# Building a Scalable Cloud Native Training Platform with Kubernetes

TDOC 2023 - Zander Havgaard

# $ whoami

## Zander Havgaard

- Senior Software Developer @ **Green.ai** before that **Eficode**
  - This presentations covers the last project I did with Eficode before moving on to new adventures
- Interests: `DevOps`, `Cloud Native`, `Containers`, `Orchestration`, `IaC`, `CI/CD` and more
- I have taught courses in: `Kubernetes`, `Docker`, `Helm`, `Istio`, `Git` and more
- Speaker at meetups

Email: contact@pzh.dk | zanderhavgaard@green.ai

GitHub: `@zanderhavgaard`

# Agenda

- The **rationale** behind the design and architecture of our new infrastructure.
- The **open-source technologies** that power our platform.
- The **rapid MVP development** of our platform in just two weeks --> enabled by cloud-native technologies and AI tools.
- How we **tested it in production**: delivering a DevOps summer course at the University of Southern Denmark (SDU) to nearly 100 students.
- A **discussion on the scaling bottleneck** we encountered and the strategies used to overcome it.

Format: Slides and live demonstrations of the platform

Feel free to ask questions and discuss after the talk !

These slides are on github: https://github.com/zanderhavgaard/talk-building-a-scalable-cloud-native-training-platform

# The context

Eficode provides a number of trainings to it’s customers in topics such as `kubernetes`, `docker`, `git`, `helm` and many more

Each training consists of a trainer presenting the material as well as a number of hands-on exercises, which we call the **katas**

> All of the katas live on Github and are open source! e.g. https://github.com/eficode-academy/kubernetes-katas

Students thus need an environment in which they can do the exercises without having to set up their own machines

# The "old" Infrastructure

To solve the problem of provisioning infrastructure for each training session we created an infrastructure that could be deployed with `terraform`

The project would deploy a number of `ec2` instances to `AWS` and an optional `EKS` cluster. Each `ec2` instance runs `code-server` to provide a workstation.

This was a great solution for a long time. But over time we outgrew the infrastructure:

- It was hard to maintain --> monolithic architecture with many moving parts --> changes / updates were cumbersome and time-consuming
- It was difficult to extend with new courses
- Too few people had the knowledge to work on it --> *low busfactor*
- Once an infrastructure had been deployed changes could not be made to it in-place, forcing a redeployment --> which would easily take ~30 minutes

… So it was time to invest in a new infrastructure

# Requirements for the New Infrastructure

A generic platform for running (cloud native) courses:

- Must work with all existing katas
- The trainer deploying the infrastructure should only have to read a readme to be able to deploy it
- Must deploy successfully every time
- Should be **simple** to maintain
- Everything should be declarative --> avoid running scripts to configure things
- Should be able to run in a pipeline --> for testing and automation

