🌱 Phase 4 : Local Kubernetes Deployment - Guide & Installation

Phase 4 = Taking your working app (Phase 3) and packing it into boxes (Docker), placing those boxes into an automated factory (Kubernetes), and controlling the factory with a remote control (Helm).

NOTE: Install everything first and then create in detail specifications to proceed further! This is just a guide to help you understand what is happening in Phase 4!

🧠 Mental Model (IMPORTANT – memorize this)

Think in **4 layers**:

```
Your Code
   ↓
Docker (boxes)
   ↓
Kubernetes (factory)
   ↓
Helm (remote control)
```

We'll now walk **top → bottom**, step by step.

🧩 STEP 0: What you already have (Phase 3)

You already built this 👇

```
Browser (User)
    ↓
Next.js Frontend (UI)
    ↓
FastAPI Backend (API + AI)
    ↓
PostgreSQL (Neon DB)
```

Right now:

- It runs **locally**
- Manually started
- Not scalable
- Not production-safe

Phase 4's job is to **professionalize** this.

📦 STEP 1: Docker — "Put everything in boxes"

Why Docker?

Because Kubernetes **ONLY understands containers**.

So first:
👉 **We turn frontend and backend into Docker images**

Backend Docker flow (FastAPI)

```
FastAPI Code
    ↓
Dockerfile
    ↓
```

```
Docker Image (todo-backend)
```

Inside that image:

- Python 3.12
- Your FastAPI app
- All dependencies
- Runs as non-root user
- Starts using: fastapi run

Result:

> “This backend can run anywhere. Same behavior. Always.”

Frontend Docker flow (Next.js)

```
Next.js Code
   ↓
Dockerfile
   ↓
Docker Image (todo-frontend)
```

Important idea:

- Build happens once
- Output is **standalone**
- Final image is small & fast

Result:

> “This frontend is lightweight and production-ready.”

🔑 Key idea of Step 1

After Docker:

- Your app is no longer “code”
- It is now **two images**:
    - `todo-backend`
    - `todo-frontend`

INSTALLATION:

```
**Docker**** (24+)
  ```bash
  # Verify installation
  docker --version
  ```
  Install: https://docs.docker.com/get-docker/
```

🏭 STEP 2: Kubernetes — “Run the boxes automatically”

Docker gives boxes.
Kubernetes decides:

- how many boxes
- when to restart
- who gets traffic
- what happens if one dies

Kubernetes flow (VERY IMPORTANT)

`Docker Images`

```
    ↓
Pods (running containers)
    ↓
Deployments (manage pods)
    ↓
Services (networking)
    ↓
Ingress (public access)
```

Let's break that slowly.

2.1 Pods — "Run the container"

A **Pod** = one running container.

But:
❌ Pods die
❌ Pods restart
❌ Pods are not stable

So we **never use Pods directly**.

2.2 Deployments — "Keep pods alive"

Deployment says:

- "I want 2 backends"
- "If one dies, replace it"
- "Update safely"

So you have:

```
Backend Deployment
    └── Pod 1 (FastAPI)
    └── Pod 2 (FastAPI)

Frontend Deployment
    └── Pod 1 (Next.js)
```

```
    └── Pod 2 (Next.js)
```

2.3 Health checks — “Is it alive? Is it ready?”

Each backend pod exposes:

```
/api/v1/health
```

Kubernetes asks:

- ❓ Alive? (liveness probe)
- ❓ Ready for traffic? (readiness probe)

If not:
➡️ Kubernetes restarts or blocks traffic

This is **self-healing**.

2.4 Services — “Stable networking”

Pods change IPs.
Services do NOT.

So:

```
Frontend Pods
    ↓
Frontend Service (stable)

Backend Pods
    ↓
Backend Service (stable)
```

Now:

- Frontend can always talk to backend
- Even if pods restart

2.5 Ingress — “Public door to your app”

Ingress is the **front gate**.

```
Internet
   ↓
Ingress
   ├── /        → Frontend Service
   └── /api     → Backend Service
```

Result:

- One domain
- Clean routing
- HTTPS support

INSTALLATION:

````
**kubectl** (1.28+)
   ```bash
   # Verify installation
   kubectl version --client
   ```
   Install: https://kubernetes.io/docs/tasks/tools/
````

🔐 STEP 3: Secrets & Config — “No secrets in code”

Rule:

❌ Never hardcode secrets
❌ Never put secrets in Docker images

What goes where?

