1 Introduction Cloud Operations and DevOps

1.1 The DevOps pipelines

What is DevOps?
DevOps team focuses on CI/CD. DevOps team would like to be continuously pushing small changes out to production, monitor what happens in production, see he result of that and use that to make further improvements to the system.
Plan stage: Backlog filled, Sprint Planning
Code stage: Code is written, reviewed, and merged
Build stage: CI Pipeline builds code and runs tests
Test stage: Deploy to test environment, run automated tests
Release stage: Tag code snapshot, Document features and changes
Deploy stage: Deploy to production environment, Monitor
Operate Stage: Ensuring application runs smoothly, troubleshoot problems **Monitor Stage**: Monitor and logging

1.2 The Devops lifecycle

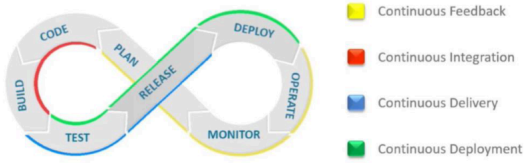

Figure 1: DevOps Stages

Continuous integration
- Commit early, commit often
- Push code to the repository in the smallest possible size
- Push code to the repository frequently
- CI testing (DAST - Dynamic Application Security Testing, Unit Tests, Code Coverage)

Continuous delivery & deployment
- Testing/Production Environments are provisioned and configured
- Continuous Delivery requires a trigger to deploy the application while in Continuous Deployment this is automated.
- Verify how the application works under load
- Strategies: Rolling deployment: Deploy in groups. Blue-Green deployment: Deploy a new version alongside the old one. Canary deployment: Deploy to a subset of users

Continuous Feedback
- Happens at the beginning and end of the continuous lifecycle
- Metrics, statistics, analytics, and feedback from the customer and teams involved are considered to continuously improve the product

1.3 DevOps in GitLab CI

GitLab CI is a Continuous Integration server built right into GitLab
- Build, test and deploy code automatically on every change
- Keep building scripting together with the code
- Works on gitlab.com or your GitLab server

Building and testing with Docker
Building a docker image from within a docker container that the GitLab Ci is running for us.
A docker registry is built into the repository (`$CI_REGISTRY`) `$CI_REGSITRY_IMAGE` is the path where the build image is going to be located. Use docker login to login into the registry using the `$CI_JOB_TOKEN`

```yaml
stages:
  - build
  - test
  - deploy

variables:
  DOCKER_IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA

build:
  stage: build
  image: docker:stable
  services:
    - docker:dind
  variables:
    DOCKER_HOST: tcp://docker:2375/
    DOCKER_DRIVER: overlay2
  before_script:
    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
    - docker info
  script:
    - docker build -t $DOCKER_IMAGE_TAG .
    - docker push $DOCKER_IMAGE_TAG

test:
  stage: test
  image: node:10
  services:
    - name: postgres:10
```

1.4 DevOps with GitLab CI: Build stages, artifacts, and dependencies

Using build stages
Each stage contains one or more jobs. Stages are run in the order they are declared. Any job failure marks the stage as failed. When you do not declare stages, the default stages are
- build
- test
- deploy

Running build steps in parallel
By assigning two jobs to the same stages, Gitlab CI runs the two jobs in parallel together. The parallel tag creates copies of the same job and runs them in parallel.
Speeding up builds with the cache
We can specify paths to cache the modules and speed up future builds. (The `$CI_COMMIT_REF_SLUG` is a variable that identifies the branch.) This Variable can be used to reference the cache folder from the branch.

```yaml
cache:
  key: $CI_COMMIT_REF_SLUG
  paths:
    - node_modules/
```

Defining artifacts
We can define artifacts of files we would like to keep. You can configure it to only keeping artifacts when a job has succeeded. Artifacts can be used in future jobs of the pipeline and downloaded via the Gitlab web interface or API. In the .gitlab-ci.yml, a new key "artifacts is defined.

```yaml
artifacts:
  paths:
    - package-lock.json
    - npm-audit.json
```

