Docker packages applications into portable units called containers, and Kubernetes is a tool for managing and running those containers at a large scale.

What is Docker? 📦

Docker is a platform that solves the classic "it works on my machine" problem. It lets you package your application, along with all its necessary code, libraries, and settings, into a single, isolated package called a **container**.

Think of it like a standard shipping container for software. This container can be moved around and will run exactly the same way on a developer's laptop, a testing server, or in the cloud (like AWS or Azure).

- **Core Job:** To **build** and **run** individual containers.
- **Key Benefit: Portability** and **consistency**.

What is Kubernetes? 🚢

Running a few Docker containers is easy. But what about running hundreds or thousands of containers for a large application like Flipkart or Zomato across many different servers? This is where Kubernetes comes in.

Kubernetes, also known as **K8s**, is a **container orchestrator**. Its job is to automate the deployment, scaling, and management of containerized applications.

It handles complex tasks like:

- **Scheduling:** Automatically deciding which server should run a container.
- **Self-healing:** If a container crashes, Kubernetes automatically restarts it.
- **Scaling:** If traffic to your app increases, Kubernetes can automatically create more copies of your container to handle the load.
- **Load Balancing:** Distributing network traffic so that no single container gets overwhelmed.

The Relationship: How They Work Together 🤝

Docker and Kubernetes are not competitors; they work together as part of a modern application deployment pipeline.

You use **Docker** to create the container image. You then tell **Kubernetes** to run and manage that container for you in production.

In short: Docker builds the box; Kubernetes ships and manages thousands of those boxes.

Feature	Docker	Kubernetes
Primary Role	Packages an application into a single container	Runs and manages many containers across many machines
Scope	A single machine or node	A cluster of machines (nodes)
Analogy	The shipping container	The port, cranes, and logistics system

Event-Driven Architecture: Observer/subscriber patterns using Kafka/RabbitMQ
Service Management: Version control, JAR handling, backward compatibility.
Deployment Strategies: Canary, blue-green, and manual vs. automated scaling.
Deployment Management: GCP/AWS integrations, project structuring.
Common Challenges: JVM DNS caching, proxy configs, port mapping.

Docker & Kubernetes: Container orchestration, networking, Spring Boot-MySQL setup.

Internet Infrastructure: DNS, Anycast, BGP, ISP interactions.

CDNs: Global content delivery using edge servers.

Message Queues: Kafka, RabbitMQ, GCP Pub/Sub for async processing.

Messaging Models: Push vs. pull, fan-out cache refresh, message persistence.

System Robustness: Handling dependencies and failure points.

Traffic Analysis 📊

The given traffic pattern highlights a highly active, read-heavy system.

- **Total Peak QPS:** 50,000 RPS
- **Read QPS:** Business Search (20k) + Review Fetching (15k) + User Profile (5k) = **40,000 RPS**
- **Write QPS:** Image Uploads (2.5k) + New Review Submission (2.5k) = **5,000 RPS**
- **Read-to-Write Ratio:** 8:1

This traffic volume justifies dedicated microservices for each function (especially Search) and necessitates an aggressive, multi-layered caching strategy to handle the high read load.

Storage Estimations (5-Year Projection) 💾

Here are the storage calculations based on the data you provided.

1. Business Data Storage

This calculation is for the new businesses being onboarded.

- **Onboarding Rate:** 10,000 businesses/month
- **Total New Businesses (5 years):** `10,000 * 12 months * 5 years = 600,000` businesses
- **Assumption:** Each business record (metadata, address, hours, etc.) requires `2 KB` of storage.
- **Calculation:** `600,000 businesses * 2 KB/business = 1.2 GB`
- **Total Business Storage: ~1.2 GB**

2. Review Data Storage

We can estimate this based on the daily active users.