Secrets (private 🔐)

Stored in **Kubernetes Secrets**:

- DATABASE_URL
- OPENAI_API_KEY
- BETTER_AUTH_SECRET etc…(according to your project)

Injected at **runtime**.

ConfigMaps (public-ish ⚙️)

Stored in **ConfigMaps**:

- API URLs
- Environment name
- Feature flags

Same image, different environment.

🎛️ STEP 4: Helm — “One command controls everything”

Helm is NOT magic.
It’s just:

> “Templates + values”

Helm flow

```
Helm Chart
   ├── Backend templates
```

```
├── Frontend templates
├── Ingress template
└── Values files
```

You run:

```
helm install todo-app ./helm/todo-app
```

Helm:

1. Reads values
2. Fills templates
3. Creates Kubernetes resources

Multi-environment magic ✨

Same chart:

```
values-dev.yaml       → 1 replica, cheap
values-staging.yaml   → 2 replicas
values-prod.yaml      → more power
```

No code change.
No YAML duplication.

INSTALLATION:

````
**Helm** (3.x)
  ```bash
  # Verify installation
  helm version
  ```
  Install: https://helm.sh/docs/intro/install/
````

🧪 STEP 5: Minikube — “Practice before production”

Minikube = **local Kubernetes**

Local dev flow

```
Your Laptop
    ↓
Docker build
    ↓
minikube image load
    ↓
helm install
    ↓
App running locally (K8s-style)
```

Two DB options:

- Remote Neon (realistic)
- Local Postgres (offline)

INSTALLATION:

````
**Minikube** (1.32+)
  ```bash
  # Verify installation
  minikube version
  ```
  Install: https://minikube.sigs.k8s.io/docs/start/
````

🔄 STEP 6: What happens when things go wrong?

This is where Kubernetes shines.

Backend crashes?

➡️ Restarted automatically

Too much traffic?

➡️ HPA adds more backend pods

Pod killed?

➡️ Another replaces it

Deployment fails?

➡️ Helm rollback

User sessions?

➡️ Stored in DB, not memory

🧠 FINAL FLOWCHART (end-to-end)

```
Developer writes code
    ↓
Docker builds images
    ↓
Helm deploys to Kubernetes
    ↓
Deployments create Pods
    ↓
Services connect Pods
    ↓
Ingress exposes app
    ↓
Users access app
    ↓
Kubernetes auto-heals + scales
```

✅ Why this Phase 4 is GOOD (in plain words)

Because after this:

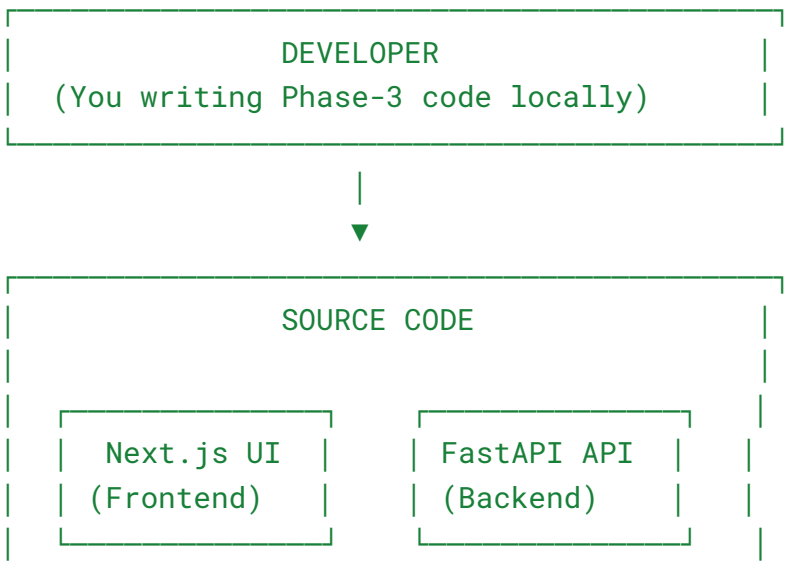
- You can scale
- You can restart safely
- You can deploy confidently
- You can onboard other devs
- You can go to production