The key when can be added. 3 options
- On success
- On failure
- Always

The default is on success
Using artifacts in future stages
By default, any artifacts produced in one job is going to be available to the other jobs.
- The test job will get both artifacts (because it has no explicit dependencies)
- The test2 job will get only the artifact from the build2 (because of the dependency)

```yaml
build:
  stage: build
  script:
    - echo "I am an artifact" > artifact.txt
  artifacts:
    paths:
      - artifact.*

build2:
  stage: build
  script:
    - echo "I am also an artifact" > artifact2.txt
  artifacts:
    paths:
      - artifact2.*
```

```yaml
test:
  stage: test
  script:
    - cat artifact*.*

test2:
  stage: test
  dependencies:
    - build2
  script:
    - cat artifact2.*
```

Passing variables to builds
There are a few ways to define variables for a script:
- Per step in the pipeline definition
- Globally for all steps in the pipeline definition
- When executing a pipeline manually
- On repository level
- On group level

These variables are available as environment variables during the build.

2 DevOps with Gitlab

```yaml
default:
  image: alpine:3.19.1
------
rules:
  - if: $CI_COMMIT_BRANCH == "main"
rules:
  - changes:
    - README.md
```

2.1 Automated application deployment

2.1.1 Declaring deployment environments

Environments in Gitlab define where code gets deployed. It allows to see a history of deployments and allows to rollback to earlier version.

```yaml
deploy:
  stage: deploy
  environment:
    name: production
    url: https://example.com
```

2.1.2 Kubernetes application resources

You can build images and push them into your Gitlab container registry. Passing the Kubernetes YAML into `envsubst`, you can replace the environment variable with the actual value.

2.1.3 Deploying an application to Kubernetes

Using the Gitlab Kubernetes integration, you can easily deploy your application to Kubernetes.
- You can define multiple Kubernetes cluster to connect to and then define via "environment

scope" which environment corresponds to which Kubernetes cluster
- The Kubernetes cluster credentials are available to any job.

2.1.4 Deploy tokens and pulling secrets

We need Kubernetes authentication information so it can pull the Docker image from our Gitlab registry. Gitlab has a feature named "deploy tokens" that can be defined per project and have a defined scope like "read_registry".
Then you expose the deploy token with environment variables to the build environment.
In Kubernetes, you can create a secret with the type "docker-registry" and pass the deploy token to it.

```yaml
deploy:
  stage: deploy
  environment:
    name: production
    url: http://todo.deploy.k8s.anvard.org
  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_PULL_USER --docker-password=$KUBE_PULL_PASS
    - cat k8s.yaml | envsubst | kubectl apply -f –
```

And then reference the secret from your k8s.yaml.

```yaml
spec:
  replica: 3
  selector:
    matchLabels:
      app: todo
  template:
    metadata:
      labels:
        app: app
    spec:
      imagePullSecrets:
        - name: gitlab-auth
      containers:
        - name: todo
          image: "${DOCKER_IMAGE_TAG}"
          env:
            - name: NODE_ENV
              value: "production"
            - name: DATABASE_URL
              value: "postgres://${DBUSER}:${DBPASS}@tododb/todo"
          ports:
            - containerPort: 3000
```

2.1.5 Dynamic environments and review apps

Idea: Have a separate deployment for each branch for the devs to fiddle around with. (`todo-${DEPLOY_SUFFIX}`) This can be achived by adding a DEPLOY_SUFFIX and DEPLOY_HOST to all k8s resources. Another job is added that deletes the resources when the branch is deleted. This job can be triggered with `on:stop: job_name` in the environment section.

2.2 Application quality and monitoring

2.2.1 Integration and functional testing

To run integration tests, you can define services per Gitlab job, e.g., a Postgres database container. A service is an extra container that GitLab CI will run for us as part of our build or test process. The extra container is going to be available to the mail container.

```yaml
test-10:
  stage: test
  image: node:10
  services:
    - name: postgres:10
      alias: db
  variables:
    POSTGRES_DB: todo
    POSTGRES_USER: "${DBUSER}"
    POSTGRES_PASS: "${DBPASS}"
    DATABASE_URL: "postgres://${DBUSER}:${DBPASS}@db/todo"
  script:
    - ./ci-test.sh
```

This service will be reachable as "db" from our job script.

