

Index

Project: Description here	3
1. Introduction	3
1.1 Project repository at GitHub	3
1.2 Prerequisites (knowledge, accounts and software installs)	3
2. Overview of the project	4
2.1 Choices, choices.. why when what how?.....	5
2.2 General overview: Infrastructure.....	5
2.3 General overview: Kubernetes architecture	6
2.4 General overview: GitOps + Deployment strategy.....	6
2.5 General overview: CI/CD pipeline	7
2.6 General overview: Application stack.....	7
2.7 General overview: Security and Secrets Management.....	7
2.8 General overview: Session persistence strategy.....	8
2.9 The system in action.....	8
2.9.1 GitOps Deployment Flowchart.....	8
2.9.2 User Interaction Flowchart	9
2.9.3 Admin (guestbook) Interaction Flowchart	9
2.10 General overview: LLM in this project	10
2.10.1 LLMs, reading through code suggestions and mistakes.....	10
3. In depth: Infrastructure in-depth	10
3.1 Verification and troubleshooting: Infrastructure	11
4. In depth: Kubernetes in-depth	11
5. In depth: GitOps + Deployment	13
5.1 Key Learnings: HELM and ArgoCD in-depth	Error! Bookmark not defined.
5.2.1 Let us stay on top of things, like a HELM.....	13
5.2.2 Further down the road, we got to StatefulSet	13
5.3 Key Learnings: Do not ArgoCD with me	14
5.3.1 How does ArgoCD replace GitHub Actions workflow?	14
6. In depth: CI/CD pipeline	14
7. In depth: Application stack.....	14
8. In depth: Security and Secrets Management.....	14

8.1 In depths: Sealed Secrets	15
9. In depth: Session persistence strategy.....	15
9.1 Key Learnings: Redis and Distributed Session Storage.....	15
9.1.1 Why This Matters in Kubernetes and the Redis-solution	16
9.1.2 Key configuration in .NET and session cookie configuration.....	16
9.1.3 The Middleware Order Trap.....	17
9.1.4 Key Takeaways.....	18
10. Where to go (system analysis and improvement possibilities).....	18
10.1 Security	18
10.2 CI/CD analysis.....	18
10.3 Scalability analysis.....	18
10.4 Cost analysis.....	19
10.5 Logs and metrics	19
10.6 Other areas that might be worth considering	19
11. Challenges and lessons learned	19
11.1 Versioning funkyness	19
11.2 Kubernetes secrets handling and automation	20
11.3 Port Forwarding	21
11.3.1 The Problem: Silent Port Conflicts	21
11.3.2 The Solution: Alternate Port Binding	21
12. Build timeline	21
12.1 Phase 1 – Let the Kube-Snake loose!	22
12.1.1 Phase 1:1 – Local Development & Remote Infrastructure	22
12.1.2 Phase 1:2– CI/CD Pipeline Implementation	22
12.2 Phase 2, microscopic services under the hood.....	23
Goal: Implement GitOps tooling (Helm + ArgoCD) and deploy MongoDB as first stateful workload.	23
12.2.1 Phase 2:1 – who's at the HELM anyway?.....	23
12.2.2 Phase 2:2 – ArgoCD: GitOps in Action(s).....	24
12.2.3 Phase 2:3 – GitOps Workflow Integration.....	25
12.2.4 Phase 2:4 – MongDB: Stateful workloads	25
12.2.5 Phase 3:1 – .NET Guestbook API: Microservice Architecture	27
12.2.6 Phase 3:2 – Guestbook API: GitOps deployment and integration	29

12.2.7 Phase 3:3 – Multi-Replica Session Persistence & Automated Image Rollout	30
13. References and links.....	32
13.1 Our class' studyguide/tutorial.....	32
13.2 LLM: Partners (and enemies) in crime	32
13.3 More references?	32

Project: Description here

1. Introduction

This project is based on an assignment in the Kubernetes-course at Campus Mölndal 2025.
The assignment had several core constraints:

- Containerization – Application must be run in containers
- Persistence – Data must survive pod restarts
- Cloud deployment – Utilize cloud infrastructure
- Multiple deployment methods – Implement at least two deployment strategies (ultimate goal: ArgoCD + Helm)
- Ingress configuration – Understand and implement reverse proxy patterns

Goal: Build a basic .NET-application with MongoDB, deployed in a Kubernetes cluster.

On top of these requirements, I added a critical personal constraint: *sustainability*.

The project must remain operational after completion and submission without ongoing costs. This meant avoiding expensive cloud services like Azure AKS with Load Balancers, which would incur continuous charges just to keep the system running.

1.1 Project repository at GitHub

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

1.2 Prerequisites (knowledge, accounts and software installs)

To replicate this system, you need..

Knowledge requirements:

- Basic understanding of Bash or other CLI tools

- Linux fundamentals (install and configure system files)
- Basic understanding of cloud service setup and configuration
- Container concepts (Docker)
- Kubernetes fundamentals
- CI/CD pipeline basics
- HTMX or at least HTML/CSS basics

Required accounts:

- GitHub (source control + container registry)
- Docker Hub (costs!) or GitHub Container Registry (free) – image store
- Cloud service provider (Hetzner, Azure or similar)
- Domain registrar (if you chose to configure DNS for subdomain redirection like I did)

Software and tools – local development:

- Docker Desktop (initial testing)
- IDE of choice (I used VS Code)
- Kubectl (Kubernetes CLI), Git CLI
- Helm (package manager)
- Kubeseal CLI (for Bitnami Sealed Secrets)

Software and tools – deployed to cluster (via Helm/manifests):

- ArgoCD (GitOps CD – continuously delivery)
- Redis (Bitnami Helm Chart – distributed session storage)
- Bitnami Sealed Secrets (encrypt secrets)
- MongoDB (StatefulSet with persistence)

