CldOps Cheatsheet

Fabio Zahner

1 General Definitions

Definitions:

- **Cloud Operations** is the practice of managing and optimizing cloud-based services and infrastructure.
- **GitOps**: Git-based infrastructure and application deployment; uses Git as single source of truth; enables CI/CD, automation, version control, and declarative configuration.
- **DevOps** combines development and operations; focuses on automation, collaboration, CI/CD, monitoring, and agile delivery.

1.1 DevOps Cycle

Plan (add Objectives and Requirements to Backlog), **Code** (add Code to Repo), **Build** (Pipelines runs on push, builds and unit tests software), **Test** (Build is deployed to staging environment, tested using E2E, load, accessibility tests), **Release** (snapshot of code is versioned, changes are documented), **Deploy** (release is installed onto production environment), **Operate** (application should run smoothly, issues are troubleshooted and documented, infrastructure is scaled), **Monitor** (Application Data is gathered and used for planning)

Difference Between Continuous Delivery & Continuous Deployment: Deployment automatically pushes from staging to production, in Delivery this is manual.

CD&D Deployment Strategies:

Rolling Deployment: Update infrastructure gradually, minimal downtime

Blue-Green: Two environments: Old and new versions respectively

Canary: Small user group tests first

Feature Flag: Deploy but activate later, can be toggled

Dark Launching: Rolling out a feature invisible for users, test its performance in the background

2 GitLab Pipelines

```yaml
stages:
  - build
  - test
  - deploy
cache:
  paths:
    - .cache/
build:
  stage: build
  script:
    - echo "Building..."
    - mkdir -p artifacts && echo "artifact
      " > artifacts/output.txt
  artifacts:
    paths:
      - artifacts/
    expire_in: 1 hour
test:
  stage: test
  dependencies:
    - build
  script:
    - cat artifacts/output.txt
deploy_staging:
  stage: deploy
  environment:
    name: staging
    url: https://staging.example.com
    on_stop: stop_staging # Unstages env
  script:
    - cat k8.yaml | envsubst | kubectl
        apply -f -
  artifacts:
    expire_in: 1 hour
stop_staging:
  stage: deploy
  environment:
    name: staging
    action: stop
  script:
    - echo "Stopping staging"
```

2.1 Environments

Describe where the code gets deployed (e.g. Local, Integration, Testing, Staging, Production). Can be linked to a K8 cluster (needs to be set up via GitLab UI):

2.2 Push- vs. Pull-Based Deployments

Push-Based: + Easy to use, + flexible deployment targets, - firewall needs to be opened, - pipeline needs to be adjusted for new environments **Pull-Based:** + no need for open firewall, + better scaling, - agent needs to be installed in every cluster

3 Terraform

TF doesn't speak directly with an SDK, but rather Terraform -> Provider -> Client SDK. Different providers enable diffrent platforms (AWS, Azure, Kubernetes, ...). A sample in HCL (Hashicorp Configuration Language):

```hcl
variable "instance_type" {
  default = "t2.micro"
}
provider "aws" {
  region = "us-east-1"
}
resource "aws_instance" "web" {
  ami = "ami-0c55b159cbfafe1f0"
  instance_type = var.instance_type
}
output "public_ip" {
  value = aws_instance.web.public_ip
}
```

To deploy infrastructure, write HCL in files like `main.tf`, then run `terraform init`, `terraform plan` to show changes that would be made, `terraform apply` to actually apply the changes. Use `terraform destroy` to delete all made changes.

3.1 State

Terraform stores state in the `terraform.tfstate` file. When working in teams, this state file also has to be shared as the terraform command relies on is validity. This could for example be done via an S3 Bucket.

4 Ansible

Ansible can be used to provision servers. It does not have statefiles and is idempotent, meaning it won't make changes unless it has to.