2.2.2 Analyze code quality

Gitlab offers automated code quality checks which can run with each pipeline. The result of the analysis will be displayed alongside the merge request. Code quality jobs usually take quite a bit of time. (`include: (pagebreak) - template: Code-Quality.gitlab-ci.yaml`)

2.2.3 Dynamic application security testing (DAST)

Gitlab can check an application "from outside" for security issues.

2.2.4 Application monitoring with Prometheus

Prometheus identifies metric endpoints from an application and scrapes periodically metrics from those endpoints. If the Kubernetes integration with Gitlab is activated, Gitlab will deploy a Prometheus server to said cluster. To define that your service has endpoints to get data from for Prometheus, you need to add annotations to the Kubernetes service. (In the kubernetes Service file)
The data can be reviewed in Gitlab directly. Spaceships are added to the timeline to indicate when a deployment happened.

2.3 Custom CI infrastructure

2.3.1 Launching dedicated runners

When using gitlab.com, your jobs are running on shared infrastructure. You can also launch dedicated runners for your project and connect them to gitlab.com. Below is the configuration page and an overview of runners assigned to the project.

When launching dedicated runners, you can assign them tags. In your pipeline definition, you can force to run the job on dedicated runners by also mentioning the tags used for dedicated runners in the pipeline.
There is different type of runners:
- Shell: Run builds directly on runner host. Useful for building virtual machine or Docker

images.
- Docker: Run builds in Docker containers. Can be used for building Docker images through "Docker in Docker" configuration.
- Docker machine: Launches virtual machines to run builds in Docker. Useful for autoscaling.
- Kubernetes: Runs builds in Docker containers inside Kubernetes. Useful to consolidate build infrastructure when using Kubernetes.

2.3.2 Custom Kubernetes runners

You can also deploy Gitlab runners from Gitlab to a Kubernetes cluster.

3 Terraform

The idea of Terraform is to help provision infrastructure (Infrastructure as code). It has support for all common cloud provider.
Terraform and Ansible are complementary. Terraform deals with the infrastructure stack, Ansible deals with the application stack.
In Terraform you interact with Provider Plugins. A provider converts the terraform syntax to something that the SDK can consume. The SDK then sends API calls to the cloud provider (for example AWS). The different plugins are managed by the community (AWS, GCP, Azure, ...).

```
provider "aws" {
  access_key = "<AWS_ACCESS_KEY>"
  secret_key = "<AWS_SECRET_KEY>"
  region     = "us-east-1"
}
```

```
resource "aws_instance" "excercise_0010" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "HelloWorld"
    Terraform = true
  }
}
```

Terraform workflow
- Write your Terraform stuff
- Let Terraform plan the changes that must be applied
- Let Terraform apply the changes

Common Terraform commands
- `terraform init` (Initialize Terraform, perform where the .tf files are located)
- `terraform plan` (tells the operator what's going to be changed)
- `terraform apply` (applies the changes)
- `terraform destroy` (This command figures out what has been done and undo all the changes / Folder Specific)

`terraform.tfstate` file
This file knows what has already been deployed. So, if you run terraform apply twice in a row, it will know that there is nothing to do. When you manually change something (e.g using the AWS web UI), terraform will revert your changes as the .tf files are the single source of truth. Terraform creates the terraform.tfstate file the first time you run the command terraform apply. Any other times it makes a 3-way diffs from the state's NEW, EXISTING and PREVIOUS to get a full picture of the current situation. State files can get corrupted (not that often, but it can happen).
References
References in Terraform are like variables. You can reference references to get actual values defined in other places of your Terraform script.
Output
Output are return values of a function. You ultimately define references, which you want to display at the end of terraform apply.

```
output "example-ip" {
  value = aws_instance.excercise_0010.public_ip
  description = "The public IP of the instance"
}
```

Data
The data is an input for the Terraform provider. It uses the cloud to look up data and feed this data into the terraform script. -> `data "aws_ami "latest_ubuntu" {...}`
Variables
If a variable is defined without a default value, it will prompt the operators to enter the value.

```
variable "port" {
  description = "The port to expose on the server"
  type = number
  default = 8080
}
```

Variables can also be overwritten as an argument or environment variable.
Organization (Best practice):
- Create one folder for each environment
- Every folder gets its own state file

Module (Code re-use)
- A module is just re-usable code
- A module can be invoked and supplied with all variables