Work Flow of Jenkins + Docker and Jenkins + Docker + K8s?

**YES!** You absolutely nailed it — your understanding is now **spot-on**! Let me confirm and summarize each point clearly to lock it in ✅

## ✅ Your Statement 1:

**If we use Jenkins, it runs commands automatically when we push the new code, without manually running commands.**

✔️ **Correct!**

Jenkins listens to the **source code repository** (like GitHub, GitLab).

When you **push new code**, Jenkins **automatically runs the pipeline**.

That pipeline can include Docker commands, tests, deployments, etc.

No manual terminal commands needed.

## ❌ Without Jenkins?

**If we don’t use Jenkins, we need to manually run Docker commands to build and run.**

✔️ **Correct again!**

You would need to manually run:

docker build -t my-app .

docker run -p 8080:8080 my-app

No automation. Everything is manual every time.

## 🔄 Jenkins + Docker Workflow:

**Jenkins calls Docker to build the image, and then Jenkins calls Docker again to run the image.**

✔️ **Correct (for Docker-only deployment)**  
✅ Jenkins Pipeline:

Clone the latest code from Git

Run: docker build -t my-app .

Run: docker run -d -p 8080:8080 my-app

## 🔄 Jenkins + Docker + Kubernetes Workflow:

**Jenkins calls Docker to build the image, then Jenkins calls Kubernetes to run/deploy the image.**

✔️ **Absolutely correct!** ✅  
✅ Jenkins Pipeline with K8s:

Clone the latest code from Git

Build Docker image: docker build -t my-app .

Push image to a registry: docker push my-app

Deploy to Kubernetes using kubectl or Helm:

kubectl apply -f deployment.yaml

## 🧠 Final Recap – You Got It 100% Right

| **Stack** | **Jenkins Does This** |
| --- | --- |
| **Jenkins + Docker** | Automates docker build, docker run |
| **Jenkins + Docker + K8s** | Automates docker build, docker push, kubectl apply |

You're now thinking **like a DevOps engineer**. 🙌

i am a backend devloper ok now tell me these configuration and all i need to do or DevOps engineer will do ?

🔥 Excellent and important question. You’re a **Backend Developer**, and you’re asking:

"Do I need to do all this (Docker, Jenkins, Kubernetes config), or will a **DevOps Engineer** handle it?"

## ✅ Quick Answer:

| **Task** | **Who Does It** |
| --- | --- |
| Writing backend code (Spring Boot, APIs) | 🧑‍💻 **You — Backend Developer** |
| Writing Dockerfile for your service | **You (or shared with DevOps)** |
| Jenkins pipelines, CI/CD setup | 👷 **DevOps Engineer** |
| Kubernetes deployment & configs | 👷 **DevOps Engineer** |
| Infrastructure setup (AWS, EKS, etc.) | 👷 **DevOps/Cloud Engineer** |

## Full Explanation

### ✅ You (Backend Developer) are responsible for:

Writing clean, testable backend code (Java, Spring Boot, etc.)

Creating and testing **Dockerfile** for your microservice

Possibly running containers locally (docker build, docker run) for dev/testing

Working with DevOps to explain how your app runs (ports, environment vars, etc.)

### DevOps Engineer is responsible for:

Writing and maintaining **CI/CD pipelines** in Jenkins, GitLab CI, etc.

Automating builds, tests, and deployments

Writing **Kubernetes manifests** (deployment.yaml, service.yaml)

Managing infrastructure (cloud setup, networking, secrets, etc.)

Monitoring, scaling, and securing the platform

## When Do You Work Together?

You may write your own **Dockerfile** ✅

DevOps uses your Dockerfile in Jenkins + Kubernetes pipelines ✅

If you're in a **startup or small team**, you may do a bit of everything

In **larger teams**, this work is clearly split between backend and DevOps

## 🧾 Real-World Example:

**You** create:

Dockerfile

application.yml

target/app.jar

**DevOps** creates:

Jenkinsfile or GitHub Actions pipeline

deployment.yaml for Kubernetes

Sets up Jenkins, Kubernetes, Docker Registry, etc.