4.1 Infrastructure

In a network of servers, one server is the **host**. The host can connect to other machines using SSH. On the host, playbooks can be written in `yaml` files. Run a playbook by using `ansible-playbook playbook.yaml`

```yaml
- name: Example Playbook
  hosts: web
  become: true
  vars:
    packages:
      - nginx
      - curl
    enable_service: true
    secret_password: "{{ vault_password }}
      "
  roles:
    - myrole
  tasks:
    - name: Install packages
      apt:
        name: "{{ item }}"
        state: present
      loop: "{{ packages }}"
      notify: restart nginx
    - name: Configure app if enabled
      template:
        src: app.conf.j2
        dest: /etc/app.conf
      when: enable_service
      tags: config
  handlers:
    - name: restart nginx
      service:
        name: nginx
        state: restarted
```

4.2 Vaults

Vaults can be used to encrypt data: The file `vault.yaml` with the contents `vault_password: "mysecret"` can be encrypted using `ansible-vault encrypt vault.yml` and then included in a play: `ansible-playbook playbook.yml -ask-vault-pass` To create a file, use `ansible-vault create foo.yaml`

4.3 Collections, Roles & Tags

Collections are bundles of plugins, roles and modules. Install them using `ansible-galaxy collection install <name>`, or define a requirements.yaml to install multiple collections at once. **Roles** are an abstraction above playbooks, allowing to reuse configuration steps: create a role using `ansible-galaxy init <name>`, then use a role like in the example above. **Tags** can be used to execute a subset of tasks instead of the whole playbook. Run only specific tags by appending `-tags <name>` at the end of the ansible-playbook command. There are also two special commands: Tag *always* runs every time, except when explicitly skipped: `-skip-tags=always`. Tag *never* does not run unless specified with `-tags=never`

4.4 Jinja2

Jinja2 is the templating engine which is used by Ansible. It is used to generate configuration files.

5 Kubernetes (K8)

> **K8 Objects:** Persistent entities which signal an intent (e.g. for something to be created on the cluster)
>
> **K8 Controller:** Tracks an Object and is responsible for bringing the current State closer to the desired State.

Pod: Represents a process running on your cluster. Should contain one container, multiple are possible.

Sidecars: Sidecars are containers that run along the primary container in the same pod. Example use cases might be logging, security, data synchronization.

Init Containers: Similar to sidecars, but run and finish before app containers.

Volume: Assigns physical Storage to a Pod

ReplicaSet: Makes sure that a specified number of replica pods are running. In practice, deployments are used.

Deployment: Allows to manage one or multiple Pods.

Service: API Resource to expose logical set of Pods in the namespace. Acts as a load balancer (round-robin).

Ingress: Provides external Access to a Service.

DaemonSet: Ensures that all (or some) Nodes run a copy of a pod. This can be used for running supporting applications (logs, storage)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app
spec:
replicas: 3 # automatically deploys
    replicaset
selector:
  matchLabels:
    app: web
template:
  metadata: # pod
    labels:
      app: web
  spec:
    strategy:
      type: RollingUpdate # rolling update
      rollingUpdate:
        maxUnavailable: 1
        maxSurge: 1
    initContainers:
    - name: init
      image: busybox
      command: ['sh', '-c', 'sleep 10']
    containers: # container
    - name: web
      image: nginx
      ports:
      - containerPort: 80
    - name: sidecar
      image: busybox
      command: ['sh', '-c', 'while true;
         do sleep 30; done']
```

5.1 Namespaces

Used to separate resources. Only resources in same namespace can communicate directly.

5.2 Rolling Updates

Rolling updates can be used in order to ensure that enough pods are always running. Rolling updates can be using maxUnavailable (maximum No. of Pods upgrading at the same time) and maxSurge (max No. of Pods allowed to run beyond specified No. of replica)

5.3 Scheduling

The `kube-scheduler` determines which nodes run which Pods. We can influence this decision process:

```yaml
kind: Pod
spec:
  nodeSelector:
    disktype: ssd # this label needs to be
        in pod.spec
```

To evaluate if a Node is eligible to run a Pod, the following things are considered: Port availability, CPU & Memory resources, available volumes, specified labels. Additionally, scoring is used to evaluate remaining nodes with criteria: pods of same service should be on different nodes, nodes with few used resources are prioritized, node affinity. *Taints* can also be applied on nodes and pods, pods won't be deployed on nodes with matching taints. *Tolerations* can be used to make exceptions to taints. Types of taints: *NoSchedule, PreferNoSchedule, NoExecute*

5.4 Commands

Apply a manifest.yaml: `kubectl create|apply|replace -f manifest.yaml`
Connecting to a Pod: `kubectl exec -it nginx-xxx - sh` Undo rollout: `kubectl rollout undo`

6 Helm

A package manager for K8, enabling to reuse configurations for common use cases (DB, monitoring). Helm provides *Charts*, which are a collection of yaml files describing different K8 Objects. When deploying a chart on your cluster, it is called a *Release*. Charts are available through different *Repositories*. Helm charts use templating (like `{{Release.Name}}`, for information about package, or `{{.Values.xyz.abc | default ëxample"}}` for information passed by values.yaml or via commandline `-set`)

6.1 Commands