# New infrastructure: `k8s-infra`

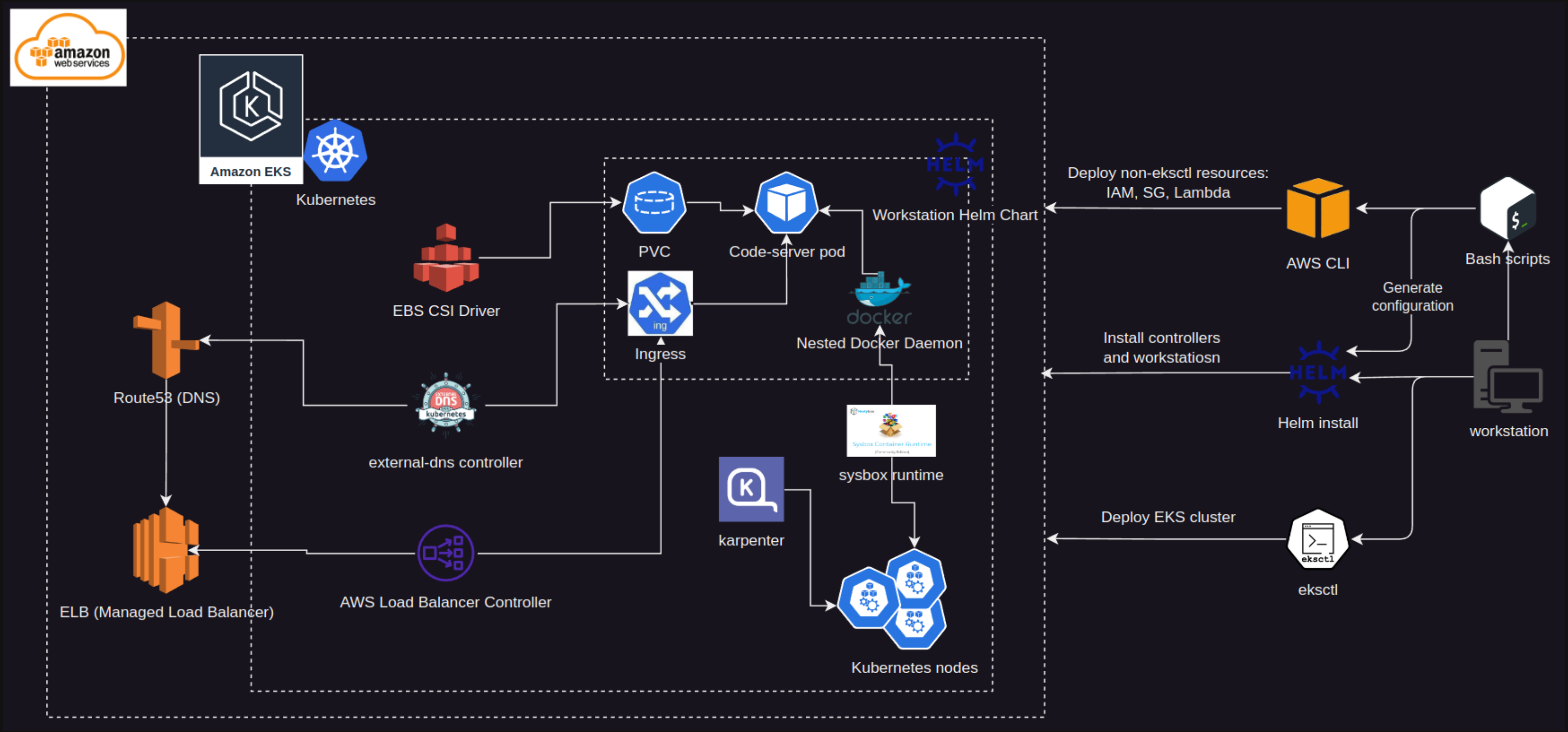
The infrastructure that I developed to meet these requirements centers around `Kubernetes`

Kubernetes allows us to declare **everything** that we want Kubernetes to control, both *inside* and *outside* of the cluster.