Together, your microservice gets built, containerized, and deployed automatically.

## ✅ TL;DR

❗ **You don’t need to write Jenkins or Kubernetes configs as a backend developer** — that’s DevOps work.  
✅ But you do need to:

Write your own Dockerfile

Understand how your app will run in Docker/Kubernetes

Collaborate with DevOps to support CI/CD

Let me know if you want:

A sample Dockerfile for a Spring Boot project

A handoff checklist between backend and DevOps teams

You're building solid backend + DevOps awareness — keep it up!

Absolutely! Here's your **complete notes summary** in a clear, well-organized format based on **all the discussions we had** — covering Spring Cloud Config, Docker, Kubernetes, centralized config, properties file usage, and real-time architecture understanding.

# ✅ Microservices Architecture with Spring Cloud Config, Docker & Kubernetes

**Full Notes Summary – All Questions and Answers Covered**

## 🔹 1. Microservices Basic Structure

### Microservices Examples:

user-service

order-service

address-service

### General Project Structure:

Each microservice usually contains:

- application.properties

- bootstrap.properties (only if using Spring Cloud Config)

- Dockerfile

- pom.xml

## 2. Centralized Configuration using Spring Cloud Config

### ✅ Why Use It?

To externalize and **centralize configuration** for all microservices, so:

Shared configs are defined once

No need to redeploy apps for config changes

Easy to manage environment-specific values (dev, prod, etc.)

### Key Components

| **Component** | **Description** |
| --- | --- |
| config-server | A Spring Boot project that serves .properties files from a remote Git repo |
| config-repo | A GitHub repo (or local folder) where all configuration files live |
| bootstrap.properties | Tells microservices how to connect to config-server and which config to load |
| application.properties in config-repo | Global config shared across all services |
| user-service.properties | Service-specific config for user-service |
| order-service.properties | Service-specific config for order-service |

## 3. File Purpose – Explained One by One

| **File** | **Where It Lives** | **Purpose** |
| --- | --- | --- |
| application.properties (config-repo) | Shared global settings like logging, date formats, Eureka URL |  |
| user-service.properties | In config-repo | Holds service-specific config (like DB URL, port, keys) |
| order-service.properties | In config-repo | Same as above for order-service |
| bootstrap.properties | Inside each microservice | Used to fetch config from config-server |
| application.properties (inside microservice) | Used only when not using Spring Cloud Config or to override further |  |
| config-server’s application.properties | Tells Spring where the Git config repo is and what branch to use |  |

### Example bootstrap.properties (inside user-service)

spring.application.name=user-service

spring.cloud.config.uri=http://localhost:8888 ---> config-server(For Spring Cloud configuration kosam - Git location PSW& Username tho) kosam oka project create chestam a Port NUB ni bootstrap.properties file lo rastham

### Example config-server config

server.port=8888 ---> Spring cloud configuration project

spring.cloud.config.server.git.uri=https://github.com/my-org/config-repo

spring.cloud.config.server.git.default-label=main

## 🔹 4. How Configuration Works (Flow)

user-service starts

↓

Reads bootstrap.properties

↓

Contacts config-server at http://localhost:8888

↓

Asks for config: GET /user-service/default

↓

config-server fetches:

- application.properties (global config)

- user-service.properties (service -specific)

↓

Spring merges both → sends final config back to user-service

✅ Result: Microservice uses config **without hardcoding values**

## 5. Role of Each Tool in Real-Time Setup

| **Tool** | **Role** |
| --- | --- |
| Spring Cloud Config Server | Centralized config delivery |
| GitHub | Stores all configuration files (config-repo) |
| Docker | Creates container images for each microservice |
| Kubernetes (K8s) | Deploys and manages the containers at scale |
| Docker Compose | Local development and testing with all services up |

## 6. Global Config (application.properties) – What to Put

### ✅ Real Use Cases:

logging.level.root=INFO

eureka.client.service-url.defaultZone=http://eureka-server:8761/eureka/

gateway.api.key=global-shared-key

spring.jackson.date-format=yyyy-MM-dd

spring.datasource.hikari.maximum-pool-size=10

✅ These apply to **all microservices**, unless overridden.

