CLO24 -- Kubernetes -- Niklas Häll (2025)

Index

[Project: Description here 2](#_Toc210400299)

[1. Introduction 2](#_Toc210400300)

[1.1 The entire project can be found at GitHub 2](#_Toc210400301)

[1.2 Prerequisites (knowledge, accounts and software installs) 2](#_Toc210400302)

[2. Overview of the project 3](#_Toc210400303)

[2.1 Choices, choices.. why when what how? 3](#_Toc210400304)

[2.2 General overview: Infrastructure 3](#_Toc210400305)

[2.3 General overview: Kubernetes 3](#_Toc210400306)

[2.4 General overview: CI/CD 3](#_Toc210400307)

[2.5 General overview: .NET 3](#_Toc210400308)

[2.6 General overview: LLM in this project 3](#_Toc210400309)

[2.7 The system in action 3](#_Toc210400310)

[2.7.1 bla bla flowchart 3](#_Toc210400311)

[2.9.2 blu blu flowchart 4](#_Toc210400312)

[2.9.3 User interaction flowchart 4](#_Toc210400313)

[3. Infrastructure in-depth 4](#_Toc210400314)

[3.1 Verification and troubleshooting: Infrastructure 4](#_Toc210400315)

[4. Kubernetes in-depth 4](#_Toc210400316)

[4.1 Verification and troubleshooting: AWS 4](#_Toc210400317)

[9. Where to go (improvement possibilities) 4](#_Toc210400318)

[9.1 Security 4](#_Toc210400319)

[9.2 CI/CD analysis 5](#_Toc210400320)

[9.3 Scalability analysis 5](#_Toc210400321)

[9.4 Cost analysis 5](#_Toc210400322)

[9.5 Logs and metrics 5](#_Toc210400323)

[9.6 Other areas that might be worth considering 5](#_Toc210400324)

[10. Challenges and lessons learned 5](#_Toc210400325)

[10.1 Number one 5](#_Toc210400326)

[10.2 Number two 5](#_Toc210400327)

[10.3 Number three 6](#_Toc210400328)

[11. Build timeline 6](#_Toc210400329)

[11.1 Phase one – Let the Kube-Snake loose! 6](#_Toc210400330)

[11.1.1 Phase one, step one – Local Development & Remote Infrastructure 6](#_Toc210400331)

[11.1.2 Phase one, step two – CI/CD Pipeline Implementation 7](#_Toc210400332)

[11.2 Phase two, microscopic services under the hood 7](#_Toc210400333)

[Goal: Implement GitOps tooling (Helm + ArgoCD) and deploy MongoDB as first stateful workload. 7](#_Toc210400334)

[11.2.1 Phase two, step one – who’s at the HELM anyway? 7](#_Toc210400335)

[11.2.2 Phase two, step two – ArgoCD: GitOps in Action(s) 8](#_Toc210400336)

[11.2.3 Phase two, step three – GitOps Workflow Integration 8](#_Toc210400337)

[11.2.4 Phase two, step four – MongDB: Stateful workloads 8](#_Toc210400338)

[11.2.5 Phase two, step five – .NET API Integration 8](#_Toc210400339)

[12. References and links 8](#_Toc210400340)

[11.1 Our class’ studyguide/tutorial 8](#_Toc210400341)

[11.2 LLM: Partners (and enemies) in crime 9](#_Toc210400342)

[11.3 More references? 9](#_Toc210400343)

# Project: Description here

# 1. Introduction

Lorum Ipsum

## 1.1 The entire project can be found at GitHub

Made you look! <https://github.com/mymh13/kube-snake>

## 1.2 Prerequisites (knowledge, accounts and software installs)

To replicate this system, you are expected to know and have:

* Point 1
* Point 2
* Point 3

# 2. Overview of the project

Lorum Ipsum overview-um

## 2.1 Choices, choices.. why when what how?

A rough overview would be this: Lorum Ipsum

## 2.2 General overview: Infrastructure

choices

## 2.3 General overview: Kubernetes

Choices

## 2.4 General overview: DevOps, GitOps, CI/CD, IaC

We elaborate about the various DevOps-concepts in the in-depth section below, but we really should what is mutual and what is different across these concepts.

