Understood. Below is a clean, **high-signal, very detailed** technical write-up documenting exactly what you built, how it worked, how we debugged, and what the root cause was. I am writing it like internal engineering notes rather than tutorial fluff.

Kubernetes + Minikube Flask + MongoDB Full Debug + Deployment Documentation

1. Goal / Architecture

You built a local Kubernetes deployment using Minikube with the following components:

- **Flask App**
  - Exposed externally via **NodePort**
  - Connects internally to MongoDB using **ClusterIP service**
- **MongoDB**
  - Runs as a **Deployment**
  - Has a **PersistentVolume (PV)** and **PersistentVolumeClaim (PVC)** for data durability
  - Auth enabled via env variables
- **Storage**
  - PV: hostPath
  - PVC: bound to PV
- **Networking**
  - Internal: service name mongodb-service
  - External: NodePort on Minikube

Architecture Diagram:

Browser → NodePort Service → Flask Pods → MongoDB Service → MongoDB Pod → PV/PVC

---

2. Files / Manifests You Used

You created the following YAMLs:

(a) Flask Deployment

- 3 replicas
- readiness & liveness probes
- env vars for Mongo credentials

(b) Flask Service (NodePort)

Used to expose Flask publicly:

type: NodePort

port: 5000

nodePort: 30000

(c) MongoDB Deployment

- Uses containerPort: 27017
- Uses MONGO_INITDB_ROOT_USERNAME and MONGO_INITDB_ROOT_PASSWORD
- Mounts volume /data/db

(d) MongoDB Service (ClusterIP)

Internal service, DNS name becomes:

mongodb-service.default.svc.cluster.local

(e) PersistentVolume + PersistentVolumeClaim

PV used hostPath:

hostPath: mongo/data

PVC requested 2Gi storage.

---

3. Minikube Execution Workflow (Correct Order)

This is the correct minimal workflow you followed:

Step 1: Start Minikube

From project root:

minikube start

Step 2: Configure Docker to Build Images Inside Minikube

Critical step for local images to be visible to Kubernetes:

Windows (CMD):

FOR /f "tokens=*" %i IN ('minikube -p minikube docker-env') DO %i

This sets environment variables:

DOCKER_HOST

DOCKER_TLS_VERIFY

MINIKUBE_ACTIVE_DOCKERD

Step 3: Build Flask Image

From project root:

docker build -t flask-app .

This ensures that the Kubernetes cluster can pull flask-app locally (no registry push needed).

Step 4: Apply Kubernetes Manifests

Move into manifests folder:

cd Manifests

kubectl apply -f .

This created:

- Deployments
- Services
- PV & PVC

Step 5: Verify Resources

Commands used:

kubectl get pods

kubectl get svc

kubectl get pv

kubectl get pvc

Step 6: Access Flask

Option A:

minikube service flask-service --url

Option B (manual):

minikube ip

kubectl get svc flask-service

Then browse:

http://<minikube-ip>:30000

---

4. Debugging Session Recap (Very Detailed)

Issue #1: CrashLoopBackoff on MongoDB

Initial logs showed:

DBPathInUse: Another mongod instance is already running

Meaning:

- Old MongoDB pod **already owned /data/db**
- Your new MongoDB tried to use same volume → lock conflict

Root Cause: Old Stateful workloads still using the PV/PVC from 309 days ago.

Action Taken:

1. Examined running pods:

kubectl get pods

Saw old deployments: flask-todo, mongodb, etc.

2. Checked PVC:

kubectl get pvc

Saw mongo-pvc in Terminating state bound to old PV.

3. Deleted old RC and Deployments:

kubectl delete deployment flask-todo ...

kubectl delete rc todo-rc

4. Deleted legacy PV/PVC:

kubectl delete pvc mongo-pvc

kubectl delete pv <pv-name>

5. Re-applied new PV/PVC

kubectl apply -f pvc.yml

kubectl apply -f pv.yml

Result:

- MongoDB could now start cleanly
- Lock file issues disappeared

Conclusion of Debug #1

Mongo **never had authentication or image issues**.
The root cause was **old pods occupying the same PVC path**, not credentials.

Issue #2: Flask CrashLoopBackoff

Initially, Flask pod crashed because Mongo pod wasn't ready.

Logs showed socket connection refused.

Sequence:

- Flask started → tried Mongo connect → Mongo not up → container crash
- Kubernetes restarts container → eventually all stabilize

After fixing Mongo:

kubectl get pods

Showed all Flask pods READY = 1/1

---

5. Networking Resolution & Verification

Internal resolution:

Flask env variables:

MONGO_HOST=mongodb-service

MONGO_PORT=27017

Inside cluster:

mongodb-service.default.svc.cluster.local

External resolution:

minikube ip → 192.168.49.2

NodePort → 30000

Final browser URL:

http://192.168.49.2:30000

or via helper:

minikube service flask-service --url

This launched browser → Flask served HTML → Confirmed DB ops work.

---

6. Authentication Clarification

On Docker Compose:

- Mongo auth succeeded

On Kubernetes:

- Auth also succeeded once Mongo was booted correctly
- No change was actually required to authentication logic
- The real blockers were stale PVC + stale pods, not credentials

You correctly concluded here:

"Most of my issue wasn't due to any authentication error. It was mainly because my earlier pods were running and they were messing up."

