Frequently Asked Questions (FAQ)

1. Why did you choose Go for building this ETL pipeline?

Go provides high performance, native concurrency, and is well-suited for building lightweight, scalable microservices.

2. How does KRaft mode in Kafka differ from Zookeeper-based Kafka?

KRaft (Kafka Raft) eliminates the need for Zookeeper by using an internal Raft-based consensus protocol, simplifying deployment and improving performance.

3. Why is Kubernetes essential for this setup?

Kubernetes provides scalability, self-healing, rolling updates, and resource efficiency for managing distributed components like Kafka, MongoDB, and microservices.

4. What are the resource requirements for running this pipeline locally?

Ideally, a machine with **16GB RAM**, **4-core CPU**, and **Docker + Minikube** installed.

5. Is this pipeline production-ready?

The architecture is extendable to production, but for real-world use, you'd need to add logging, error handling, authentication, CI/CD, and observability layers.

6. Can I run this pipeline on a public cloud?

Yes. You can easily deploy this setup on AWS, GCP, or Azure using managed Kubernetes (like EKS/GKE/AKS) and integrate with managed Kafka and MongoDB services.

7. What are some monitoring tools recommended for this setup?

You can use **Prometheus + Grafana** for app metrics, **Kubernetes Dashboard**, and **Kiali** if you integrate service mesh like Istio.

8. What kind of data is being used in the demo?

A dummy data generator in Go simulates event streams that mimic real-world sensor data or logs for ETL processing.

9. How do I scale the ETL application dynamically?

Kubernetes Horizontal Pod Autoscaler (HPA) is used based on CPU/memory usage or custom metrics from Prometheus.

Frequently Asked Questions (FAQ)

Architecture & Design

1. **Why choose Go for building data pipelines?**
   → Go provides native concurrency, a small memory footprint, fast compilation, and excellent performance—ideal for ETL and microservices.

2. **Why use Kubernetes instead of Docker Compose for this demo?**
   → Kubernetes enables autoscaling, resilience, and production-grade orchestration; it simulates real-world deployment scenarios.

3. **Why is Kafka used in the pipeline?**
   → Kafka is ideal for decoupled, scalable, high-throughput event streaming between microservices.

4. **What is KRaft mode in Kafka?**
   → KRaft removes the dependency on ZooKeeper, simplifying Kafka deployments and improving stability and scalability.

5. **Why MongoDB for storage? Why not SQL?**
   → MongoDB provides flexible document storage, making it easier to ingest unstructured/semistructured data commonly seen in real-world pipelines.

Implementation & Tools

6. **What does the Dummy Data Generator do?**
   → It simulates realistic event streams for ETL ingestion, mimicking a production data source.

7. **Is the ETL app scalable?**
   → Yes. It's containerized and horizontally autoscaled via Kubernetes HPA (Horizontal Pod Autoscaler).

8. **How is autoscaling configured?**
   → The ETL app uses Kubernetes HPA with custom metrics exposed via Prometheus for load-based scaling.

9. **What is the role of Minikube here?**
   → Minikube simulates a local Kubernetes cluster for development and testing on a single node.

10. **How are observability and benchmarking handled?**
    → Tools like Prometheus, Grafana, and `pprof` in Go help with monitoring CPU/memory usage and application performance.

Best Practices & Performance

11. **How does Go perform under high load vs Python?**
    → Go's performance is generally superior for concurrent, network-heavy workloads due to its native goroutines and compiled nature.

12. **What are typical performance bottlenecks in Go pipelines?**
    → Blocking I/O, lack of backpressure in channels, and unoptimized database writes are common bottlenecks.

13. **Is KRaft mode ready for production use?**
 → Yes, Kafka 3.5+ makes KRaft stable and officially recommended, but it should be tested per use case.

14. **Can this architecture handle large volumes of data?**
 → Yes, with production-grade tuning like multi-broker Kafka clusters, sharded MongoDB, and service meshes.

15. **How can we make this pipeline fault-tolerant?**
 → Use retries, dead-letter queues in Kafka, stateful apps for checkpointing, and monitoring/alerting for failures.

Production & DevOps

16. **How would you CI/CD this setup?**
 → Use GitHub Actions or Jenkins for Go builds, Docker image creation, Helm charts for deployment, and K8s rollout strategies.

17. **How secure is this stack?**
 → Add TLS for Kafka, RBAC policies in K8s, image scanning, and secret management with tools like Vault.

18. **Is this cost-effective for production?**
 → Very efficient for self-hosted setups; managed services may offer better ROI for non-ops-heavy teams.

19. **What logging strategy is used here?**
 → Standard Go logging with potential for structured logging using `zap` or `logrus` piped to EFK/ELK stacks.

20. **What are the next steps to productionize this project?**
 → Add CI/CD, Helm charts, secret management, SLOs, monitoring dashboards, and performance/load testing.

Where can I access the source code and setup instructions?

GitHub Repo: https://github.com/rahulsidpatil/go-k8s-data-pipeline