2. Overview of the project

The project tree structure looks like this:

```
Kube-Snake          # Project root
|
└── .github
    └── workflows      # GitHub Actions CI/CD pipelines
|
└── apps
    ├── backend        # (Future: Snake game backend)
    ├── frontend        # (Future: Snake game frontend)
    ├── guestbook        # .NET 9 Minimal API guestbook app
    │   ├── Endpoints    # API endpoints (Auth, Messages)
    │   ├── Extensions    # Service configuration extensions
    │   ├── Helpers        # HTML rendering, authentication helpers
    │   ├── Models         # Data models (Message)
```

```

    ┌── Properties          # Project properties
    └── Services           # MongoDB service
      └── healthcheck      # System health monitoring page

  ┌── argocd
  └── applications       # ArgoCD Application manifests

  ┌── docs                # Project documentation (phases, architecture)

  ┌── helm
  └── charts
    ┌── guestbook
    └── templates         # K8s manifests (Deployment, Service, etc.)
      └── healthcheck     # Healthcheck application chart
      └── templates
    ┌── mongodb
    └── templates         # MongoDB StatefulSet chart
    └── secrets            # SealedSecrets for credentials

  ┌── k8s
  └── cert-manager        # TLS certificate management
  ┌── guestbook
  └── traefik             # Ingress controller configuration

```

2.1 Choices, choices.. why when what how?

Choices to be made initially or during the project included the below. It is not a complete list, and in-depth explanation will be found in other sections, but to name a few:

- Why K3s instead of managed Kubernetes (AKS/ESK)
- Why Hetzner Cloud over Azure/AWS/GCP
- Why HTMX instead of React/Vue/Node etc
- Why MongoDB for persistence
- Why Redis for session storage
- Why Caddy instead of Traefik/Nginx
- Why Sealed Secrets instead of external secret managers?

2.2 General overview: Infrastructure

The infrastructure is built on minimal, cost-effective components that provide production-grade reliability without ongoing cloud expenses. The foundation is a single VM running K3s, with persistent storage and automated TLS management.

- Hetzner Cloud VM (Ubuntu 22.04, 2 vCPU, 4GB RAM) – Cost-effective cloud hosting (€4.49/month – already pre-paid by other projects so this is essentially “free”)
- K3s lightweight Kubernetes distribution – Production-ready K8s without the overhead of managed services
- Local-path provisioner – K3s built-in storage provisioner for Persistent/Volumes (uses VM disk space)
- Caddy reverse proxy – Automatic TLS certificate management via Let’s Encrypt
- Domain setup – Subdomain kube-snake.mymh.dev with external DNS provider for -a-record pointing to VM IP

2.3 General overview: Kubernetes architecture

The Kubernetes cluster follows a microservices pattern with stateless application pods and stateful data stores. All components run in the default namespace, with ArgoCD managing deployments from its own namespace. The architecture enables horizontal scaling, persistent storage, and zero-downtime deployments.

- Namespaces: default (applications), argocd (GitOps controller), kube-system (k3s components)
- StatefulSets: MongoDB (persistent database), Redis (distributed session cache)
- Deployments: Guestbook (2 replicas), Healthcheck (2 replicas)
- Services: ClusterIP for internal pod-to-pod communication
- Ingress: Path-based routing - / (healthcheck), /guestbook (guestbook API)
- Persistent storage: PVCs for MongoDB data and ASP .NET DataProtection keys

2.4 General overview: GitOps + Deployment strategy

The deployment strategy combines three complementary tools to create a fully automated GitOps workflow. While each tool serves a distinct purpose, they work together to ensure that Git is the single source of truth for both application code and infrastructure configuration. Changes pushed to the main branch trigger an automated pipeline that builds, tags, and deploys new versions without manual intervention.

- Helm - Package manager for K3s manifests (templates, versioning, reusability)
- ArgoCD - Continuous delivery via Git polling (syncs cluster state with Git repo)
- GitHub Actions - CI/CD pipeline (build Docker images, tag with commit-SHA, push to GHCR)
- Trunk-based development - All changes committed directly to main branch
- Automated rollout - SHA-tagged images trigger ArgoCD sync, which pulls new image and performs rolling deployment

Key principle: Developers push code -> CI builds image -> CI updates manifests -> ArgoCD deploys. Zero manual kubectl commands in production.

2.5 General overview: CI/CD pipeline

The CI/CD pipeline is fully automated and triggered by code changes. When a developer pushes to the repository, GitHub Actions builds a new Docker image, tags it with the commit SHA for traceability, and updates the Helm values file. This GitOps approach ensures that every deployment is traceable to a specific commit and can be rolled back if needed.

- Trigger - GitHub Actions triggers on apps/guestbook/** changes
- Build – Multi-stage Dockerfile creates optimized production image (SKD stage → runtime stage)
- Tag - Image tagged with commit SHA (main-<sha>) for version tracking
- Push – Image pushed to GitHub Container Registry (GHCR)
- Update manifests – CI bot automatically updates values.yaml with new image tag
- Commit & push – CI commits updated values-file using Personal Access Token (PAT)
- ArgoCD sync – ArgoCD polls every 30 minutes, detects change, and triggers rolling deployment

Key principle: Git is the single source of truth. Developers never manually edit cluster resources—all changes flow through Git, ensuring auditability and repeatability.

2.6 General overview: Application stack

The application layer uses a minimal technology stack that prioritizes simplicity and maintainability. The backend is a .NET Minimal API that serves HTML partials instead of JSON, enabling the frontend to use HTMX for dynamic interactions without heavy JavaScript frameworks. Authentication and session management are handled server-side, with stateless cookies and distributed session storage enabling horizontal scaling.

- Backend - .NET 9 Minimal API (guestbook)
- Frontend - HTMX embedded in static HTML (healthcheck)
- Database - MongoDB StatefulSet with PVC (Persistent Volume Claim)
- Session Store - Redis (Bitname Helm Chart) for distributed sessions
- Authentication - Cookie-based auth with admin credentials in Sealed Secrets

Key principle: Hypermedia over JSON APIs. The server sends ready-to-render HTML instead of raw data, reducing client-side complexity and eliminating the need for state management libraries.

2.7 General overview: Security and Secrets Management

Security is implemented through multiple layers, ensuring that sensitive data is never exposed in Git repositories or transmitted insecurely. Secrets are encrypted at rest (when stored in Git) using Bitnami Sealed Secrets, which allows safe storage in version control while

maintaining GitOps principles. Runtime secrets are managed by Kubernetes, session cookies are configured with security flags to prevent common web vulnerabilities, and DataProtection keys are shared across replicas to ensure consistent cookie encryption.