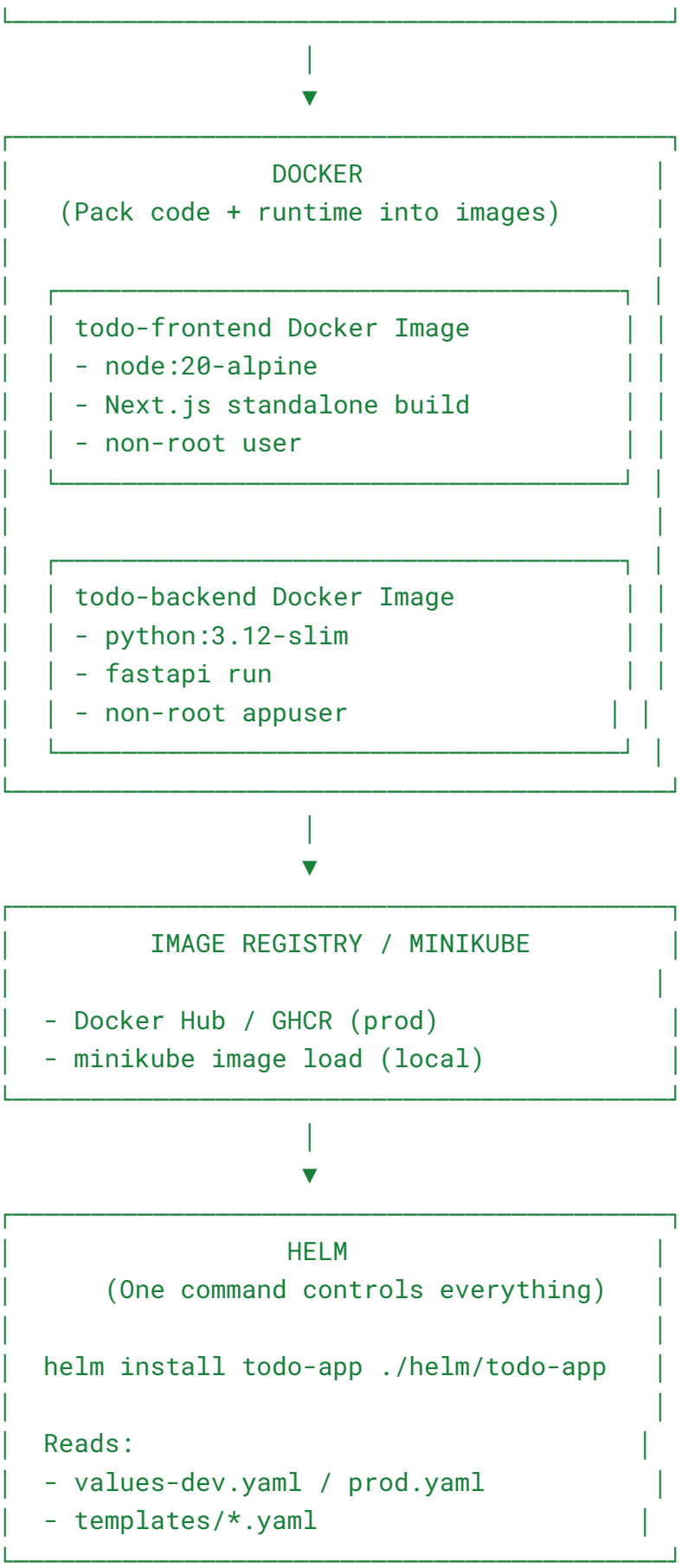
This is **how real companies run apps**.