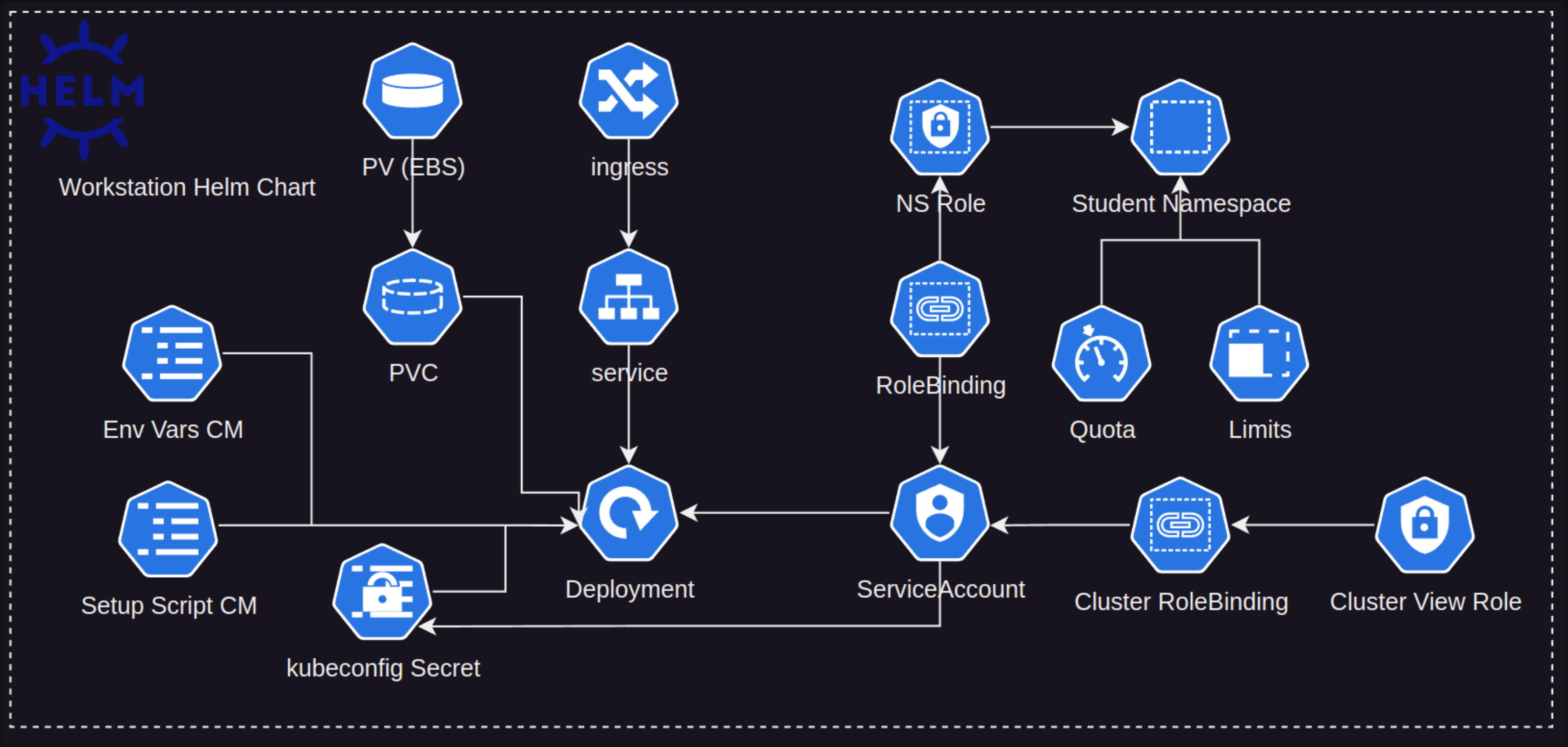
The idea is to deploy a `Kubernetes cluster` to the `cloud` using the simplest tooling possible.

When we have our cluster, we use the Kubernetes `control-plane` to automate the provisioning and configuration of the infrastructure, in a completely declarative way.