- Sealed Secrets - Bitnami Sealed Secrets for Git-safe encryption
- Kubernetes Secrets - MongoDB credentials, admin credentials, Redis configuration
- GitHub Secrets - Personal Access Token (PAT) for CI/CD
- Cookie security – Session cookies configured with HttpOnly (prevents XSS), Secure (HTTPS only), SameSite=None (cross-origin requested)
- DataProtection keys – ASP .NET DataProtection keys stored on PersistentVolume, ensuring all replicas use the same encryption keys for cookie consistency.

Key principle: Secrets are encrypted before committing to Git. The Sealed Secrets controller running in the cluster is the only entity that can decrypt them, ensuring that even if the Git repository is compromised, secrets remain protected.

2.8 General overview: Session persistence strategy

Stateless applications require external session storage to support horizontal scaling. The default in-memory session cache only works with a single pod—when multiple replicas exist, users would be randomly routed to different pods that don't share session state. Redis solves this by providing a distributed cache accessible by all pods, enabling seamless session persistence across the cluster.

- Problem: In-memory sessions don't work with multiple replicas
- Solution: Redis as distributed cache (Bitnami Helm chart deployment)
- Session data stored in Redis (accessible by all pods via ClusterIP)
- Cookie configuration for HTTPS (SameSite=None, Secure=true)
- Horizontal scaling enabled (any pod can serve requests, query Redis)

Key principle: Pods remain stateless. All session state lives in Redis, allowing pods to be added, removed, or restarted without disrupting user sessions. This enables true horizontal scaling and zero-downtime deployments.

2.9 The system in action

An overview of the natural flow within the system:

2.9.1 GitOps Deployment Flowchart

```
Developer Push
  ↓
GitHub Actions (build + tag image)
  ↓
Push to GHCR
```

```
Update values.yaml (automated)
  ↓
ArgoCD polls Git (every 30 min)
  ↓
Detect change -> Sync
  ↓
K3s pulls new image
  ↓
Rolling deployment
```

2.9.2 User Interaction Flowchart

```
User Browser (HTTPS)
  ↓
Caddy Reverse Proxy (VM:443) (HTTPS)
  ↓
K3s Ingress
  ↓
Healthcheck Service -> Healthcheck Pod
  ↓ (HTMX request to /guestbook/api/*)
Guestbook Service -> Guestbook Pod (any replica)
  ↓
MongoDB (read/write messages)
  ↓
Redis (session validation)
  ↓
Response (HTML partial via HTMX)
```

2.9.3 Admin (guestbook) Interaction Flowchart

```
Admin Login Form
  ↓
POST /guestbook/api/login
  ↓
Validate credentials (env vars from SealedSecret)
  ↓
Store session in Redis
  ↓
Set cookie (HttpOnly, Secure, SameSite=None)
  ↓
Return admin panel HTML
  ↓
Subsequent requests validate cookie -> Redis lookup
  ↓
Any pod can serve authenticated requests
```

2.10 General overview: LLM in this project

LLMs have been used in a variety of ways, and different types of LLMs.

Claude 4.5 has been my primary code support, it suggested tools I asked for when I was looking to solve problems. Redis is one such tool. ChatGPT 5 has been a resource when I asked general questions, and ChatGPT 4.1 has had a few looks at the code too. Claude has been helpful to automatically document in the docs/phase_number.md when I finished a section – I tell it to record what was built in and it kept those for me as “logs”/reference.

I want to stress that I primarily asked the LLMs for tools and options, virtually using it as a guide when I am trying to figure out system architecture or what software tools I have available.

2.10.1 LLMs, reading through code suggestions and mistakes

I have used LLMs extensively to generate or review code for me, but I have also been reviewing everything it does. For example: several times the LLMs have created either public logfiles (GitHub) or files that are open in my repository – where sensitive account information has been posted in plain sight.

Since I am here to learn, and I want to verify that whatever code is generated for me, I have been reviewing their code and found at least a handful of worrying instances. I have given them strict instructions not to generate more than a few lines of code at a time, so I can read through them myself, and I always ask what something means if I do not recognize it.

I want to be transparent though with what usage there has been. The two primary sources of generation has been to convert my code into markdown (the phase_number.md documents) and to generate sections of code when I felt it either a) speeds me up without me losing control or overview, and b) it has complexity levels and I realize they read through text faster than I do.

3. In-depth: Infrastructure in-depth

The infrastructure layer consists of a single Hetzner Cloud VM running K3s, a lightweight Kubernetes distribution that includes built-in storage provisioning and ingress capabilities. Caddy serves as the reverse proxy, providing automatic HTTPS certificate management through Let's Encrypt without manual configuration.

K3s includes a local-path provisioner that automatically creates persistent storage on the VM's disk when pods request PersistentVolumeClaims. This eliminates the need for external storage solutions while maintaining data persistence across pod restarts. The subdomain kube-snake.mymh.dev points to the VM's public IP via an A-record, allowing Caddy to handle TLS termination and route traffic to the K3s ingress controller.

This minimal infrastructure approach keeps operational costs below €5/month (actually for free, since it shares the VM) while providing production-grade features like automatic TLS renewal, persistent storage, and container orchestration. The single-VM design is suitable for low-traffic applications and educational projects where high availability is not critical.

3.1 Verification and troubleshooting: Infrastructure

Common issues and where to look:

- K3s not starting -> Check /var/log/syslog or journalctl -u k3s, verify firewall rules allow ports 6443, 80, 443
- DNS not resolving -> Run dig kube-snake.mymh.dev, verify A-record points to correct VM IP, wait 5-60 minutes for propagation
- TLS certificate fails -> Check sudo journalctl -u caddy -f, ensure ports 80/443 are open for Let's Encrypt validation, verify DNS is correct
- PVC stuck in Pending -> Run kubectl describe pvc <name>, verify local-path provisioner is running: kubectl get pods -n kube-system
- Out of disk space -> Check df -h, clean up old images: sudo k3s crictl rmi --prune, review storage usage: du -sh /var/lib/rancher/k3s/storage/*
- Caddy not routing traffic -> Verify Caddyfile syntax: sudo caddy validate --config /etc/caddy/Caddyfile, restart: sudo systemctl restart caddy
- Firewall blocking connections -> Check rules: sudo ufw status, add missing ports: sudo ufw allow <port>/tcp

4. In-depth: Kubernetes in-depth

The Kubernetes architecture separates stateful components (MongoDB, Redis) from stateless application pods (Guestbook, Healthcheck). StatefulSets provide stable network identities and persistent storage for databases, while Deployments handle stateless workloads that can be freely scaled and restarted.