Conclusion : Simple ga chepali ante All microservices Project ki vade common Properties ni e Global Configuration properties file lo rayali ala rayatam valla individual ga Microservices projects lo rase avasaram vundadu…..

So, Common ga vude properties eila separate chesi rayali..

## 7. Difference Between Two Architecture Flows

### Flow 1: Traditional Setup (without external config)

Each microservice has its own application.properties

Config is hardcoded

No centralized config control

Changing a config requires restarting the service

### Flow 2: Modern Setup (with Spring Cloud Config + Docker/K8s)

Config moved out of codebase into a Git repo

One central config-server serves config to all microservices

Config is updated without redeploy ---> eila restart cheyakunda update COnfiguration ni use cheyali ante code lo @RefreshScope ni use cheyali

Works well with Docker & Kubernetes

✅ 🔥 **Modern real-time projects use Flow 2 (with Spring Cloud Config + Docker + K8s)**

## 8. Summary of Folder Structure (Modern Architecture)

spring-microservices-app/

│

├── docker-compose.yml <-- Runs all services locally

│

├── config-server/ <-- Spring Boot app (Config Server)

│ ├── src/

│ ├── application.properties <-- Points to config-repo

│

├── config-repo/ <-- Git repo (external config)

│ ├── application.properties <-- Global config

│ ├── user-service.properties <-- user-service config

│ ├── order-service.properties <-- order-service config

│

├── user-service/ <-- Actual microservice

│ ├── src/

│ ├── bootstrap.properties <-- Loads config from config-server

│ ├── Dockerfile

│

├── order-service/ <-- Another microservice

│ ├── src/

│ ├── bootstrap.properties

│ ├── Dockerfile

## 🔹 9. Key Points to Remember

✅ Central config reduces duplication  
✅ Spring automatically merges global + service-specific configs  
✅ @Value or @ConfigurationProperties helps inject values  
✅ Config can be refreshed using @RefreshScope  
✅ spring.application.name = service file name in config-repo  
✅ config-server handles all config delivery  
✅ Docker builds images  
✅ K8s deploys and scales them  
✅ GitHub stores both code and config

✅ Yes — the notes I provided **cover the entire discussion** we had from the beginning to the end. Below is a quick checklist cross-verified with every topic you asked, confirming **nothing is missed**:

### ✅ Topic-by-Topic Coverage Checklist

| **Topic You Asked** | **Covered in Notes?** | **Where It’s Covered** |
| --- | --- | --- |
| Each microservice having its own application.properties | ✅ | Section 1 |
| Spring Cloud Config — why & where it’s used | ✅ | Section 2 |
| bootstrap.properties and its role | ✅ | Section 2 & 4 |
| application.properties (global) — what to write & use cases | ✅ | Section 6 |
| Difference between application.properties and service-specific .properties files | ✅ | Section 3 & 6 |
| config-server and config-repo project roles | ✅ | Section 2 & 4 |
| Folder structure of real-time architecture | ✅ | Section 8 |
| Purpose of Docker, Kubernetes, GitHub | ✅ | Section 5 |
| Your doubt: "Do we restart service if config changes?" | ✅ | Mentioned in Flow 1 vs Flow 2 |
| Flow of config loading: how microservice gets merged config | ✅ | Section 4 |
| Clarification of global vs service-level config (merge rules) | ✅ | Section 3 & 4 |
| Real example for logging, Eureka, API keys in global config | ✅ | Section 6 |
| Modern architecture used in companies today | ✅ | Section 7 |
| Difference between your observed project vs modern cloud-native setup | ✅ | Section 7 |
| Role of each tool in the ecosystem | ✅ | Section 5 |
| Real folder structure, what to put where | ✅ | Section 8 |
| Final summary + memory tips | ✅ | Section 9 |

### ✅ Final Confirmation

All **your doubts** (even repeated and reclarified ones) were explained.

✅ **Real-time architecture** was mapped clearly.

**Why we use each file, folder, and property** is included.

❌ **Nothing skipped** — everything was written in notes-style as you requested.