Cloud Operations, FS2022

Luzia Kündig

July 14, 2022

1 Introduction

CI/CD	Deployment environment, provides name and url for monitoring
Ansible	Installation and configuration of applications
Terraform	Provisioning of infrastructure
Kubernetes	Configuration + application: Deployment, scaling, managing workloads
Helm	Package- and Lifecycle Manager for K8s
Kustomize	Overlaying declarative specifications on top of existing K8s Manifests
Prometheus	Monitoring K8s Infrastructure and applications for reliability
Service Mesh	Traffic Management, Security, Observability and Service Discovery
GitOps	Everything as code, declarative system operation definition, control loop

1.1 DevOps

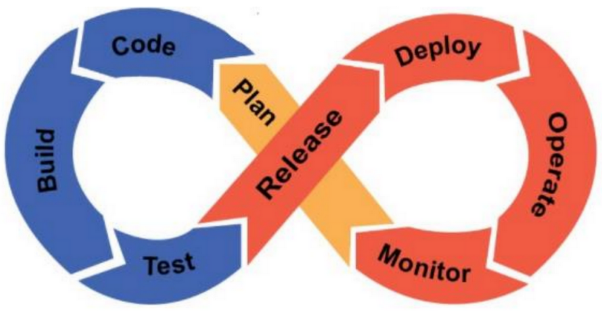


Figure 1: DevOps Cycle

The DevOps pipeline focuses on Continuous Integration / Continuous Deployment. It applies a systems thinking and avoids too much focus on only one piece of the workflow.

It values feedback, automated and personal to keep production healthy. Monitoring is an important type of automated feedback.

Learning and continued experimentation lead to constantly improved systems and workflows.

1.1.1 Plan

Add objectives and requirements to the **backlog**, feedback from end users and operations team. Add backlog **tasks to sprints**. Track and plan activities using **board and project management tools**.

1.1.2 Code

Code from all developers gets integrated into a **central source code repository**.

1.1.3 Build

Continuous Integration pipeline is invoked every time code is pushed into the central repository. Includes **automated unit and integration tests**. Only after successful build and test, code can be reviewed and merged.

1.1.4 Test

Automated deployment of code into a testing environment. Tests executed include **load, accessibility, performace and end-to-end testing**. Manual work like **user acceptance testing** also usually happens at this stage.

1.1.5 Release

Tag a snapshot of the code with a **semantic versioning number**. Changes, features, breaking changes and deprecated features are documented. Release can include artifacts such as **binaries** and **packages**.

1.1.6 Deploy

Installs release into **production environment**. Can be automated or manual.

1.1.7 Operate

Infrastructure and Operations team ensure **smooth operation of the product**. **Scaling infrastructure** to meet demands.

Issues in infrastructure can be troubleshooted and resolved. **Document issues** for next planning stages.

1.1.8 Monitor

Collect data on usage, performance, errors and more. Data collected is used for next iteration of DevOps cycle.

1.2 Automated DevOps with GitLab CI

Using GitLab CI pipeline, code can be built, tested and deployed automatically on every change. Tasks are defined in .gitlab-ci.yml file kept together with the code repository.

1.2.1 Integrating Kubernetes

1.2.2 Auto-DevOps: Setup with zero configuration

1.2.3 Creating gitlab CI Configuration

1.2.4 Testing with Docker

2 CI/CD

2.1 Automated application deployment

2.1.1 Declaring deployment enviroments

Environments in CI definitions describe where code gets deployed. It can be linked to a kubernetes cluster and usually defines a name and url for monitoring the overall application state.

Monitoring a deployment in GitLab requires installation and configuration of Prometheus.

2.1.2 Kubernetes application ressources

Docker Image Tag can be built by combining GitLab variables:

`$CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA`

To apply a kubernetes yaml manifest during CI/CD Pipeline, use the following script line:

`cat k8s.yaml | envsubst | kubectl apply -f -`

Inside the kubernetes yaml manifest, the image can be specified using a variable such as "${DOCKER_IMAGE_TAG}"

2.1.3 Deploying an application to Kubernetes

Kubernetes Cluster needs to be configured as deployment destination in GitLab repository. For this, an agent must be installed inside the cluster. This agent can then be used to communicate through a NAT, access cluster API endpoints in real time, push information about events as well as enable a cache of kubernetes objects.