Persistent storage is managed through PersistentVolumeClaims (PVCs), which request a specific amount of storage from the cluster. When a pod restarts, K3s automatically mounts the same PVC, ensuring data survives pod lifecycle events. This is critical because pods are ephemeral—they can be killed, rescheduled, or scaled down at any time.

4.1 StatefulSets vs Deployments

StatefulSets (MongoDB, Redis):

- Stable network identity (pod names stay consistent: mongodb-0, redis-0)
- Ordered deployment and scaling (pods created sequentially)
- Persistent storage automatically attached via PVCs
- Used for databases and stateful services

Deployments (Guestbook, Healthcheck):

- Random pod names with generated suffixes (e.g., guestbook-7b8f9c-xq4z2)
- Parallel scaling (all replicas start simultaneously)
- Stateless by design (any pod can serve any request)
- Used for application workloads

4.2 Persistent Storage with PVCs

How it works:

1. Pod requests storage via PersistentVolumeClaim (e.g., 5Gi)
2. K3s local-path provisioner creates a directory on VM disk
3. PVC binds to the new PersistentVolume
4. Pod mounts the volume at specified path
5. On pod restart, same PVC is mounted (data intact)

Why this matters: Pods are ephemeral and can die at any time. PVCs ensure MongoDB data and DataProtection keys persist across restarts, scaling events, and node failures.

4.3 Verification and troubleshooting: Kubernetes

- PVC stuck in Pending -> Run kubectl describe pvc <name>, check if local-path provisioner is running: kubectl get pods -n kube-system | grep local-path
- StatefulSet pods not starting -> Check kubectl describe statefulset <name>, verify PVCs are bound, ensure sufficient disk space on VM
- ImagePullBackOff errors -> Verify image exists in GHCR: kubectl describe pod <name>, check imagePullSecrets if repository is private
- CrashLoopBackOff -> View logs: kubectl logs <pod-name>, common causes: missing environment variables, database connection failures, port conflicts
- Service not routing traffic -> Check service selector matches pod labels: kubectl get svc <name> -o yaml, verify endpoints: kubectl get endpoints <name>
- Pod evicted due to disk pressure -> Check kubectl describe node, clean up old images: k3s crictl rmi --prune, review PVC usage
- Secrets not mounting -> Verify secret exists: kubectl get secret <name>, check SealedSecret is unsealed: kubectl get sealedsecret <name>
- Liveness/Readiness probes failing -> Check probe path is correct (e.g., /guestbook/api/messages), verify application is listening on correct port, review pod logs for startup errors

5. In-depth: GitOps + Deployment

GitOps treats Git as the single source of truth for both application code and infrastructure configuration. Changes are made by committing to Git, not by running manual kubectl commands. Helm packages Kubernetes manifests into reusable charts, while ArgoCD continuously monitors Git and automatically syncs changes to the cluster. This approach ensures that the cluster state always matches what's in Git, enabling auditable deployments and easy rollbacks.

5.1 Let us stay on top of things, like a HELM

Helm is to Kubernetes what apt/yum is to Linux. Instead of managing dozens of individual YAML files, Helm packages everything into a Chart. This enables templating (same chart, different values for dev/staging/prod), versioning (rollback to previous deployments), and dependency management (automatically deploy Redis when deploying Guestbook).

Analogy: 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`

5.2 Further down the road, we got to StatefulSet

..which we defined in statefulset.yaml. This literally works like a config file (for the deployment, it is a Kubernetes configuration for running MongoDB – MongoDB has its own config files), a general understanding of the symbiosis between them could be this:

- Chart.yaml says “this is a MongoDB chart, version x.x.x”
- values.yaml then says “here are the default settings, username, password, storage size etc”
- statefulset.yaml then says “here’s how Kubernetes should deploy and run MongoDB using those settings”

5.3 Key Learnings: Do not ArgoCD with me

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 in its own namespace, `argocd`. It runs several components:

- Server pod – API + web server
- Controller – Watches Git, syncs to Kubernetes
- Repo server – Clones Git repositories

5.3.1 How ArgoCD replaces manual deployments

ArgoCD uses polling, not webhooks or triggers. Workflow:

1. Every 3 minutes (default, I set it to 30), 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 (30, in my case) 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.

6. In-depth: CI/CD pipeline

Introduction here

7. In-depth: Application stack

Introduction here

8. In-depth: Security and Secrets Management

Introduction here

8.1 In depths: Sealed Secrets

To handle [Kubernetes Secrets and the Helm charts](#) in a way that does not expose credentials, I decided to use Sealed Secrets. What is it?

- A Kubernetes controller that runs in your cluster
- You encrypt secrets locally with a public key
- Only the controller (with the private key) can decrypt them
- You commit the encrypted SealedSecret to Git safely
- The controller automatically creates the actual Secret in K3s

The process of working with this is simply to SSH to the VM and run this:

```
kubectl apply -f https://github.com/bitnami-labs/sealed-secrets/releases/download/v0.24.5/controller.yaml
```

Then we need a kubeseal CLI tool locally on our workstation. I downloaded kubeseal, set the .exe in a /bin/kubeseal directory behind the user and added it to Windows PATH. Now I can use kubeseal commands in my Bash terminal. There is a flurry of commands after this, but the timeline is roughly:

1. Copy the kubeconfig from the VM to your workstation
2. Fix the server IP in K3s-config
3. Verify it changed (grep "server:" ~/.kube/k3s-config)
4. Backup current config, then merge the local and VM-K3s-configs
5. See available contexts (should see docker-desktop and default = K3s)
6. Switch to K3s (kubectl config use-context default)
7. Verify (kubectl get nodes)
8. Create a secret yaml-file hidden locally
9. Encrypt it so only the K3s-cluster can decrypt it
10. Verify (I cat'd the secret, noticed the SealedSecret tag + encrypted string, then deleted the cat'd file)

That was it, now there is a connection and encryption running. At this stage I updated the charts to reflect these changes too.