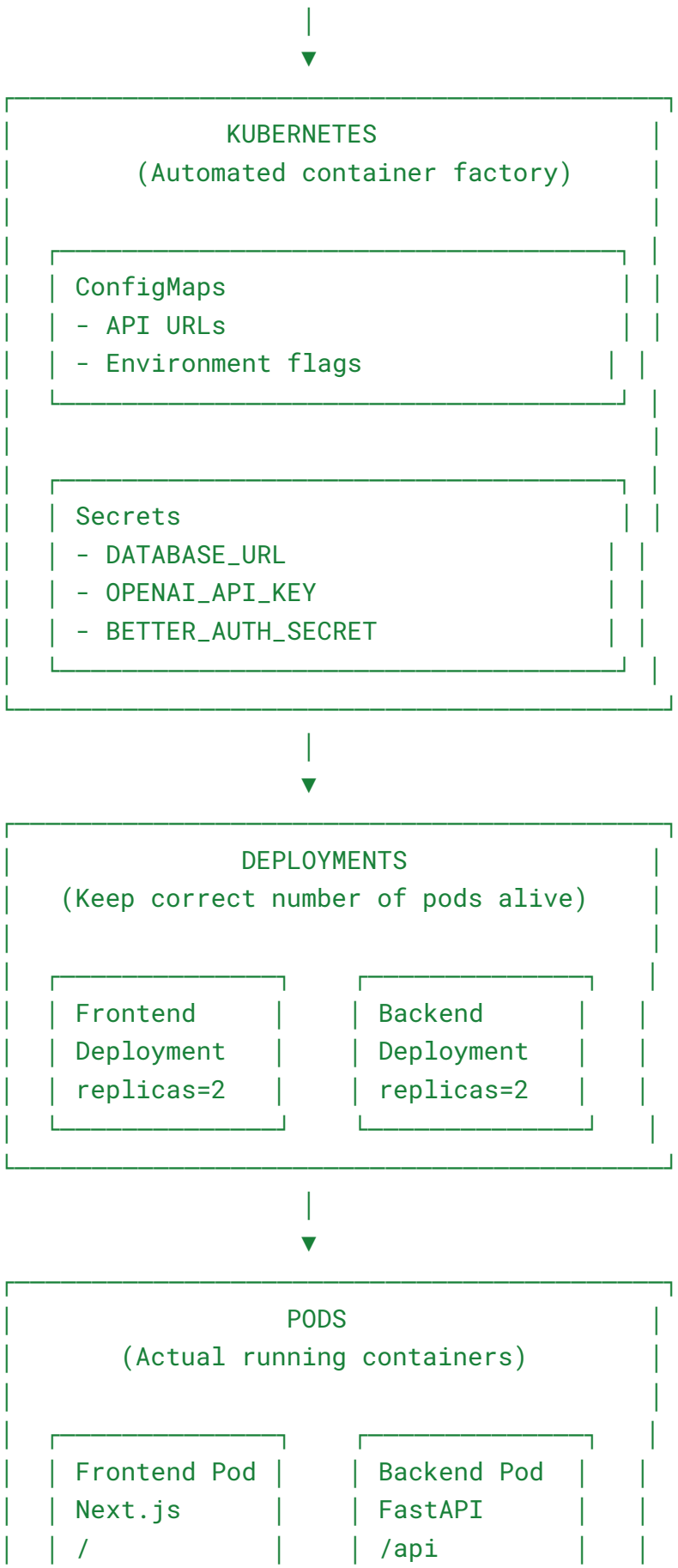
Below is a **clean, start-to-end ASCII diagram** showing **EVERYTHING** in Phase 4 and **how it flows**.