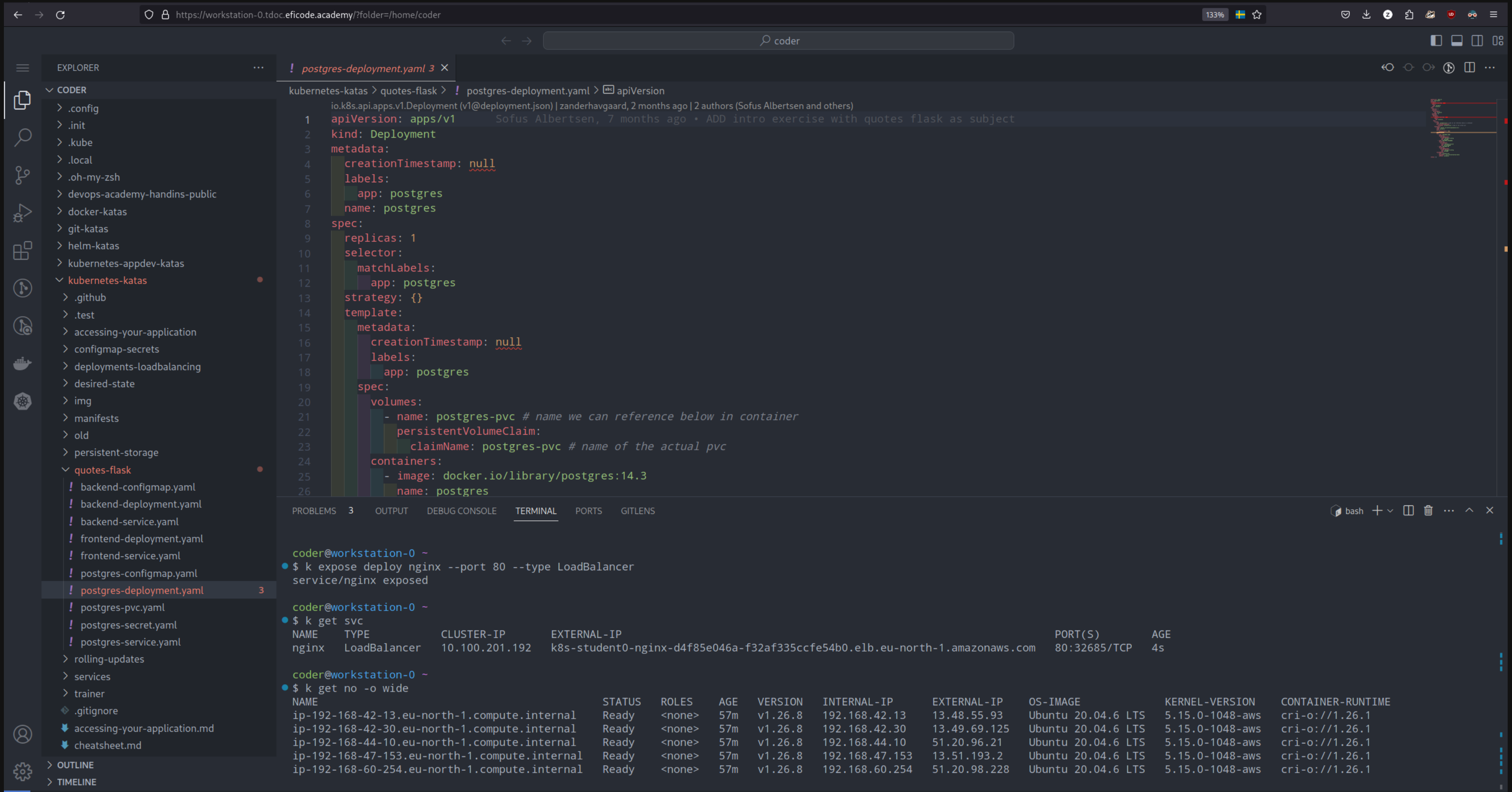
# Architecture



# Workstation components



# Demo: Workstation, kubectl, docker



# An overview of the technology that powers the platform

In the cloud native ecosystem we tend to have a tool for every problem. Here are the (important) ones that make up this platform within the categories of:

- `Provisioning`
- `Controllers`
- `Nested Containers`
- `Remote Workstation`
- `Scaling to zero`

# Provisioning

We use `EKS` on AWS to get us a Kubernetes cluster

We deploy the cluster using `eksctl`

We orchestrate the running of `eksctl` and giving our declarative specification to Kubernetes using `task`

Configuration is handled in `vars.env`

■

After we have deployed the `EKS` cluster we let Kubernetes do the rest of the actual provisioning using a number of controllers!

Trainer simply runs `$ task deploy` to deploy and `$ task destroy` to destroy the infrastructure after the training.

Taskfile.yaml

```yaml
version: "3"
# load env vars from file
dotenv:
  - "vars.env"
# load extra taskfiles
includes:
  eks: "./Taskfile.eks.yaml"
  helm: "./Taskfile.helm.yaml"
  workstations: "./Taskfile.workstations.yaml"
  ...
tasks:
  ...
  deploy:
    cmds:
      - task: eks:create-eks-cluster
      - task: eks:create-eks-public-access-sgr
      - task: eks:install-metrics-server
      - task: dns:create-route-53-records
      - task: dns:request-tls-cert
      - task: deploy-cluster-wide-resources
      - task: helm:install-sysbox
      - task: helm:install-aws-lb-controller
      ...
```

# Controllers

We install a number of controllers into the cluster to automate the provisioning of dependent resources: `dns`, `load balancing`, `persistent storage`, `auto scaling` and more.

We use: `aws-load-balancer-controller`, `external-dns`, `ebs-csi-driver`, `karpenter` to automate these needs!