9. In-depth: Session persistence strategy

Introduction here

9.1 Key Learnings: Redis and Distributed Session Storage

Let's use a hotel analogy:

Imagine a hotel with multiple front desks (pods). A guest (user) checks in at Desk A, gets a room key (session cookie), and goes to their room. Later, they come back to ask for towels, but this time they talk to Desk B.

Without Redis (in-memory cache):

- Each desk has its own guest registry (local memory)

- Desk B says "Sorry, we don't have you in our system" because they only know about guests who checked in at Desk B
- Guest has to check in again every time they talk to a different desk

With Redis (distributed cache):

- All desks share one central guest registry (Redis server)
- Guest checks in at Desk A -> entry added to central registry
- Guest talks to Desk B -> Desk B checks central registry, sees the guest is registered, provides towels
- No matter which desk the guest talks to, their session is recognized

9.1.1 Why This Matters in Kubernetes and the Redis-solution

When you scale an application to multiple replicas (pods), Kubernetes uses a load balancer to distribute incoming requests. Without shared session storage:

- Request 1 (login) -> Pod A -> Session stored in Pod A's memory
- Request 2 (post message) -> Pod B -> Pod B has no knowledge of the session -> 401 Unauthorized

This is exactly what happened with our guestbook.

The solution / How Redis Fixes It:

Redis is a separate service that all pods can connect to:

1. User logs in -> Session data written to Redis (not pod memory)
2. User posts message -> Any pod can read session data from Redis
3. User logs out -> Session deleted from Redis

So it is a "consolidated temporary memory", a shared cache.

9.1.2 Key configuration in .NET and session cookie configuration

```
services.AddStackExchangeRedisCache(options =>
{
    options.Configuration = "guestbook-redis-master:6379";
});
```

To guestbook pods: "Store session data at this Redis address, not in your own memory."

For cookies to work correctly in Kubernetes with HTTPS and path-based routing:

- HttpOnly = true -> JavaScript can't access the cookie (security)
- SameSite = SameSiteMode.None -> Cookie sent in cross-origin requests (required for modern browsers)
- Secure = true -> Cookie only sent over HTTPS
- Path = "/guestbook" -> Cookie only sent for requests under `/guestbook` path

Without these settings, browsers may reject cookies or session state may not persist correctly. I set the cookie settings in the session handling at ServiceExtensions.cs.

9.1.3 The Middleware Order Trap

In ASP.NET Core, middleware order is critical:

```
app.UsePathBase("/guestbook"); // 1. Set base path
app.UseSession();           // 2. Enable session BEFORE endpoints
app.MapGuestbookEndpoints(); // 3. Map endpoints
```

If you call UseSession() after endpoint mapping, session data won't be available when processing requests. Duh. 😊 But I had a challenge here actually:

I had set the correct order, in a sense, because I build using the Extension Method Pattern – this means instead of having a long Program.cs that initiate all mechanics, I break out segments into their own classes and methods and initiate settings there. In the case of Session, I have this:

```
// Map endpoints using extension method
app.MapGuestbookEndpoints();
```

The Endpoints were set in the MapGuestbookEndpoints.cs – EndpointExtensions class – with this method:

```
public static WebApplication MapGuestbookEndpoints(this WebApplication app)
{
    // Session middleware (should be before endpoints)
    app.UseSession();

    // Map endpoint groups
    app.MapMessageEndpoints();
    app.MapAuthEndpoints();

    return app;
}
```

This was not good enough, the Session was initialized too close to the Endpoints – and this is just me guessing but I have seen this behaviour before – many services need an initialization-time, like when you start your computer. But the process calculations can sometimes go very fast and simultaneously, so if the MapMessageEndpoints start a fraction after UseSession but it manage to finish before UseSession: you fire them up in the “wrong” order.

An easy solution that often works is to make some kind of process separation between them; in this case I simply moved the app.UseSession() into Program.cs and put it just ahead of app.MapGuestbookEndpoints(). Guess what? It was enough to solve my problem. The only

difference technically is that we have one process in-between them → the redirection through the MapGuestbookEndpoints method! Silly but it works.

9.1.4 Key Takeaways

- In-memory session storage does NOT work with multiple replicas - Each pod has its own memory
- Redis (or any distributed cache) is essential for horizontal scaling. - All pods share session state.
- Cookie configuration matters. - Modern browsers enforce strict cookie policies—configure correctly for HTTPS.
- Middleware order matters. - Session middleware must run before endpoints.
- Health probes must use correct paths. - Include base path in probe configuration (e.g., `/guestbook/api/messages`).

Analogy recap: Redis is the hotel's central guest registry. Without it, each front desk (pod) only knows about guests who checked in there. With it, all desks share knowledge, and guests (users) have a seamless experience no matter which desk (pod) they interact with.

10. Where to go (system analysis and improvement possibilities)

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

10.1 Security

Strengths: strengths listed

Improvements: improvements listed

10.2 CI/CD analysis

Strengths: strengths listed

Improvements: improvements listed

10.3 Scalability analysis

Strengths: strengths listed

Improvements: improvements listed

10.4 Cost analysis

Thoughts on the topic here

10.5 Logs and metrics

Strengths: strengths listed

Improvements: improvements listed

10.6 Other areas that might be worth considering

Lorum ipsum

11. Challenges and lessons learned

List here

11.1 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!

11.2 Kubernetes secrets handling and automation

This one is causing some interesting problems. In the Helm chart values.yaml, we set the credentials for (among other things) MongoDB. We could set the secrets like this:

```
auth:  
  rootUsername: admin  
  rootPassword: changeme123 # CHANGE THIS in production!  
  database: snake_game
```

But that would mean our secrets would be exposed in our GitHub repo. Not good. We also need to make sure we do not put template values here that we override, because that would mean if the override ever failed, then our real credentials will be overwritten by the template values!

