**Overview:**

As per my understanding and knowledge, I have creted the documentation for mentioned tasks:

**Local Environment (Docker Compose Based)**

The local setp uses Docker Compose to orchestrate multiple containers for Laravel, MySQL, Redis, and optionally MongoDB or Elasticsearch.

* **Docker Compose** manages application dependencies.
* **Laravel App** runs inside a container with mounted volumes for code.
* **MySQL container** stores local dev data.
* **Redis and MongoDB containers** support queueing and non-relational storage.
* **.env file** defines local runtime variables.

This setup is ideal for rapid iteration, local testing, and isolating services bfore deploying.

**Production Environment (Kubernetes + Helm)**

Production deployments use Helm charts to define and manage Kubernetes resources across environments.

* The chart laravel-app-chart includes:
  + Deployment manifests with autoscaling, probes, resource limits
  + ConfigMap and Secrets for environment injection
  + Persistent Volume Claim (PVC) support for logs
  + Ingress for routing external traffic

Deployment targets simulated EKS clusters, with values customized for production, staging, and development via values.yaml files.

------------------------------------------------------------------------------------------------------------------------

**CI/CD Workflow (Simulated using Jenkins)**

**Purpose of Each Stage**

1. **Test**
   * Runs phpunit on Merge Requests to main
   * Blocks merges if tests fail
2. **Build**
   * Runs after a successful merge to main
   * Builds Docker image
   * Simulates push to ECR
3. **Deploy**
   * Simulates deployment to EKS via Helm
   * Deploys to:
     + Dev (values.dev.yaml)
     + Staging (values.staging.yaml)
     + Production (values.yaml)

**Trigger Conditions and Flow:**

Merge Request → Jenkins triggers Test Stage

* If Passed

Merge to main → Jenkins triggers Build Stage

->

Simulated Docker push to ECR

->

Jenkins triggers Deploy Stage (Dev → Staging → Production)

All stages are dry-run safe and simulate interactions with EKS and ECR without requiring real AWS credentials.

------------------------------------------------------------------------------------------------------------------------

**Getting Started:**

1. Run the Project Locally via Docker Compose:

# Start containers

docker-compose up -d

# Run Laravel migrations

docker-compose exec app php artisan migrate

# Access the application

<http://localhost:9000>

2. Build Docker Image Manually:

# Build the Laravel app image

docker build -t laravel-app .

# Tag image for simulated registry

docker tag laravel-app:latest 123456789012.dkr.ecr.ap-south-1.amazonaws.com/laravel-app:latest

# Simulated pus

echo "docker push 123456789012.dkr.ecr.ap-south-1.amazonaws.com/laravel-app:latest"

3. Deploy Using Helm (Simulated):

# Development

helm upgrade --install laravel-dev ./laravel-app-chart -f values.dev.yaml

# Staging

helm upgrade --install laravel-staging ./laravel-app-chart -f values.staging.yaml

# Production

helm upgrade --install laravel-prod ./laravel-app-chart -f values.yaml

Each values.\*.yaml file defines replica count, image tag, resource limits, ingress settings, and environment-specific config.

-------------------------------------------------------------------------------------------------------------------

**Database Upgrade Guide (Self-Hosted)**

MongoDB Upgrade Guide (Self-Hosted):

1. Backup & Restore Strategy:

- We can use mongodump for logical backups and setup a duration for time to store the data.

Ex: mongodump --out /backup/mongo --authenticationDatabase admin

Restore:

- We can use mongorestore to restore the data

Ex: mongorestore /backup/mongo

2. Rolling Update Practices:

We can upgrade replica set members one at a time and ensure each member completes upgrade and rejoins before moving to next node

Also, we can use rs.status() to avoid upgrading primary first.

rs.stepDown()

sudo apt install mongodb-org=NEW\_VERSION

3. Compatibility Checks:

First, we need to check our current features using **featureCompatibilityVersion** command and we need to upgrade it to minor version first before jumping major versions.

4. Database Tuning:

a. We ned to enable the compression with wiredTiger for default engine.

b. Tune wiredTiger.cacheSizeGB for RAM usage.

db.collction.createIndex{ field: 1 })

mongotop – For chccking a audit slow queries.

**Elasticsearch Upgrade Guide (Self-Hosted):**

1. Backup & Restore Strategy:

With the help of snapshot API’s

PUT /\_snapshot/backup\_repo

POST /\_snapshot/backup\_repo/my\_snapshot

Register the repositoty using s3 or NFS volumes.

Restore:

POST /\_snapshot/backup\_repo/my\_snapshot/\_restore

2. Rolling Update Practices:

For clusters, we need to disable shard allocation before restarting nodes:

Ex: PUT /\_cluster/settings

{

"persistent": { "cluster.routing.allocation.enable": "none" }

}

Second method, we have to upgrade one node at a time

Third step, reenable allocation afterward:

Ex: "cluster.routing.allocation.enable": "all"

3. Database Tuning:

First, need to increase the heap size (need to take the heap or thread dump)

Ex: export ES\_JAVA\_OPTS="-Xms2g -Xmx2g"

Use index lifeccle management

Also, we can monitor the /\_cluster/status with the help of Promotheous or Grafana Dashboard

-------------------------------------------------------------------------------------------------------------

**PostgreSQL Tuning (Relational DB):**

1. Memory & Cache Settings:

shared\_buffers: Set to 25–40% of available system RAM — this is PostgreSQL’s internal cache.

work\_mem: Increase for queries with sorts, joins, or aggregations.

2. Connection & Parallelism:

max\_connections: use connection pooling (e.g., PgBouncer) for scalability.

3. Indexing Strategy

**MongoDB Tuning (Document DB):**

1. Storage Engine Considerations:

2. Indexing & Schema Design:

Use **compound indexes** for multi-field queries

Design schema for query patterns (denormalize when needed)

3. Monitoring:

Use mongostat, mongotop, and **MongoDB Atlas** dashboards

**Elasticsearch Tuning (Search Engine):**

JVM Heap & RAM:

Set heap size to ~50% of system RAM, but never exceeded to 32gb due to compressed pointers

For example: export ES\_JAVA\_OPTS="-Xms16g -Xmx16g"

Index Design: void many small indices — prefer fewer large ones.

Use **doc\_values** for aggregations, not for text search.

Sharding Strategy:

Match shard count to hardware:

Too many shards → wasted memory

**Optional:**

**Dockerfile Security Best Practices:**

* Use minimal base images like distroless, alpine
* Avoiding Installing Unnecessory packages
* Restricting Root privileges before running yhe app
* File permission restrictions
* Using multi-stage builds, In Dockerfile we can use As build for the base image
* Taking a hardened images for the security way
* Also, we can use Trivy app for scan the code and vulnerabilities.

**Kubernetes Deployment Without Helm (Raw YAML):**

We can deploy the applications using YML manifest files

We can create a folder like K8(staging) under that we can save all the files below

Deployment.yml

Service.yml

Ingress.yml

Configmap.yml

Secret.yml

For ex, we are deploying the application on staging environment means we can define in the Jenkins pipelinf

In the deploy stage, we can define like below

kubectl apply -f k8s/staging/deployment.yaml

kubectl apply -f k8s/staging/service.yaml

kubectl apply -f k8s/staging/ingress.yaml