```bash
# Repo commands
helm repo list
helm repo add <repo> <url>
helm repo rm <repo>

helm list # list installed releases
```

```
helm install -f values.yaml <release> <
      chart> # install with custom values
helm show <chart | readme | values | all>
      <chartname | repo>

helm upgrade <release> <chart>
helm rollback <release> <revision>
```

7 Kustomize

The kustomize.yaml defines the resources and transformations. *Transformers* transform your manifests by allowing common fields in the resources to be specified in one place.
In case you want multiple variations on a common resource, the files in `base/` can declare common elements and the files in `overlays/` will declare differences.
Patches allow you to change (patch) already set values.
Traditionally, when editing ConfigMaps or Secrets, the deployment did not change, therefore no pods were restarted. **Generators** allow to define values in the `kustomize.yaml` file, where changes get detected and pods automatically restarted.

```
# A sample kustomize.yaml file
apiVersion: kustomize.config.k8s.io/v1
kind: Kustomization
resources:
  - deployment.yaml
  - service.yaml
# Transformers:
namePrefix: dev-
namespace: my-namespace
commonLabels:
  app: my-app
# Patch:
patches:
- patch: |-
    - op: replace
      path: /metadata/name
      value: nginx-server
  target:
    kind: Service
    name: nginx-app
# Generator:
configMapGenerator:
  - name: app-config
    literals:
      - LOG_LEVEL=debug
# Adding component from component example
      below
components:
- ../../components/mysql
```

7.1 Components

Components encapsulate a set of modifications. Below is an example component which adds a mysql database. This component can then be included in overlays, base directories (see above) or other components.

```
# Sample component file. Path: components/
      mysql/kustomization.yaml
apiVersion: kustomize.config.k8s.io/
      v1alpha1
kind: Component # not Kustomization
resources: # resource files are also under
      /mysql/
- deployment.yaml
- service.yaml
```

7.2 Commands

```
kustomize build <dir> # Renders manifests.
kubectl apply -k <dir> # Applies
      kustomization via kubectl.
```

8 K8s Monitoring & Logging

Monitoring: metrics, alerts, trends – quantitative (resource usage, request counts, error rates)
Logging: detailed event records – qualitative (stack traces, system messages, business events)
Kubernetes does not persist metrics out of the box. **kubectl logs** only persists logs for running containers, logs lost when container dies. **Agent-Based Monitoring/Logging:** A push-based system where software installed on server collects data and sends it to a central monitoring server. **Agentless Monitoring/Logging:** A central monitoring server requests data from the devices being monitored.

Example Stack: *Monitoring:* Exporters (custom for each app, exposes metrics), Prometheus (metrics collection), Grafana (dashboard), Alertmanager (rule-based alerting) *Logging:* Fluent bit/fluentd (log collection), Loki (aggregation), Grafana (visualization) *Alternative: EFK (see subsection EFK)*

8.1 Monitoring

Prometheus: Written in Go, stores data in a time-series DB. Scrapes metric endpoints over HTTP. Has client libraries to instrument custom apps written in Go, Java, Python, Node.js, .NET, ... Uses PromQL language to aggragate data across labels. Some PromQL examples:

