize templating, we can also manage repeatable infrastructure across environments. Config Connector's true power lies in its ability to assert its configuration as your infrastructure's desired state. One downside with Config Connector is that it does not mutate resources well, so changes require some orchestration.

```
apiVersion: iam.cnrm.cloud.google.com/v1beta1
kind: IAMPolicy
metadata:
  name: cortex-workload-identity
  namespace: ctx
spec:
  bindings:
  - members:
    - serviceAccount:bluecore.svc.id.goog[ctx/ctx]
    role: roles/iam.workloadIdentityUser
  resourceRef:
    apiVersion: iam.cnrm.cloud.google.com/v1beta1
    kind: IAMServiceAccount
    name: ctx
---
apiVersion: v1
kind: ServiceAccount
metadata:
  annotations:
    [illegible]
  [illegible]
  namespace: ctx
```

Kubernetes   Google Cloud Platform   Dev Ops   Software Development   Bluecore

Shipping New Services To Production

What does this all look like end to

ngineer starts a new service at Bluecor a template project is used to generate starter project code and Kubernetes manifests. The manifests are then checked into a mono-repository for Kubernetes where pre-commit hooks are run, manifest linters enforce sane defaults, and OPA constraints must be met. This provides early feedback on potential issues.

After the initial commit, subsequent commits to the application code repository trigger t CI cycle and upload a Docker image to a private Google Container Registry. When the

ad r invokes an in-house tool, named *Dinghy,* to update t Kustomize file's image tags in our mono repository with the new hash of the updated Docker Image. Finally, a change to the Kustomize file triggers a Github Webhook to Argo which invokes its sync policy for your application, deploying the application.

We had a blast building this infrastructure — if you're interested in a smooth deployment process, or in general tackling some of our technical challenges around our email platform, data science or infrastructure, check out our careers page!