- **Daily Active Users (DAU):** 25 million
- **Assumption:** 1 in every 1,000 active users writes a review each day.
- **New Reviews per Day:** `25,000,000 / 1000 = 25,000 reviews/day`
- **Total New Reviews (5 years):** `25,000 * 365 days * 5 years = 45,625,000` reviews (approx. **46 million**)
- **Assumption:** Each review record (text, rating, user ID, business ID) requires `2 KB` of storage.
- **Calculation:** `46,000,000 reviews * 2 KB/review = 92 GB`
- **Total Review Storage: ~92 GB**

3. Image Data Storage

This is often tied to review submissions.

- **Total New Images (5 years):** Let's assume 1 photo is uploaded for every new review submitted, which is **46 million** images.

- **Assumption:** The average size of a compressed, processed image is 500 KB.
- **Calculation:** 46,000,000 images * 500 KB/image = 23,000,000 MB = 23,000 GB
- **Total Image Storage: ~23 TB**

Summary of Estimates

- **Business Data:** ~1.2 GB
- **Review Data:** ~92 GB
- **Image Data:** ~23 TB
- **Total Estimated Storage (over 5 years): ~23.1 TB**

Conclusion: Image storage is the dominant factor, overwhelmingly larger than all other metadata combined. This confirms the architectural need for a dedicated object storage solution like **Amazon S3** or **Google Cloud Storage** for images, while the structured business and review data can be managed in a scalable database like a sharded PostgreSQL or a NoSQL alternative.

Kafka Overview & Architecture 🏗️

Kafka is a distributed streaming platform built around a central **architecture** of brokers that manage data.

- **Topics:** A topic is a named category or feed to which records are published. Think of it as a table in a database (e.g., `user-clicks`, `order-updates`).
- **Partitions:** Each topic is split into one or more **partitions**. A partition is an ordered, immutable sequence of records. Splitting a topic into partitions is the primary way Kafka achieves scalability.
- **Producers:** Applications that write (publish) data to Kafka topics.
- **Consumers:** Applications that read (subscribe to and process) data from Kafka topics.

Scalability & Consumer Groups 🚀

Kafka's performance comes from its ability to process data in parallel.

- **Partition-based Processing:** Since a topic is divided into partitions, multiple consumers can read from the same topic simultaneously, with each consumer handling a different partition. This allows for massive parallelism.
- **Consumer Groups:** A set of consumers that cooperate to consume data from a topic. Kafka ensures that each partition is consumed by **exactly one consumer** within the group at any given time. This prevents messages from being processed twice and allows work to be distributed efficiently.

Reliability & High Availability 🛡️

Kafka is designed to be fault-tolerant and highly available.

- **Replication:** Each partition can be replicated across multiple brokers. One broker acts as the **leader** for that partition (handling all reads and writes), while others act as **followers**. If the leader fails, a follower is automatically promoted to be the new leader, ensuring no data is lost.
- **Raft Protocol (KRaft):** In modern versions, Kafka uses the Raft consensus protocol to manage its cluster metadata internally, replacing its historical dependency on **Apache ZooKeeper**. This simplifies deployment and improves efficiency.

Offset Management & Processing ⚙️

Kafka tracks which messages a consumer group has already read.

- **Offset:** An offset is a unique, sequential ID that Kafka gives to each message within a partition.

- **Polling & Offset Tracking:** Consumers **poll** Kafka for new messages. As messages are processed, the consumer group periodically commits the offset of the last processed message to a special internal topic called `__consumer_offsets`. If a consumer crashes and restarts, it can read the last committed offset and resume processing from where it left off.
- **Acknowledgements & Retries:** Producers can configure **acknowledgements** (`acks`) to confirm that their messages have been successfully written to the brokers. If a write fails, a producer can automatically retry.

Custom SDKs & Community Support 🤝

- **Kafka SDK:** While Kafka provides official client libraries, organizations often create custom Software Development Kits (SDKs) to abstract away complex configurations and enforce best practices, making it easier for developers to interact with Kafka.
- **Community Engagement:** Kafka has a large and active open-source community, which is a valuable resource for Q&A, peer support, and troubleshooting.