With a GitOps workflow, kubernetes manifests are kept inside GitLab, and on every change to the repository manifests, the agent inside the cluster automatically updates resources accordingly. This is considered pull-based, because the cluster agent actively pulls from the repository.

The classical CI/CD workflow pushes new configuration from the GitLab repository to the claster using GitLab CI script commands on the kubernetes API.

Environment scope defines which environments are automatically assigned to this cluster when created.

2.1.4 Deploying tokens and pulling secrets

How does the kubernetes cluster access information inside our gitlab repository? Creating a deploy token we can provide the kubernetes cluster with authentication credentials to pull images from the GitLab container registry. read_registry access should be enough.

These credentials must be saved as environment variables inside the GitLab repository and can then be used to create a kubernetes secret inside the CI script. This kubernetes secret is used as "ImagePullSecret" inside the manifest to create resources.

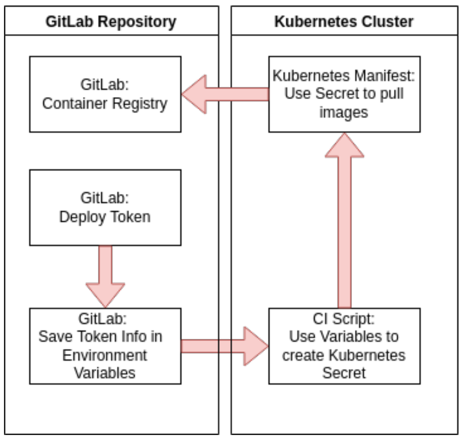
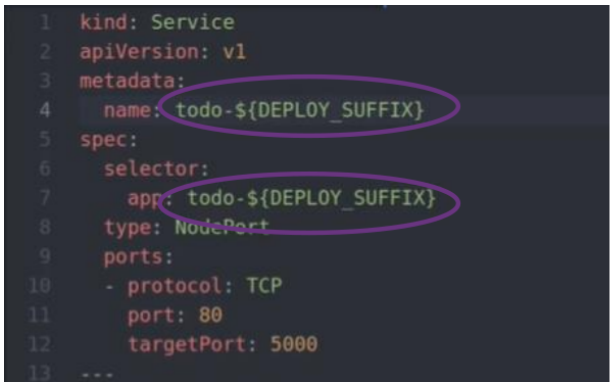

GitLab Repository
Kubernetes Cluster
GitLab: Container Registry
Kubernetes Manifest: Use Secret to pull images
GitLab: Deploy Token
GitLab: Save Token Info in Environment Variables
CI Script: Use Variables to create Kubernetes Secret

Figure 2: Cluster Authentication

2.1.5 Dynamic environments and review apps

Defining job environment variables inside the CI Script allows us to use the same k8s file for different branches, referencing different applications

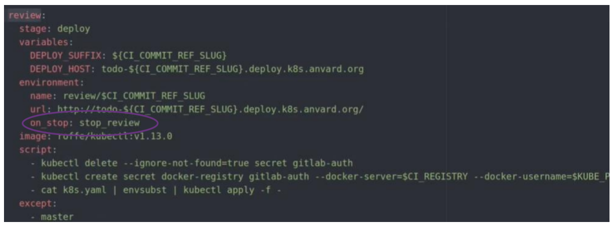
- DEPLOY_SUFFIX
- DEPLOY_HOST



```
kind: Service
apiVersion: v1
metadata:
  name: todo-${DEPLOY_SUFFIX}
spec:
  selector:
    app: todo-${DEPLOY_SUFFIX}
  type: NodePort
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
---
```