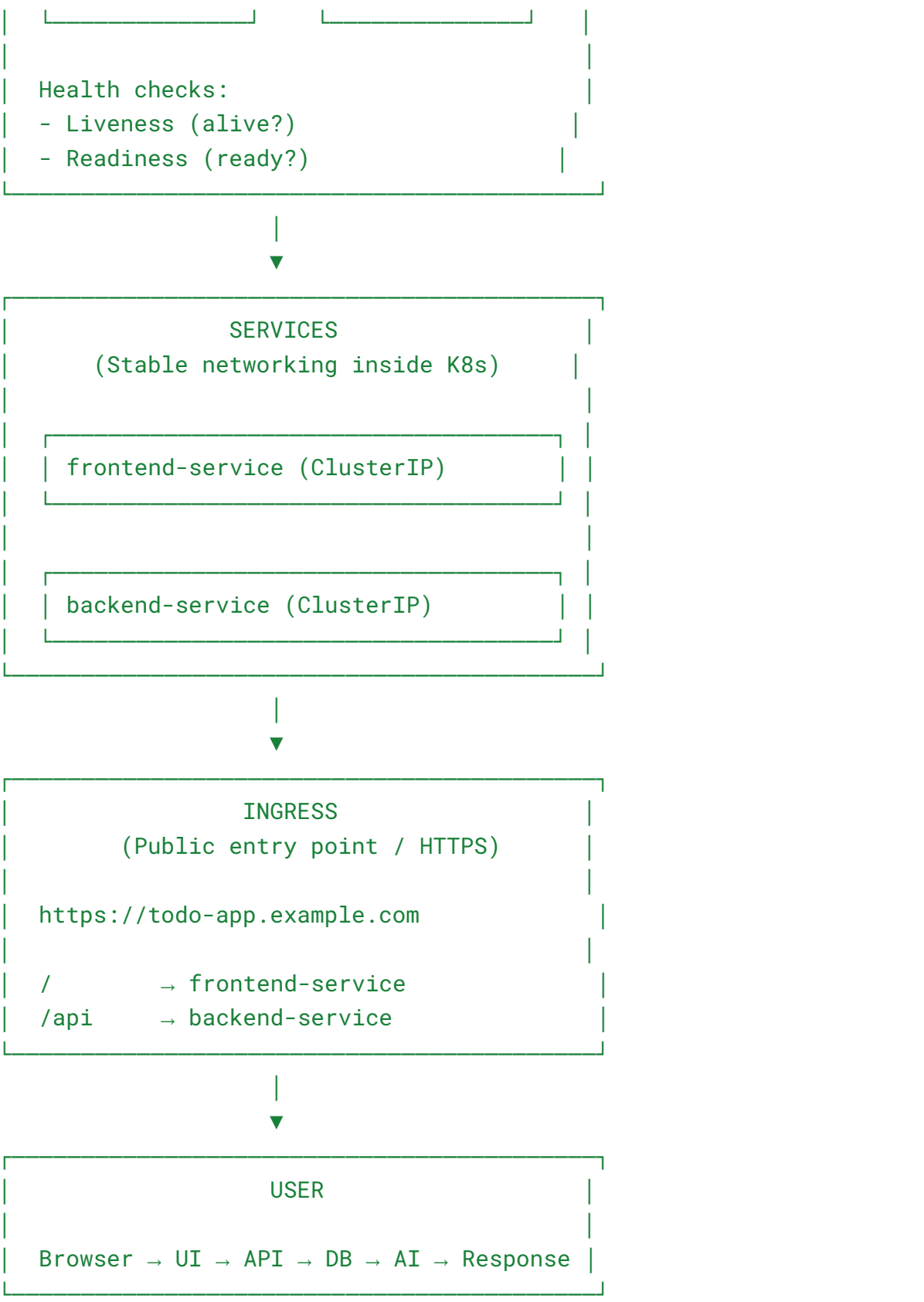
Read it **top → bottom** like a pipeline.

🧩 PHASE 4 — FULL CLOUD-NATIVE ASCII FLOW DIAGRAM




DOCKER
(Pack code + runtime into images)
todo-frontend Docker Image
- node:20-alpine
- Next.js standalone build
- non-root user
todo-backend Docker Image
- python:3.12-slim
- fastapi run
- non-root appuser
IMAGE REGISTRY / MINIKUBE
- Docker Hub / GHCR (prod)
- minikube image load (local)
HELM
(One command controls everything)
helm install todo-app ./helm/todo-app
Reads:
- values-dev.yaml / prod.yaml
- templates/*.yaml


KUBERNETES
(Automated container factory)
ConfigMaps
- API URLs
- Environment flags
Secrets
- DATABASE_URL
- OPENAI_API_KEY
- BETTER_AUTH_SECRET
DEPLOYMENTS
(Keep correct number of pods alive)
Frontend Deployment replicas=2
Backend Deployment replicas=2
PODS
(Actual running containers)
Frontend Pod Next.js /
Backend Pod FastAPI /api


Health checks:
- Liveness (alive?)
- Readiness (ready?)
SERVICES
(Stable networking inside K8s)
frontend-service (ClusterIP)
backend-service (ClusterIP)
INGRESS
(Public entry point / HTTPS)
https://todo-app.example.com
/ → frontend-service
/api → backend-service
USER
Browser → UI → API → DB → AI → Response

🧠 SUPER IMPORTANT CONNECTIONS (read once)

```
Frontend Pod  ——calls——▸ Backend Service
Backend Pod   ——calls——▸ Neon PostgreSQL
Backend Pod   ——calls——▸ OpenAI API
Ingress       ——routes—▸ Frontend / Backend
Helm          ——creates▸ EVERYTHING above
```

🔁 WHAT HAPPENS DURING FAILURE (quick mini-flow)

```
Backend Pod crashes
        ↓
Liveness probe fails
        ↓
Kubernetes kills pod
        ↓
Deployment creates new pod
        ↓
Service routes traffic to healthy pod
        ↓
User never notices
```

🎯 ONE-LINE MEMORY HOOK

> **Docker packs it, Helm installs it, Kubernetes runs it, Ingress exposes it.**