There are multiple solutions:

1. Sealed Secrets
 - Encrypt secrets before committing to Git
 - Only K3s can decrypt them
 - Tool example: Bitnami Sealed Secrets
2. External Secrets Operator:
 - Store secrets in a vault (Azure Key Vault, AWS Secrets Manager, etc.)
 - K3s pulls them at runtime
 - Git only contains references, not actual secrets
3. Manual Secret creation (Simple):
 - Remove secret.yaml from Helm chart
 - Manually create the Secret in K3s once via SSH
 - Never commit it to Git
 - Helm chart just references it
4. values.yaml override (Quick fix):
 - Keep placeholder values in Git (changeme123)
 - Override with real values during deployment using --set flags
 - Not ideal solution

My choice of solution:

Normally I would want the most robust and long-term solution. Since this is a small project, I am tempted to do option #3 here, but since option #1 is more robust and better in the long run, I will choose [Sealed Secrets](#). It will cause me more headache to set up but it is better practice.

11.3 Port Forwarding

This one is causing some interesting problems. In the Helm chart values.yaml, we set the credentials for (among other things) MongoDB. We could set the secrets like this:

11.3.1 The Problem: Silent Port Conflicts

During local development testing, we encountered authentication failures where `mongosh` could successfully authenticate to MongoDB using credentials `user:password`, but the .NET MongoDB.Driver consistently failed with `MongoAuthenticationException: Unable to authenticate using sasl protocol mechanism SCRAM-SHA-1`.

Initially, we suspected connection string parsing issues, URL encoding problems, or SCRAM mechanism mismatches between the MongoDB server and the .NET driver. After extensive troubleshooting including creating multiple test users, trying different authentication mechanisms (SCRAM-SHA-1 vs SCRAM-SHA-256), and using `MongoClientSettings` instead of connection strings, the root cause was discovered: MongoDB's default port 27017 was already bound locally by either a local MongoDB instance or a stale `kubectl port-forward` process.

11.3.2 The Solution: Alternate Port Binding

The fix was to use an alternate local port for the port-forward tunnel: `kubectl port-forward svc/mongodb-service 27018:27017 -n default`. This ensured the tunnel actually forwarded traffic to the K3s MongoDB instance instead of routing to the conflicting local process.

The .NET application's `.env` file was updated to use `mongodb://user:password@localhost:27018/?authSource=admin` for local development, while production deployments in the cluster use `mongodb://user:password@mongodb-service:27017/?authSource=admin` for direct service-to-service communication.

Key takeaways: always verify `kubectl port-forward` shows `Forwarding from 127.0.0.1:XXXX` on startup, check for port conflicts using `netstat -ano | findstr :27017`, use non-standard ports for local development to avoid conflicts, and remember that authentication errors can sometimes indicate connectivity issues rather than credential problems.

12. 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.

12.1 Phase 1 – Let the Kube-Snake loose!

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

12.1.1 Phase 1:1 – 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)

12.1.2 Phase 1:2– 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

12.2 Phase 2, microscopic services under the hood

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

12.2.1 Phase 2:1 – 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 configurable 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

12.2.2 Phase 2:2 – 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
2. 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
3. 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.

12.2.3 Phase 2:3 – 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
2. 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
3. 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
4. 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.

12.2.4 Phase 2:4 – MongoDB: Stateful workloads

First database deployment with persistent storage:

1. MongoDB Helm Chart Creation
 - Created Helm chart structure in mongodb
 - Configured StatefulSet (not Deployment) for stable pod identity

- Created Chart.yaml with metadata (MongoDB v0.1.0, app version 7.0)
 - Created values.yaml with configurable parameters (1 replica, 5Gi storage, resource limits)
 - Templated statefulset.yaml, service.yaml, and sealedsecret.yaml
 - Validated chart with helm lint (passed) and helm template (verified output)
2. Sealed Secrets Implementation
 - Installed Bitnami Sealed Secrets controller (v0.24.5) in K3s cluster
 - Installed kubeseal CLI tool (v0.24.5) locally on Windows machine
 - Configured local kubectl to connect to K3s cluster from local machine (merged kubeconfig)
 - Created plain Secret with MongoDB credentials (username, password, database)
 - Encrypted Secret using kubeseal with cluster's public key
 - Generated SealedSecret with encrypted data safe for public Git repository
 - Removed plaintext credentials from values.yaml and temporary files
 3. StatefulSet and Storage Configuration
 - StatefulSet provides stable network identity: mongodb-0.mongodb-service.default.svc.cluster.local
 - VolumeClaimTemplate automatically creates PersistentVolumeClaim per pod
 - 5Gi storage using K3s local-path storage class with ReadWriteOnce access
 - Headless Service (clusterIP: None) for StatefulSet DNS resolution
 4. ArgoCD Application Manifest
 - Created Application manifest in mongodb.yaml
 - Configured source and destination
 - Enabled automated sync policy with prune: true and selfHeal: true
 - Ready for deployment via kubectl apply -f argocd/applications/mongodb.yaml

Updated Flow: MongoDB credentials encrypted locally -> SealedSecret committed to Git -> ArgoCD syncs chart -> K3s Sealed Secrets controller decrypts -> MongoDB pods start with credentials -> PVC attached to pod -> Data persists on VM disk

Key Learnings:

- StatefulSets vs Deployments: StatefulSets provide stable pod identities (mongodb-0, mongodb-1) required for databases, while Deployments treat pods as interchangeable
- Persistent Storage: VolumeClaimTemplates auto-create PVCs that survive pod deletion/recreation, ensuring data persistence
- Sealed Secrets Security: Credentials encrypted with cluster's public key before Git commit; only cluster's private key can decrypt them
- Headless Services: Required for StatefulSets to provide direct pod DNS resolution without load balancing

Security Achievement: MongoDB credentials encrypted end-to-end - safe to store in public GitHub repository, only K3s cluster can decrypt and use them.

11.2.4.1 Let us glance at ArgoCD now with the MongoDB-implementation:

healthcheck

- Project: default
- Labels:
- Status: Healthy Synced
- Repository: <https://github.com/mymh13/kube-snake.git>
- Target Revi...: main
- Path: helm/charts/healthcheck
- Destination: in-cluster
- Namespace: default
- Created At: 10/06/2025 14:22:05 (21 hours ago)
- Last Sync: 10/07/2025 10:49:30 (6 minutes ago)

mongodb

- Project: default
- Labels:
- Status: Healthy Synced
- Repository: <https://github.com/mymh13/kube-snake.git>
- Target Revi...: main
- Path: helm/charts/mongodb
- Destination: in-cluster
- Namespace: default
- Created At: 10/07/2025 10:51:08 (4 minutes ago)
- Last Sync: 10/07/2025 10:51:09 (4 minutes ago)

APP HEALTH Healthy

SYNC STATUS Synced to main (02a9725) Sync OK

Auto sync is enabled.