Figure 3: Using environment variables defined in CI-Job

Review Apps can be created automatically when pushing to any non-production branch. CI_COMMIT_REF_SLUG creates a valid k8s name from the branch name. A new environment in GitLab is created automatically.



```yaml
review:
  stage: deploy
  variables:
    DEPLOY_SUFFIX: ${CI_COMMIT_REF_SLUG}
    DEPLOY_HOST: todo-${CI_COMMIT_REF_SLUG}.deploy.k8s.anvard.org
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    url: http://todo-${CI_COMMIT_REF_SLUG}.deploy.k8s.anvard.org/
    on_stop: stop_review
  image: roffe/kubectl:v1.13.0
  script:
    - kubectl delete --ignore-not-found=true secret gitlab-auth
    - kubectl create secret docker-registry gitlab-auth --docker-server=$CI_REGISTRY --docker-username=$KUBE_P
    - cat k8s.yaml | envsubst | kubectl apply -f -
  except:
    - master
```

Figure 4: Automatically creating review apps for branches

2.2 Application Quality and Monitoring

Integration and functional testing can be supported either using review apps or defining instances of containers as services. This makes the service available to the main container built inside the CI job.

2.2.1 Analyzing code quality

Separate jobs that check code quality inside the build pipeline provide results either as reports in merge requests or as separate job artifacts. Code Quality jobs usually run parallel with the other test jobs to reduce overall pipeline time.

Gitlab provides an image specifically for code quality which creates JSON reports – comparing results to previous ones automatically is not a free feature in GitLab.

2.2.2 Dynamic application security testing – DAST

Usually occurs after the deploy stage. Creates a JSON report and compares it with the last one to check for any differences to the reports from previous merges. Not a free feature.

2.2.3 Application monitoring with Prometheus

2.3 Custom CI Infrastructure

2.3.1 Runners

3 Terraform

3.1 Basics

Infrastructure reqirements for deploying applications include VMs, security, network access, firewall rules, availability and maintainability.

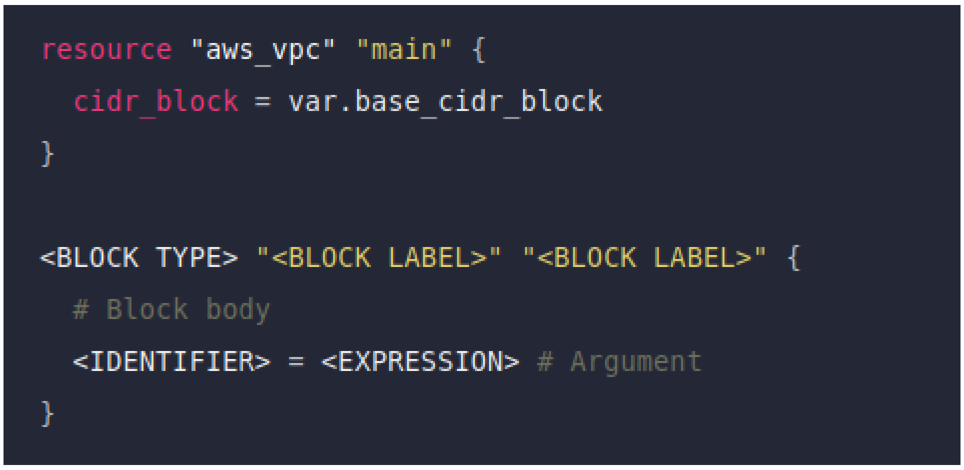
Terraform is responsible for deploying and maintaining infrastructure. It supports infrastructure providers such as AWS, Microsoft Azure, Google Cloud and many more via provider plugins.

These plugins convert the terraform calls into something that can communicate with the client SDK of the cloud provider.

Terraform commands are written in Hashicorp Configuration Language HCL which

- is simple, easy to learn
- has in-built type system
- supports for-loops, dynamic blocks, contitionals and string interpolation

