1. Define the Architecture

- **Minikube Setup**: Install Minikube on your local machine and enable necessary addons.
- **Deploy a Sample Microservice**: Run a test application with fluctuating traffic (e.g., Nginx, a Python Flask API, or a load-balanced web service) (Use `hey` or `Locust` to generate fluctuating traffic).
- **Enable LoadBalancer**: Minikube doesn't support LoadBalancer services by default, so use `minikube tunnel` to simulate one.

2. Collect and Analyze Data

- **Use Prometheus & Grafana for Monitoring**:
- **Collect Metrics**:
  - Request per second (RPS) using **Prometheus and Kube-state-metrics**.
  - CPU and memory usage using **metrics-server**.
  - Store data in **InfluxDB, SQLite, or a local JSON/CSV file** instead of AWS S3.

3. Train an AI Model for Predictive Scaling

- **Use Local AI/ML Environments**:
  - Train **LSTM, XGBoost, ARIMA, or Prophet** using Jupyter Notebooks, TensorFlow, or PyTorch.
  - Store trained models locally using **Pickle (.pkl) or ONNX format**

4. Implement AI-Based Scaling in Kubernetes

- **Develop a Kubernetes Custom Controller (Operator)** to:
  - Run predictions from the trained model.
  - Scale pods dynamically by modifying the `replicas` field in a `Deployment` manifest.
  - Use **KEDA (Kubernetes Event-Driven Autoscaling)** to trigger scaling events based on AI predictions.
- **Deploy KEDA on Minikube**
- **Use Python Script to Scale Pods Using AI Predictions**

5. Deploy and Compare with Traditional Scaling

- **Set up a baseline using Kubernetes Horizontal Pod Autoscaler (HPA)**
- **Simulate Traffic**: Use `hey` or `Locust` to generate synthetic workloads.
- **Compare Metrics**:
  - Compare AI-based predictive scaling vs. HPA response time, resource utilization, and cost savings.
  - Visualize results in **Grafana** with local logs instead of AWS CloudWatch.