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 environement, tested using E2E, load, accessibility tests), **Release** (snapshot of code is versioned, changes are documented), **Deploy** (release is installed onto production environement), **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 environements: Old and new versions respectively
- *Canary:* Small user group tests first
- *Feature Flag:* Deploy but activate later, can be toggeled
- *Dark Launching:* Rolling out a feature invisible for users, test its performance in the background

2 GitLab

Example GitLab pipeline:

```
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
        the 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 Environements

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 environements **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. Diffrent providers enable diffrent platforms (AWS, Azure, Kubernetes, ...). A sample in HCL (Hashicorp Configuration Language):

```
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 wont 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`

```
- 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: ßupersecret"` 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 a 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 excplicitly 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