The terraform language consists of blocks, arguments (assigning values to a name) and expressions.



```
resource "aws_vpc" "main" {
  cidr_block = var.base_cidr_block
}

<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

A Terraform configuration is a complete document in the Terraform language that tells Terraform how to manage a given collection of infrastructure. A configuration can consist of multiple files and directories.

3.2 Components

version.tf – describes which terraform version and which providers are required
provider.tf – contains access keys, etc.

main.tf – contains actual infrastructure descriptions

terraform init – initializes working directory with specified provider selections, etc.
terraform plan – calculate the changes based on declarations and state, display what would be executed
terraform apply – run the deployment, create, update or delete resources as needed
terraform destroy – remove all resources defined in terraform configuration files

.tfstate – current state information. is used for terraform plan command.

3.3 Resource referencing

<ResourceType>.<Name> represents a managed resource of the given type and name.

VAR.<Name> is the value of an input variable oof the given name.

3.4 Outputs

3.5 Data

3.6 State file sharing

The state file *terraform.tfstate* kept for every environment is best stored in a central location when working in a team. Remote storage providers enable locking of the state file for every operation, in case several users apply modifications at the same time.

The refresh operation synchronizes the state file with the actual status of the managed infrastructure and maps object IDs to the resource instances defined inside terraform configuration.

3.7 Modules for Code Reuse

Modules are containers for multiple resources that are used together. A module consists of a collection of *.tf* and / or *.tf.json* files kept together in a directory. The root module is usually the main folder of the terraform resources.

4 Ansible

Focus in ansible is writing configuration as code that can easily be pushed out to managed devices, reducing or eliminating the need to manually configure single devices. There is no client installation needed on the endpoints.

Alternatives to ansible include Puppet, Chef and SaltStack. Based on python and extensible via modules, ansible has about 65% market share.

4.1 Basics and Setup

Ansible is usually installed on a control node, e.g. an operators workstation. Inventory, code and playbooks can then be managed and shared through version control systems. Connections to assets that should be configured are initiated for every task that is run, using the ssh protocol for Linux and winrm for Windows devices..

To run tasks, the following is required on an operators machine:

- Python installation
- Ansible installation
- `/etc/hosts` for dns resolution
- ansible.cfg for configuration (default in `/etc/ansible/ansible.cfg`) used for default settings including privilege and remote user, "fallback" for settings that aren't provided at a more specific level
- inventory file (default in `/etc/ansible/hosts`)
  to identify all managed hosts, can be dynamic using central inventory software or static file with host lists

On the managed devices:

- Python installation
- Dedicated ansible user account with ssh / sudo access, typically key-based

4.2 Ad-Hoc Commands and Modules

Executing a single module on some hosts in the inventory can be done by command line:

```
ansible
   -i <inventory file> all
   -m <module name>
   -a <module argument>
   -u <username>
```

`-k` *(ask for password)* Running the same task a second time, ansible recognizes if changes had to be done or the command did not have any effect.

Modules to avoid: *command, shell, raw* because there is no idempotency. Ansible cannot determine, if the configuration is already there or has to be applied. If a simple command is run (like useradd), instead of recognizing the "ok state" (user already exists, do nothing) ansible returns the actual error the command outputs. They should only be used as a last resort, if there is no appropriate ansible module available.

4.3 Playbooks

A playbook contains multiple tasks that are run in order. Errors in one task will cause the whole play to fail.

Each playbook is written in yaml. It consists of several p

4.3.1 Facts and Variables

4.3.2 Working with Conditionals

5 Service Mesh

5.1 Microservice Principles

The modern architecture of microservices focuses on the following aspects of operation.

1. Deployment Independence
2. Oranized by business capability
3. Products not Projects
4. API Focused
5. Smart endpoints and dumb pipes
6. Decentralized governance
7. Decentralized data management
8. Infrastructure Automation (IaC)
9. Design for failure
10. Evolutionary design

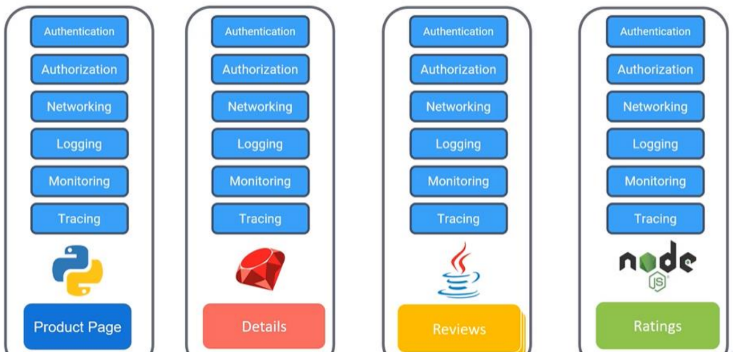


Figure 5: Traditional Service Architecture

5.2 Service Mesh

A service mesh is a dedicated and configurable infrastructure layer that handles the communication between services without having to change the code in a microservice architecture.

Instead of every service implementing important functionality themselves, a proxy is deployed to intercept network traffic to each container inside a pod.

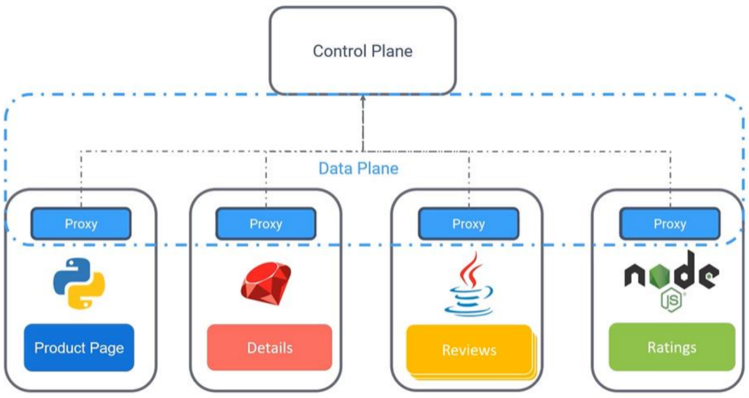


Figure 6: Service Mesh Architecture

Service Mesh is responsible for Traffic Management, Security, Observability and Service Discovery inside each microservice. This is done by embedding capabilities via sidecar containers directly into kubernetes pods.

5.3 The Istio Approach

Grafana	Visualization of metrics collected by Prometheus
Istio Ingress Gateway	Envoy Proxy which serves as an entry point for the service mesh
Istiod	Backbone of the Istio control plane, configures sidecar proxies
Jaeger	Provides tracing functionality for traffic inside the service mesh
Kiali	Istio dashboard
Prometheus	Collects monitoring information from sidecar proxies

5.4 Canary Deployment

Canary deployments are used to test a new version of a microservice in production. To do so, the new version gets deployed, but only a small percentage of traffic gets sent to it. If no problems occur and no customers complain, the traffic rate to the new version gets increased.

5.5 Virtual Services

Inside a VirtualService configuration file, a set of traffic routing rules can be defined that should be applied when a host is addressed. Each routing rule defines matching criteria for traffic of a specific protocol.

If traffic is matched, it is sent to a named destination service or a specific subset / version of it. This can be used for load balancing and different deployment strategies.

5.6 Jaeger Spans and Traces

A trace represents a single request through the service mesh which gets handled by the services. Each unit of work inside a trace is called a span. Spans can be requests to other services, for example.

6 GitOps

The concept of GitOps enforces using Git as *single source of truth defining the application state.*

Instead of different people using *kubectl create / apply* or *helm install / upgrade* commands from their own laptops, the whole configuration of a kubernetes cluster is kept inside a (separate!) Git repository.

This brings the known advantages of a version control system to Continuous Delivery for Kubernetes Clusters:

- **Versioned and immutable**
- **Declarative** definitions of apps, environments and configurations
- **Automated and repeatable**, less opportunity for errors
- Code review for changes
- Tracking of who did what
- Rollback via Git
- Whole infrastructure can be recreated from source control

Separating the git repository holding kubernetes manifests and the actual application code provides more simplicity when updating some deployment information.

No build and test pipeline will be triggered without any change to the actual code. Git history will be cleaner. Application may be distributed over several git repositories. Separation of access is possible.

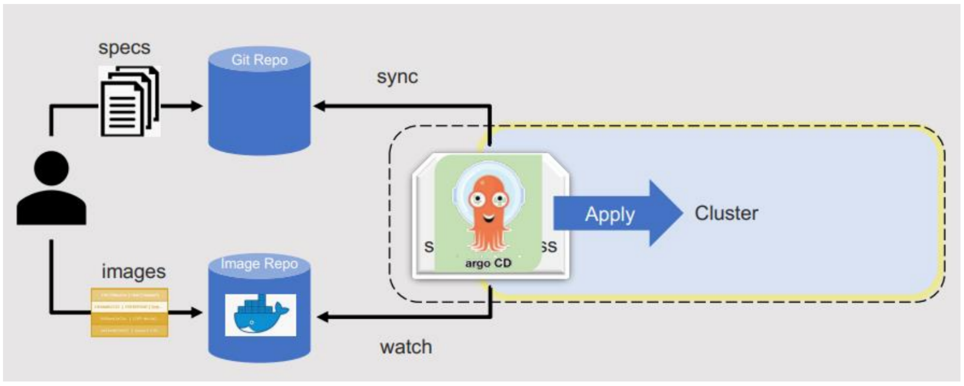
6.1 ArgoCD

Declarative GitOps tool used to deploy applications on Kubernetes.

It is lightweight, easy to configure, built with GitOps Workflow in mind and intended only for Kubernetes.
How does it work?

- Spins up its own controller inside the cluster
- watches for changes in a repository
- compares against resources deployed in the cluster
- synchronizes both states (desired state wins)

Its key concepts are the following:



App/Application	Group of K8s resources defined by a manifest (CRD), used to monitor repository and update cluster
Application source type	Which build tool is used to build the app
Live State	Current state of the application cluster
Target State	Desired state as stored in Git
Sync Status	Is live the same as target?
Sync	Updating app to the target state
Sync operation status	Success/failure state of the sync operation
Refresh	Comparison of Code in Git with live state (.. of Sync Status)
Health Status	Is the app running correctly? Serving Requests?
Configuration Management Tool	Tool to create manifests from files in direcory (Kustomize, Ksonnet)
Configuration Management Plugin	Custom tools ...

6.2 ArgoCD Application

Specifies information such as ArgoCD project, source repo, revision, path, cluster, namespace.

Can be created via command line, Web Interface, Yaml in web, K8s Manifest (CRD).

Can be specified using kustomize, helm, ksonnet, jsonnet, plain yaml, custom configuration management tool set up as plugin to ArgoCD.