```
sum by (namespace) (kube_pod_info) --
      count of pods per cluster and
      namespace
sum by (namespace) (kube_pod_status_ready{
      condition="false"}) -- count of pods
      not ready by namespace
rate(mysql_global_status_commands_total{
      command=~"(select)"}[5m]) * 100 > 35
      -- Rate of MySQL SELECT commands over
      last 5 minutes
-- The > 35 can be used to only display
      outliers (threshold/alert conditions)
```

8.2 Logging

Node-Level vs Application Logging: With application logging, the container of the app itself is directly communicating with the Logging Backend. In Node-Level logging, the application writes to a log-file, which the logging agent periodically reads and relays to the logging backend. The same principles also apply to side-car logging, where in Sidecar streaming, the sidecar writes to a log file, while a sidecar logging agent communicates with the backend directly.

8.3 EFK stack

Also ELK, where Logstash replaces Fluentd
Elasticsearch: Search engine written in Java, works with indices to handle huge amounts of data, especially JSON documents.
Fluentd: Data Collection written in Ruby. Collects, filters and buffers logs from different sources, tries to store logging data as JSON. Typically runs as DaemonSet on each node and forwards Data to Elasticsearch.
Kibana: Similar to Grafana, Kibana can display data living in Elasticsearch.

9 Service Mesh

9.1 Microservices

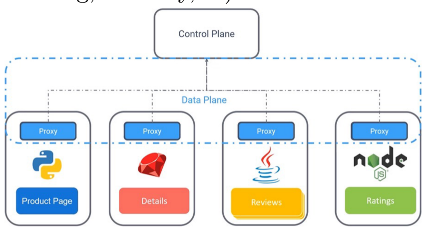
A microservice is a small, independent application that handles one specific business function and communicates with other services via APIs. This approach breaks large applications into smaller, loosely-coupled services that can be developed and deployed independently. Opposite of a Monolith.

9.2 Challenges

Communication: Services need to know all the endpoints of services they have to communicate with. A new microservice can result in many changes in existing services. **Security:** Communication in cluster is not secured, every service can talk to every other service in a cluster. Possible attackers have full access. **Monitoring:** Since services are spread out, monitoring can be tedious. Non Business logic must be added to each application, adds complexity.

9.3 Service Mesh w/ Sidecar Pattern

A Service Mesh uses Sidecars in all the Pods. The sidecars act as a proxy and also include additional functionality (Monitoring, Security, ...)



The two key components of a service mesh are: *Data Pane:* Consists of sidecar-proxies in pods which handle communication etc. (Example: Envoy) *Control Pane:* Manages and distributes configuration to data pane. (Example: Istiod)
Envoy proxies in each pod authenticate each other via **mTLS** (mutual TLS: client and server authenticate each other using TLS certificates)

9.4 Ambient Mesh

The Traffic Mesh with Sidecar pattern comes with some downsides: It is invasive, can lead to overallocation of resources per pod, breaks traffic.
Ambient mesh solves this by removing the sidecar proxies, rather providing one zTunnel per node. These zTunnels communicate with across nodes. They operate on Layer 4. zTunnels also authenticate each other using mTLS.
Additionally, optionally L7 waypoint proxies can be used for policy enforcements.

9.5 Traffic Management

In istio, K8 ingress/egress are replaced by gateways, as they allow for more control. A VirtualService binds to a gateway and determines how traffic is routed to services. DestinationRules allow to

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: my-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "example.com"
--
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: my-vs
spec:
  hosts:
  - "example.ch" # must match a gateway
      host
  gateways:
  - my-gateway
  http:
  - route:
    - destination:
        host: my-service
      weight: 99 # if a second
          destination is defined,
          weight can be used to split
          traffic accordingly
      port:
        number: 80
--
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: my-dr
spec:
  host: my-service
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

9.6 Service Resiliency

Istio allows for: **Fault Injection:** Simulate networking failures to test error-handling. **Timeouts:** Times out requests after a certain time to avoid queuing up a lot of requests. **Retries:** Configure services to retry if another service cannot be reached.
Istio also allows all the CD&D Deployment Strategies (see *General Definitions*)
Istio provides and exports Prometheus metrics per default.

9.7 Security

Authentication: Services use mTLS to authenticate each other.
Authorization: Using AuthorizationPolicies, we get Fine-grained access control using Role-based access control (RBAC).

```
# Namespace-wide mTLS enforcement using
      PeerAuthentication
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: production # Replace with
      your target namespace
spec:
  mtls:
    mode: STRICT
--
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin # applies to these pods
          in namespace foo
  action: ALLOW
  rules:
  - from:
    - source:
        namespaces: ["frontend"] # allows
            traffic from frontend NS
    to:
    - operation:
        methods: ["GET"]
        paths: ["/headers"] # allows
            traffic to
```

10 GitOps

GitOps uses Git repositories as the single source of truth for infrastructure and application configurations. Automated agents continuously monitor Git repos and sync changes to target environments automatically.

11 Site Reliability Engineering (SRE)

"Keep the site up, whatever it takes"
SL Indicators: A measurement (e.g. Response latency over a period of 10 min) **SL Objectives:** A Objective (e.g. 99.9% below 5ms) **SL Agreements:** A economic incentive (e.g. or less bonus) **Error Budget:** 1 - SLO (e.g. 100% - 99.9% = 0.1%)

11.1 Outage Handling

Goals: 1. Minimize impact (Automated, fast detection; good diagnostic information, practice remidation) 2. Prevent reoccurrence (handle event, write post-mortem)