> The deployed infrastructure is not static and can be scaled up and down after deployment!

workstation-ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTPS":443}]'
    alb.ingress.kubernetes.io/certificate-arn: {{ .Values.ingress.certArn }}
    alb.ingress.kubernetes.io/group.name: "code-server-workstations-{{ $lbidx }}"
    alb.ingress.kubernetes.io/healthcheck-path: "/healthz"
    external-dns.alpha.kubernetes.io/hostname: "{{ .Release.Name }}.{{ .Values.ingress.subdomain }}.{{ .Values.ingress.t
  name: code-server-{{ .Release.Name }}
  namespace: {{ .Release.Namespace }}
spec:
```

# Nested Containers

Each workstation is running in a container in a pod

We need to able to use this infrastructure for docker training, so we need to be able to run nested docker containers within each workstation container

Running nested containers has a lot of implications and considerations for `security`, `dependencies` and `functionlity`. Most of these have to do with running *containers as root with privileges*.

The most elegant solution (and the one recommended by the people behind code-server) is using `sysbox` which allows us to run nested containers using a native docker daemon in the container. `sysbox` uses `cri-o` under-the-hood to enable the functionality.

workstation-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: "code-server-{{ .Release.Name }}"
  namespace: {{ .Release.Namespace }}
spec:
  ...
  template:
    metadata:
      annotations:
        "io.kubernetes.cri-o.userns-mode": "auto:size=65536
    spec:
      runtimeClassName: "sysbox-runc" # use sysbox runtime
      containers:
      - name: code-server
        ...
        command: ["/bin/sh", "-c"]
        args:
          - |
            # root setup script, starts docker daemon
            sudo bash /entrypoint.d/root-setup.sh
            # non-root setup
            bash /home/coder/.init/setup.sh
      ...
```

# Remote Workstations

Each participant needs a workstation where they can do the exercises of training. The workstations must also have all of the necessary tools installed and configured.

`Containers` solve this problem, especially together with `Kubernetes`.

In each container we run `code-server` (https://github.com/codercom/code-server) which provides a graphical workstation that can be served using HTTP

Each workstation is deployed using a `helm chart` and all of the configuration is injected at runtime, for example each students `kubeconfig` as a Kubernetes `secret`.

deploy-workstations.sh

```
#!/usr/bin/env bash
# This script will deploy the code-server helm chart a
# specified number of times, depending on the desired count

for ((idx = 0; idx < $CODE_SERVER_WORKSTATIONS_COUNT; idx++
    RELEASE_NAME="workstation-${idx}"
    cowsay "Deploying workstation-${idx}"

    helm upgrade --install "${RELEASE_NAME}" ../code-server
        --set "index=${idx}" \
        --values ../code-server-values.yaml \
        --namespace code-server-workstations

    ...

    cowsay "Deploying kubeconfig for workstation-${idx}"
    bash create-kubeconfig-secret.sh "${idx}"

    ...

    echo "sleep 10 seconds to let aws lb controller deploy
    sleep 10

    ...
done
```

# Scaling to Zero

Since the training infrastructure is only needed during the training itself, but might run over multiple days, it is useful to able to scale the infrastructure to *zero* when not in use to save money.

This is done in the infrastructure by deploying to `AWS lambdas` which at a specified time will scale the EKS cluster `nodegroup` to 0 nodes, and then back up to the desired count again.

Since each workstation saves it's state to a `pvc` (persistent disk) we can safely *"undeploy"* the entire workstation infrastructure (save from the Kubernetes control-plane itself) and then simply scale it back up again.

All pods will remain in a `pending` state until nodes are available again.

lambda-handler.py

```
...
def scale_cluster(auto_scaling_group_name: str, desired_nod
    """Set the autoscaling group to the desired number of n

    print(f"Scaling the autoscaling group {auto_scaling_gro

    # modify the min, max and desired instance counts
    response = AUTOSCALING_CLIENT.update_auto_scaling_group
        AutoScalingGroupName=auto_scaling_group_name,
        MinSize=desired_node_count,
        MaxSize=desired_node_count,
        DesiredCapacity=desired_node_count,
    )

    if response:
        if DEBUG:
            print("response for update_auto_scaling_group r
            pprint(response)
        if response["ResponseMetadata"]["HTTPStatusCode"] =
            print(f"Successfully scaled ASG: {auto_scaling_
            return True
    return False
...
```