```
argocd app create, argocd list
```

Different app health statuses include

- Progressing
- healthy
- Degraded
- Suspended
- Missing

6.2.1 History and Rollback

ArgoCD keeps track of the various versions deployed and allows you to go back to a previous state. Can be accessed from the graphical user interface.

6.2.2 Manual Sync

- **Prune**: Allow deleting resources that are unexpected, meaning they no longer exist in git.
- **Dry Run**: Preview what an apply operation would do without affecting the cluster
- **Apply only**: Skip pre/post sync hooks
- **Force**: Deletes and re-creates resource(s) when patch encounters conflict after 5 retries

6.2.3 Automated Sync

Only occurs if App Sync Status is *OutOfSync*. Attempts one sync per unique combination of commit hash and parameters of the app – unless *selfHeal* flag is true.

sefHeal attempts to sync after a default timeout of 5 seconds.

6.2.4 ArgoCD Projects

Projects can be a logical group of applications, which supports organization by teams.

Features include restrictions and roles to isolate different teams and the resources available inside a project.

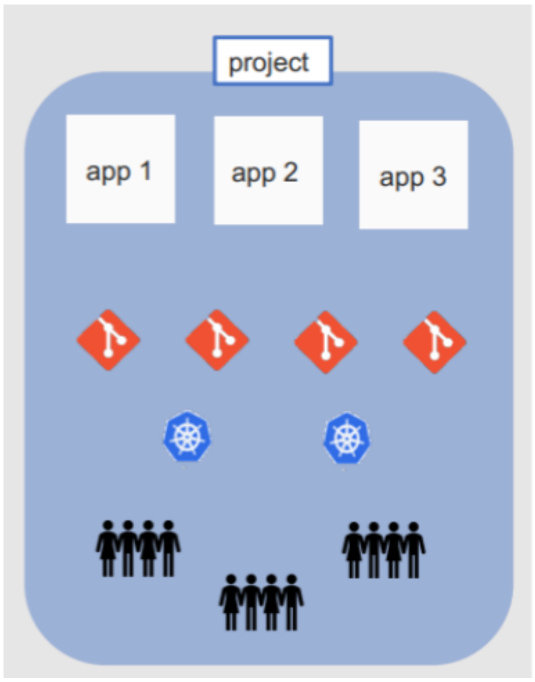


Figure 7: ArgoCD Projects