## 2.5 General overview: .NET

choices

## 2.6 General overview: LLM in this project

Explanation here

## 2.7 The system in action

An overview of the natural flow within the system:

### 2.7.1 bla bla flowchart

1. List 1
2. List 2
3. List 3
4. List 4

### 2.7.2 blu blu flowchart

1. Further listing
2. Further listing 2

### 2.7.3 User interaction flowchart

1. More lists
2. More lists 2

Example below: can look like this

# 3. Infrastructure in-depth

Intro text

## 3.1 Verification and troubleshooting: Infrastructure

Common issues and where to look:

* list

# 4. Kubernetes in-depth

Text

Explain concepts like StatefulSet and -pvc (Persistent Volume Claim), how PVC “claims”/requests a specific storage (i.e 10 GB) and how the StatefulSet then uses the PVC to give pods stable network identity and ordered deployment. A restart of Kubernetes will mount the same PVC, which means the data is intact. This is vital since pods are ephemeral (can die).

Explain Helm – make an analogy – maybe compare it to a package manager (like apt/yum for Linux). Instead of managing individual YAML files with kubectl apply, we package everything into a Helm Chart. Elaborate how (why should be kind of self-explainable: after all we want to automate our infrastructure, no? 😊 haha). Still though: Resuability (same chart but different values for dev/staging/prod), templating, versioning, dependencies etc.

## 4.1 Verification and troubleshooting: AWS

* List

# X. Section number?: HELM and ArgoCD in-depth

Some intro here

## X.1 Let us stay on top of things, like a HELM

The structure in Helm is like building a house from a blueprint:

* Chart.yaml - The blueprint cover sheet (metadata: name, version, description)
* values.yaml - The customization form (configurable settings like replica count,

image tags, ports - default values that can be overridden, similar to properties

in a .NET class)

* templates/deployment.yaml - Construction instructions for the application

(creates the pods/containers that run your code, using Chart + values to generate

the actual Kubernetes manifest)

* templates/service.yaml - Utility hookup instructions (creates the network endpoint

that exposes your application and routes traffic to the pods, also uses Chart +

values for configuration)

The deployment creates the workers (pods), the service creates the front door (network routing).

As I am running the Helm CLI locally, I can use verify syntax locally by typing this: helm lint ./helm/charts/healthcheck

## X.2 Do not ArgoCD with me

Let us keep the house building analogy:

ArgoCD is the logistics operator. We update the blueprint (Git), then ArgoCD delivers the material and make sure the construction matches the blueprint. If someone manually changes something on-site, ArgoCD says “that is not in the blueprint!” and fixes it back.

ArgoCD is installed locally on the VM, gets its own namespace within the cluster, I install its files and then I have a backend running a server pod (API + web server), a controller (watches Git, syncs to Kubernetes), a repo server (clones Git repos) and it all runs in the namespace within the cluster.

It also has a frontend, which is a web dashboard we can access via a browser. It shows all our applications, sync status, health. It lets us manually trigger syncs, view logs, etc:

* Your Browser (anywhere on internet) –>
* <https://argocd.kube-snake.mymh.dev> ->
* Caddy reverse proxy (VM, port 443) ->
* ArgoCD server service (K8s ClusterIP 10.43.247.196:443) ->
* ArgoCD server pod (running in argocd namespace)

### X.2.1 How does ArgoCD replace GitHub Actions workflow?

ArgoCD uses POLLING, not webhooks or triggers.

1. Every 3 minutes (default), ArgoCD makes a Git request to GitHub
2. It checks: "Is the commit SHA different from what I last deployed?"
3. If YES -> Triggers a sync (pulls the new code, applies it)
4. If NO –> Does nothing, waits another 3 minutes

This means: No GitHub runners, no webhooks, no triggers. Just ArgoCD constantly checking. The polling interval can be configured, or set up webhooks for instant syncs, but polling is a good way to handle traffic and costs.

# 9. Where to go (improvement possibilities)

There are quite a lot of roads leading to Rome, but let us look at a few..

## 9.1 Security

Strengths: strengths listed

Improvements: improvements listed

## 9.2 CI/CD analysis

Strengths: strengths listed

Improvements: improvements listed

## 9.3 Scalability analysis

Strengths: strengths listed

Improvements: improvements listed

## 9.4 Cost analysis

Thoughts on the topic here

## 9.5 Logs and metrics

Strengths: strengths listed

Improvements: improvements listed

## 9.6 Other areas that might be worth considering

Lorum ipsum

# 10. Challenges and lessons learned

List here

## 10.1 Versioning funkyness

There are multiple ways of writing “versions” as I notice early on. An example from a basic Helm chart template:

apiVersion: v2

name: healthcheck

description: A Helm chart for the Kube-Snake healthcheck page

type: application

version: 0.1.0

appVersion: "1.0"

apiVersion in this case means what Chart-format we are using: v1 is the old Helm 2-format, it is now deprecated, and v2 is the Helm 3 format which is the current standard. Yes, Helm 3 is v2, but it gets more confusing:

the “version” on line 5 there is our version number for the chart itself. We increment it following a standard:

* 0.1.0 – Initial version
* 0.1.1 – Bug fix
* 0.2.0 – New feature added
* 1.0.0 – First stable release

We must follow this semantic versioning (major – minor – patch).