# Demo: Deploying Workstation to an Existing Cluster

```
k8s-infra on  main [!] on ☁ (eu-north-1) on ☸ tdoc-cluster.eu-north-1 () took 4s
❯ t workstations:deploy-single-code-server-workstation -- 16
task: [workstations:deploy-single-code-server-workstation] cowsay "Installing/upgrading workstation-16"
 _____________________________________
< Installing/upgrading workstation-16 >
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
task: [workstations:deploy-single-code-server-workstation] helm upgrade --install "workstation-16" ../code-server-w
orkstations-helm-chart \
--set "index=16" \
--values ../code-server-values.yaml \
--namespace code-server-workstations

Release "workstation-16" does not exist. Installing it now.
NAME: workstation-16
LAST DEPLOYED: Sun Oct 22 20:54:33 2023
NAMESPACE: code-server-workstations
STATUS: deployed
REVISION: 1
TEST SUITE: None
task: [workstations:deploy-single-code-server-workstation] cowsay "Generating kubeconfig for workstation-16"
 ___________________________
/ Generating kubeconfig for \
\ workstation-16            /
 ---------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
```

# Cloud Native Technology Enables Rapid Development

The new infrastructure was developed by one person (me) in *roughly two weeks of "work time"*

This was possible by utilizing **Cloud Native** technology and the ways of working that they enable

Since I, for the most part, can *declare* everything that I want, and don't have to worry about *how* to actually do it --> Kubernetes does the heavy lifting for me!

I also heavily relied on projects that have *sane defaults* so that I can follow best practices *by only configuring exactly what I need*

(of course I have a lot of knowledge of the ecosystem and had discussed the idea/design with colleagues beforehand) but the actual implementation was shockingly doable for a single person over a short period of time.

The **takeaway** I want you to have is that:

> if you buy in to the Cloud Native ecosystem it enables a lot of functionality with a relatively low barrier to entry (once you are in)

# Bonus: Using AI Tools to Speed up Development

AI development tools are all the rage these days …

… But I did use them to develop this project, specifically `Chat-GPT` with `GPT 4.0`

Which is really good for very generic code such as *"take this terraform code and translate it into a bash script that creates the same resources in AWS"* which allowed me to do speed things up further by not having to dig through documentation to figure out how to do things (that I knew I could do, just not how) with the `AWS CLI`.

> Takeaway: If you are not using AI tools to help your development you are missing out, and you will eventually be left behind by people who are.

# How we Tested the Infrastructure in Production

After the initial MVP of the platform was ready we immediately went on to test it at a summer course we were teaching at the **University of Southern Denmark** for **almost 100 students**!

The course ran over two weeks, and workstations had to be persistent for the duration of the course.

We managed to run 90 workstations (and Exercise workloads) on 15 `ec2` instances!

Down from 90 instances (one per workstation) as well as a cluster (10) ~100 machines.

While the 15 instances were bigger (and more expensive) than the ones used in the old infrastructure, we sill managed to halve the cost of running the infrastructure! As well as making the provisioning and managing much simpler.

The infrastructure was stable and performance was good.

We did have few rare issues with the docker daemon in some pods bugging out. We were unable to reproduce the issues by we suspect they are caused by `sysbox` losing connection to the `cri-o` runtime - if anyone knows anything we'd happy to hear about it!

# The Inevitable Scaling Bottleneck

The infrastructure does have a scaling bottleneck that we only discovered after "going into production"

In the infrastructure we *abuse* the `ingress` resource to provide the illusion of open ports to the docker daemons running in the workstations.

For each workstation we deploy we create a number `ingress` resources to allow connection to the `code-server` and the `nested docker containers` - this means that for each workstation we are deploying at least 7 ingresses - `90*7 = 630` and before that we were deploying even more per workstation, more than 2500 total.

The `aws-load-balancer-controller` runs into issues when having to control that many `ingress` resources.

Thus we needed to decrease the total number of `ingresses in the cluster` as well as the number of targets of each load balancer. The solution was to deploy multiple clusters and "shard" the workstations across multiple load balancers for each cluster.

Potential solutions: We could deploy multiple load balancer controllers in namespaced mode in a single cluster. We could abandon the need for arbitrary open ports to each workstation. Maybe a service mesh could solve the routing problem?

# Thank you!

Email: contact@pzh.dk | zanderhavgaard@green.ai

GitHub: `@zanderhavgaard`

Slides available on github: https://github.com/zanderhavgaard/talk-building-a-scalable-cloud-native-training-platform

```
 ____________
< Questions? >
 ------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```