That is technically accurate.

---

7. Key Kubernetes Concepts You Proved

You successfully demonstrated:

Concept	Verified By
Cluster internal DNS	Flask → mongodb-service
Service discovery	selector + endpoints
Pod restart policy	CrashLoopBackoff recovery
Persistent storage	PVC + PV binding
Docker build inside cluster	docker-env env switch
External access	NodePort via minikube ip
Debugging logs	kubectl logs
Resource cleanup	deleting old RC/PVC/PV

This is basically a full production-grade workflow minus secrets.

---

8. Lessons Learned / Retrospective

1. **Minikube still retains old workloads** from previous runs unless explicitly deleted
2. **PVCs survive pod deletion**, so stale locks persist
3. **DB containers fail hard** on data directory locks
4. **NodePort vs ClusterIP is contextual**
5. **CrashLoopBackOff is often readiness not auth**
6. **Local Docker images require docker-env** to be recognized by Minikube

---

9. Final Working State

At the end:

kubectl get pods

showed:

flask-app-deployment XXX   1/1   Running

mongodb-deployment XXX   1/1   Running

and:

minikube service flask-service --url

opened the app successfully.

---

10. Optional Improvements (Future)

Not necessary, but recommended for production:

- Use StatefulSet for MongoDB
- Use Secrets instead of raw env vars
- Use ConfigMap for non-secret configs
- Use LoadBalancer on EKS instead of NodePort
- Use StorageClass for dynamic provisioning

If you want, I can also convert this into:

- Markdown doc
- PDF
- Notion wiki page
- GitHub README
- Technical blog post format
- Interview case study format

Just reply:

markdown, pdf, notion, readme, blog, or interview.

Yes, that is the correct high-level flow, and your summary is almost perfect. I'll restate it cleanly with the precise ordering and reasoning so there's zero ambiguity:

Correct End-to-End Flow (Clean Summary)

STEP 1 — Author Kubernetes YAML Manifests

You create Kubernetes manifests for:

1. **MongoDB Deployment**
2. **MongoDB Service (ClusterIP)**
3. **Flask Deployment**
4. **Flask Service (NodePort for external access)**
5. **PersistentVolume (PV)**
6. **PersistentVolumeClaim (PVC)**

At this point you have **desired state definitions**, not running resources.

STEP 2 — Start Kubernetes Cluster (Minikube)

minikube start

This boots a single-node Kubernetes cluster locally.

STEP 3 — Build Application Docker Image INSIDE Minikube Docker Daemon

Reason:

- Kubernetes pulls images from registries
- Your image is local-only
- So Minikube must “see” it

Switch Docker context:

Windows CMD:

FOR /f "tokens=*" %i IN ('minikube -p minikube docker-env') DO %i

Then build:

docker build -t flask-app .

The crucial point:

The name flask-app must match the image name in the Deployment YAML, otherwise Kubernetes won’t find it.

This eliminates the need for Docker Hub.

STEP 4 — Apply Kubernetes Manifests

Go to your Manifests folder:

kubectl apply -f .

Kubernetes now creates:

- Deployments (Mongo + Flask)
- Services (Mongo + Flask)
- PV + PVC

This is where pods start being scheduled.

STEP 5 — Kubernetes Internal Boot Order Happens Automatically

Behind the scenes:

1. **MongoDB pod starts**
2. Mongo mounts PVC
3. MongoDB Service (ClusterIP) gets a stable DNS like:
4. mongodb-service.default.svc.cluster.local
5. Flask pod starts
6. Flask reads env vars:
7. MONGO_HOST=mongodb-service
8. Flask connects to DB through service, not by IP
   (so scaling or rescheduling will not break)
9. Flask Service (NodePort) exposes app externally

STEP 6 — Access Application Externally

Option A (easiest):

minikube service flask-service --url

Option B (manual):

minikube ip

kubectl get svc flask-service

Open:

http://<minikube-ip>:<nodePort>

So Your Final Summary is Correct

Your exact statement reformulated:

"We write our deployments/services for both Flask and Mongo, then we define PV/PVC for storage, then we start Minikube, build the Flask Docker image inside Minikube, apply all manifests, and Kubernetes brings everything up. After that, application runs."

Yes.

One Slight Technical Clarification

When you said:

"so my deployment YAML can find it"

More precisely:

- Kubernetes doesn’t directly read from your Docker build command
- Kubernetes asks **container runtime** to pull the image
- Since you built the image inside Minikube's Docker daemon, Kubernetes finds it there

So the real meaning is:

The deployment pulls the image because it exists in the Minikube Docker daemon and matches the name in the Deployment YAML.

Final Flow in One Sentence

Define manifests → Build image inside Minikube → Apply manifests → Kubernetes creates pods/services → Flask connects to Mongo via service → App becomes externally accessible via NodePort.

That’s it.

If you want, I can now give you:

- A condensed cheat sheet, or
- A full "AWS EKS version" of the same workflow (with LoadBalancer + ECR), or
- A full CI/CD GitHub Actions pipeline for automation.

Just say: **cheatsheet**, **EKS version**, or **CI/CD**.