Then we also have the “api-version” in the Kubernetes manifests itself, for example apps/v1. This tells Kubernetes which API version to use for the specific resource type. Kubernetes has different APIs that evolve over time:

* apps/v1 – Stable API for Deployments, StatefulSets, DaemonSets
* v1 – Core API for Pods, Services, ConfigMaps
* batch/v1 – For Jobs, Cronjobs

Not the very least confusing and easy to mix up.. :rolleyes: but important lesson!

## 10.2 Number two

Wow

## 10.3 Number three

Wowowow

# 11. Build timeline

The GitHub repository will provide a more detailed history, which I also outline more specifically in modules in the markdown-documents inside the project (/docs/phase\_<phase-number>). But this serves as a rough outline of progression, to show how the system came to life.

## 11.1 Phase one – Let the Kube-Snake loose!

Goal: Establish K3s infrastructure and CI/CD pipeline with automated healthcheck deploy.

### 11.1.1 Phase one, step one – Local Development & Remote Infrastructure

Rudimentary infrastructure and a landing page:

1. Project Planning & Setup

* Laid down the general architecture and plan for the project
* Created GitHub repository and local project structure with pre-set directories
* Created architecture documentation (architecture.md)

2. Local K3s Testing (Docker Desktop)

* Verified K3s cluster working in Docker Desktop environment
* Created nginx-health-check-page.yaml deployment manifest
* Tested basic kubectl commands and internal cluster networking

3. Remote Infrastructure Preparation (Hetzner VM)

* Updated VM: apt update && apt upgrade && reboot
* Installed K3s with Traefik disabled: curl -sfL https://get.k3s.io | sh -s - --disable=traefik
* Configured kubectl access: set user permissions on kubeconfig, added KUBECONFIG to .bashrc

4. Deploy Healthcheck to Remote Cluster

* Deployed nginx healthcheck directly to VM via SSH using kubectl apply
* Verified internal service running (ClusterIP: 10.43.39.44:80)

5. External Access Configuration

* Configured Caddy reverse proxy: edited /etc/caddy/Caddyfile with subdomain block
* Added DNS A record pointing subdomain to VM IP
* Verified SSL certificate auto-provisioned by Caddy
* Confirmed external access at https://kube-snake.mymh.dev

Flow: Internet > Caddy (VM) > K3s Service > nginx container (pod)

### 11.1.2 Phase one, step two – CI/CD Pipeline Implementation

Automated deployment workflow using GitHub Actions

1. Custom Healthcheck Container

* Created custom index.html landing page with status information
* Created Dockerfile using nginx:alpine base image
* Added styles.css for consistent styling and favicon.ico

2. GitHub Actions Workflow Setup

* Created deploy-healthcheck.yml with build and deploy jobs
* Configured workflow to build Docker image and push to GitHub Container Registry (GHCR)
* Updated nginx-health-check-page.yaml to use custom GHCR image

3. Secrets & Authentication Configuration

* Added GitHub repository secrets: SSH\_HOST, SSH\_USER, SSH\_PRIVATE\_KEY
* Cloned repository to VM for kubectl access during deployment
* Resolved SSH authentication issues (key format, passphrase handling)

4. Deployment Automation & Testing

* Implemented automated deployment: Git push > Docker build > GHCR push > kubectl deploy
* Fixed image caching issue: added imagePullPolicy: Always and SHA-based tagging
* Modified workflow to use kubectl set image with commit SHA tags for reliable updates
* Verified end-to-end CI/CD pipeline working successfully

Updated Flow**:** Git Push > GitHub Actions > Build Image > GHCR > SSH to VM > kubectl set image > Pod Update

## 11.2 Phase two, microscopic services under the hood

### Goal: Implement GitOps tooling (Helm + ArgoCD) and deploy MongoDB as first stateful workload.

### 11.2.1 Phase two, step one – who’s at the HELM anyway?

Converting from raw manifests to reusable Helm charts:

1. Helm installation and Chart structure

* Installed Helm v3.19.0 on local machine and VM
* Created Helm chart structure in /helm/charts/healthcheck
* Created Chart.yaml with metadata (name, version, description)
* Created values.yaml with configureable parameters (image, replicas: 2, service conf)

2. Template Creation

* Templated deployment.yaml using {{ .Chart.Name }} and {{ .Values.\* }} syntax
* Templated service.yaml with dynamic naming and port configuration
* Validated chart with helm lint (passed with no errors)
* Tested template output with helm template to verify YAML generation