Author: Niklas H <156240330+mymh13@users.noreply.github.com>

Comment: Feature: Add MongoDB Helm chart with Sealed Secre...

LAST SYNC Sync OK to 02a9725 Sync OK

Succeeded 7 minutes ago (Tue Oct 07 2025 10:51:09 GMT+0200)

Author: Niklas H <156240330+mymh13@users.noreply.github.com>

Comment: Feature: Add MongoDB Helm chart with Sealed Secre...

mongodbservice (7 minutes) → mongodb (7 minutes) → mongodbdata-mongodb-0 (7 minutes) → mongodb-0 (7 minutes, running 1/1) → controllerrevision (7 minutes, rev:1) → mongodb-credentials (7 minutes) → sealedsecret (7 minutes) → mongodb-credentials (7 minutes)

Lookin' good Mister Kotter!

12.2.5 Phase 3:1 – .NET Guestbook API: Microservice Architecture

The guestbook feature was initially planned as part of the healthcheck application, but I extracted it into a standalone .NET API service following microservices architecture.

This separation provides clear boundaries between infrastructure monitoring (healthcheck) and user content management (guestbook), enables independent scaling and deployment, demonstrates technology diversity (.NET alongside Go/Python), and aligns with the GitOps architecture.

This decision creates a more realistic production setup that showcases proper service decomposition. Also, I like having a re-usable separate bit of code that suits this project.

1. Project Structure and Dependencies

- Created .NET 9 Minimal API project in guestbook with dedicated solution
- Added MongoDB.Driver NuGet package (v2.28.0) and DotNetEnv for configuration

- Organized code structure: Models/, Services/, Endpoints/, Helpers/, Extensions/
 - Implemented dependency injection, services registered as singletons, endpoints mapped via extension methods, separation of concerns between data access (Services), business logic (Endpoints), and infrastructure (Program.cs)
2. MongoDB Integration and Data Models
 - Implemented Message model (Id, Text, CreatedAt, CreatedBy)
 - Created MongoDbService reading connection details from environment variables
 - Built CRUD operations: GetRecentMessagesAsync(), CreateMessageAsync(), DeleteMessageAsync()
 3. API Endpoints and Authentication
 - Implemented minimal API endpoints returning HTML partials for HTMX
 - Public: GET /api/messages (last 10 messages as HTML)
 - Admin: POST /api/login, POST /api/messages, DELETE /api/messages/{id}, POST /api/logout
 - Configured cookie-based sessions using ASP.NET Core Session middleware
 4. Security Configuration
 - Created Kubernetes Secrets for credentials (gitignored, never committed)
 - Generated .template.yaml files as safe documentation
 - Configured .env for local development (gitignored)
 - Updated .gitignore to prevent credential exposure
 5. Local Development and Testing
 - Configured port-forward to K3s MongoDB ([resolved port conflicts - see 10.3](#))
 - Tested endpoints with curl, verified MongoDB connectivity
 - Validated authentication and session management

12.2.5.1 Quick overlap into testing the guestbook

6. Containerization and Helm Chart
 - Created multi-stage Dockerfile (sdk build -> aspnet runtime, optimized image size)
 - Added .dockerignore for build optimization (exclude bin/, obj/, .env files)
 - Built and tested Docker image locally (verified app starts on port 8080)
 - Created Helm chart structure with Chart.yaml, values.yaml, and templates/
 - Configured Deployment template with environment variables from Kubernetes Secrets
 - Configured Service template (ClusterIP, port 80 -> 8080)
 - Added health probes (liveness and readiness on /api/messages endpoint)
 - Created ArgoCD Application manifest for GitOps deployment
 - Chart ready for deployment, awaiting GitHub Actions CI workflow
7. HTMX Integration with Healthcheck
 - Updated healthcheck index.html with guestbook section
 - Added HTMX script reference (CDN: htmx.org v1.9.10)
 - Implemented message list with auto-refresh (hx-get every 30s)
 - Added admin login form with session management

- Configured HTMX to swap HTML partials from API endpoints
- Awaiting cluster deployment to test service-to-service communication

Flow for this standalone API:

Local: .env -> .NET API -> Port-forward (27018:27017) -> MongoDB service -> HTML partials

Production (Pending): GitHub Actions -> GHCR -> ArgoCD -> K3s -> Sealed Secrets -> .NET pod -> mongodb-service:27017

Key Learnings:

- Port-forward conflict resolution during local development

Security Achievement: Zero credentials in Git - all sensitive data in gitignored .env (local) or Kubernetes Secrets (production), with templates providing documentation.

12.2.6 Phase 3:2 – Guestbook API: GitOps deployment and integration

This phase focused on securely deploying the guestbook API, integrating it with the healthcheck page, and validating the GitOps workflow in a Kubernetes environment.

1. Helm Chart & ArgoCD Setup
 - Finalized Helm chart for guestbook API, ensuring correct environment variable and secret references.
 - Deployed guestbook service to cluster using ArgoCD, verified healthy sync and rollout.
2. Sealed Secrets & Secure Configuration
 - Used Bitnami Sealed Secrets for MongoDB credentials, referenced securely in deployment.
 - Confirmed no sensitive data in git; all secrets managed via Kubernetes.
3. Healthcheck Integration
 - Updated healthcheck index.html to embed guestbook via HTMX.
 - Verified service-to-service communication and correct routing in Ingress.
4. Troubleshooting & Validation
 - Resolved probe failures and port mismatches in deployment.
 - Fixed redirect loop by clearing browser cache and confirming ingress/app path configuration.
 - Compared healthy/unhealthy pod specs to ensure stable rollout.
5. Final Verification
 - Confirmed guestbook, healthcheck, and MongoDB apps are healthy and synced in ArgoCD.
 - Documented lessons learned and updated phase_three.md for future reference.