3. GitHub Actions workflow migration

* Updated workflow trigger paths to watch helm/charts/healthcheck/\*\*
* Replaced kubectl set image with helm upgrade –install
* Added --set image.tag=${IMAGE\_TAG} to override tag with commit SHA
* Added --wait and --timeout=2m for deployment verification

4. Security and Deployment

* Fixed SSH key exposure: added log-public-key: false to webfactory/ssh-agent
* Rotated compromised SSH key immediately
* Successfully deployed healthcheck with 2 replicas via Helm
* Verified both pods running and load-balanced by ClusterIP service

Updated Flow**:** Git Push > GitHub Actions > Build Image > GHCR > SSH to VM > helm upgrade –install with SHA tag 2 Pods running

Key differences from kubectl approach:

* Configurable via values.yaml instead of hardcoded manifests
* Single command deploys/updates everything
* Version-controlled chart can be reused across environments
* Foundation ready for ArgoCD GitOps automation

### 11.2.2 Phase two, step two – ArgoCD: GitOps in Action(s)

Installing and configuring continuous deployment automation:

1. ArgoCD Installation and Access

* Deployed ArgoCD v3.19.0 to K3s cluster in dedicated argocd namespace
* Created DNS A record for argocd.kube-snake.mymh.dev pointing to VM IP
* Configured Caddy reverse proxy with ClusterIP 10.43.247.196:443
* Added security headers (HSTS, X-Content-Type-Options, X-Frame-Options)
* Retrieved initial admin password from argocd-initial-admin-secret
* Successfully accessed ArgoCD UI via HTTPS

1. DNS and Network Configuration

* Discovered Caddy (running outside K8s) cannot resolve .svc.cluster.local DNS names
* Solution: Used ClusterIP addresses directly in Caddy configuration
* Applied same fix to healthcheck service (10.43.101.38:80)
* Learned: Kubernetes internal DNS only works from within the cluster

1. ArgoCD Architecture Understanding

* ArgoCD uses polling (default every 3 minutes) to check Git for changes
* No webhooks or GitHub runners required - ArgoCD actively queries GitHub
* Compares commit SHA to detect new changes
* Self-contained CD system running entirely within K8s cluster

Key Learning: ArgoCD provides GitOps without external dependencies - it polls Git, detects changes, and syncs automatically.

### 11.2.3 Phase two, step three – GitOps Workflow Integration

Transitioning from GitHub Actions deployment to ArgoCD-managed deployments:

1. ArgoCD Application Configuration

* Created Application manifest in healthcheck.yaml
* Configured source: GitHub repo mymh13/kube-snake, branch main, path healthcheck
* Configured destination: same cluster, default namespace
* Enabled automated sync policy with prune: true and selfHeal: true
* Applied manifest: kubectl apply -f argocd/applications/healthcheck.yaml

1. Self-Healing Verification

* Tested manual intervention: scaled deployment from 2 to 1 replica
* ArgoCD detected drift within seconds
* Automatically restored to 2 replicas (Git state)
* Confirmed: Git is the single source of truth, manual changes are reverted

1. GitHub Actions Workflow Refactor

* Renamed workflow: "Build and Deploy" -> "Build Healthcheck Image"
* Removed entire deploy job (SSH, Helm install steps)
* Removed helm/charts/healthcheck/\*\* from trigger paths
* Workflow now only: builds Docker image -> pushes to GHCR
* Added success message explaining ArgoCD handles deployment

1. Separation of Concerns Achieved

* CI (GitHub Actions): Builds container images on code changes
* CD (ArgoCD): Deploys Helm charts on Git changes
* ArgoCD polls Git every 3 minutes for chart updates
* Clean separation: code changes trigger builds, config changes trigger deploys

Updated Flow: Git Push -> GitHub Actions (build image only) -> GHCR | Git Push (Helm chart changes) -> ArgoCD polls -> detects change -> syncs deployment

Key Achievement: Full GitOps implementation - all deployments managed declaratively through Git, with automatic synchronization and self-healing.

### 11.2.4 Phase two, step four – MongDB: Stateful workloads

First database deployment with persistent storage:

1. First
2. Second

### 11.2.5 Phase two, step five – .NET API Integration

Testing the full stack with a simple backend service:

1. First
2. Second

# 12. References and links

## 11.1 Our class’ studyguide/tutorial

<https://cloud-developer.educ8.se/clo/4.-run-cloud-applications/3.-kubernetes/index.html>

That webpage and the content is the property of Lars Appel, I am just referring to it.

## 11.2 LLM: Partners (and enemies) in crime

Claude 4.5 and ChatGPT 5

## 11.3 More references?

Provided here