Key Outcome: Guestbook API is now securely deployed, integrated with healthcheck, and managed via GitOps—ready for admin features and further enhancements.

ArgoCD looks quite nice now, four nodes all green and happy!

The image shows four separate ArgoCD application cards arranged horizontally. Each card has a title, a summary table with details like Project, Labels, Status, Repository, Target Ref., Path, Destination, Namespace, Created At, and Last Sync, and three buttons at the bottom: SYNC, REFRESH, and DELETE.

- guestbook**: Project: default, Labels: none, Status: Healthy Synced, Repository: https://github.com/mymh13/kube-snake.git, Target Ref.: HEAD, Path: helm/charts/guestbook, Destination: in-cluster, Namespace: default, Created At: 10/08/2025 00:04:31 (12 hours ago), Last Sync: 10/08/2025 11:03:01 (35 minutes ago).
- healthcheck**: Project: default, Labels: none, Status: Healthy Synced, Repository: https://github.com/mymh13/kube-snake.git, Target Ref.: main, Path: helm/charts/healthcheck, Destination: in-cluster, Namespace: default, Created At: 10/06/2025 14:22:05 (2 days ago), Last Sync: 10/08/2025 08:46:54 (3 hours ago).
- mongodb**: Project: default, Labels: none, Status: Healthy Synced, Repository: https://github.com/mymh13/kube-snake.git, Target Ref.: main, Path: helm/charts/mongodb, Destination: in-cluster, Namespace: default, Created At: 10/07/2025 10:51:08 (a day ago), Last Sync: 10/08/2025 08:26:53 (3 hours ago).
- root-applications**: Project: default, Labels: none, Status: Healthy Synced, Repository: https://github.com/mymh13/kube-snake.git, Target Ref.: HEAD, Path: argocd/applications, Destination: in-cluster, Namespace: argocd, Created At: 10/08/2025 00:03:52 (12 hours ago), Last Sync: 10/08/2025 08:26:59 (3 hours ago).

Let us also have a look at the web front, we can see an early version of the guestbook:

The screenshot shows the Kube-Snake application's landing page. It features a large logo and the text "Kube-Snake". Below it, the page health status is listed as "HEALTHY". The page then details its architecture: "K3s cluster with GitOps (ArgoCD), 2 replicas, horizontal scaling.", "CI/CD: GitHub Actions → GHCR → ArgoCD → K3s deployment.", "Stack: MongoDB StatefulSet (5Gi persistent) → .NET 9 API → HTMX frontend.", and "Security: Sealed Secrets, ClusterIP service isolation.".

At the bottom, there is a "Guestbook" section with the heading "Loading messages...". It includes a "Guestbook" icon, a "Admin" link, and a login form with fields for "Username" and "Password" and a "Login" button.

12.2.7 Phase 3:3 – Multi-Replica Session Persistence & Automated Image Rollout

This phase addressed two critical production requirements: enabling session persistence across multiple replicas and automating the image deployment workflow to eliminate manual interventions.

1. Session Persistence Problem Discovery
 - Scaled guestbook deployment to 2 replicas for high availability.

- Discovered authentication failures: login worked, but subsequent requests (posting messages, logout) returned 401 Unauthorized.
 - Root cause: Session state stored in-memory (`AddDistributedMemoryCache()`) was local to each pod.
 - When load balancer routed requests to different pods, session cookies couldn't be validated.
2. Redis Distributed Cache Implementation
 - Deployed Redis via Bitnami Helm chart: `helm install guestbook-redis bitnami/redis --set auth.enabled=false`
 - Added Microsoft.Extensions.Caching.StackExchangeRedis NuGet package to .NET project.
 - Replaced in-memory cache with Redis configuration in `ServiceExtensions.cs`:
 - Session cookie configured with `SameSite=None` and `Secure=true` for HTTPS
 - Cookie path set to `/guestbook` to match `UsePathBase` configuration
 - Ensured `app.UseSession()` is called before endpoint mapping in `Program.cs`
 3. Session Persistence Validation
 - Tested with 2 replicas: login, post message, logout—all operations successful.
 - Verified session state persists regardless of which pod handles the request.
 - Confirmed Redis stores session data accessible by all guestbook pods.
 4. Automated Image Tagging & Deployment**
 - Problem: ArgoCD didn't detect new images with `:latest` tag, requiring manual `kubectl rollout restart`.
 - Solution: Implemented SHA-based image tagging in GitHub Actions workflow:
 - Build and push image with commit SHA tag (e.g., `main-78ec86c`)
 - Automatically update `'helm/charts/guestbook/values.yaml'` with new tag
 - Configure Git credentials for CI Bot to commit and push changes
 - ArgoCD detects manifest change and triggers automatic sync
 - Workflow step added:

```

name: Update Helm values.yaml with new image tag
run: |
  SHA=$(echo "${{ github.sha }}" | cut -c1-7)
  sed -i "s/tag: \".*\"/tag: \"$SHA\"/" helm/charts/guestbook/values.yaml
  git config --global user.email "ci-bot@example.com"
  git config --global user.name "CI Bot"
  git add helm/charts/guestbook/values.yaml
  git commit -m "Update guestbook image tag to $SHA"
  git push

```

5. Health Probe & Path Configuration
 - Fixed readiness/liveness probes to use correct path: `/guestbook/api/messages`
 - Resolved probe timeout errors that prevented pods from becoming ready
 - Ensured all HTMX requests use `/guestbook/api/*` paths consistently

6. End-to-End Workflow Validation

- Tested complete pipeline: Git push -> image build -> values.yaml update -> ArgoCD sync -> pod rollout
- Verified no manual intervention required for deployments
- Confirmed image tag in values.yaml matches pushed image in GHCR
- Validated GitOps principles: all changes tracked in Git, declarative state management

Key Outcome: Guestbook now supports horizontal scaling with session persistence via Redis, and deployments are fully automated through GitOps workflow. No manual rollouts required—ArgoCD handles everything from Git manifest changes.

13. References and links

13.1 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.

13.2 LLM: Partners (and enemies) in crime

Claude 4.5 and ChatGPT 5

13.3 More references?

Provided here