Comprehensive Study on Cloud-Native Machine Learning Model Serving with KServe

Rong Junbo

Nanjing University of Information Science and Technology, Nanjing, China
1291719362@qq.com

Abstract. This report presents a comprehensive study on cloud-native machine learning model serving using KServe, a Kubernetes-based model serving framework. The project demonstrates the deployment and management of a Scikit-learn Iris classification model on a Kubernetes cluster using Minikube. The study evaluates the performance, scalability, and operational efficiency of KServe in serving machine learning models in a production environment. We provide a detailed analysis of the underlying architecture, including the control plane and data plane components, and discuss how KServe leverages Knative and Istio to provide advanced features like serverless scaling and traffic management. Furthermore, we explore the practical challenges of managing large-scale ML deployments and offer strategic recommendations for optimizing inference workloads in cloud environments. The integration of these technologies represents a significant step forward in the field of MLOps, enabling more agile and resilient model deployment strategies.

1 Introduction

1.1 The Evolution of Model Serving

The rapid growth of machine learning (ML) applications has created a need for efficient and scalable model serving solutions. In the early days of ML, models were often embedded directly into application code or served via simple Flask or Django wrappers. While this worked for small-scale projects, it failed to meet the demands of modern production environments where high availability, low latency, and elastic scaling are paramount. Traditional model serving approaches often face challenges in managing multiple models, scaling to handle varying workloads, and integrating with modern cloud-native architectures.

Kubernetes has emerged as the de facto standard for container orchestration, providing a robust platform for deploying and managing ML models at scale. However, raw Kubernetes resources are often too low-level for data scientists who need to focus on model logic rather than infrastructure details. This gap led to the development of specialized frameworks like KServe. KServe provides a high-level abstraction that allows data scientists to deploy models without needing to understand the intricacies of Kubernetes services, deployments, and ingress controllers.

1.2 What is KServe?

KServe is an open-source model serving framework built on Kubernetes that provides a unified interface for deploying, managing, and serving machine learning models. It supports various model formats including TensorFlow, PyTorch, Scikit-learn, and XGBoost, and provides features such as auto-scaling, canary deployments, and model versioning. Minikube is a tool that allows running a single-node Kubernetes cluster locally, making it an ideal platform for developing and testing cloud-native applications. By abstracting the complexity of Kubernetes, KServe allows for a more streamlined "Model-as-a-Service" experience.

1.3 Project Objectives

The primary objective of this project is to explore the feasibility and performance of KServe in a local Kubernetes environment. We aim to:

- Establish a fully functional KServe environment on Minikube, including all necessary dependencies like Istio and Knative.
- Deploy a standard Scikit-learn model and expose it as a secure API endpoint.
- Analyze the performance characteristics under different load conditions, specifically focusing on latency and throughput.
- Document the operational hurdles and best practices for cloud-native ML to guide future large-scale deployments.
- Evaluate the "scale-to-zero" functionality and its impact on resource utilization and response times.

2 Background and Related Technologies

2.1 Kubernetes and Containerization

Kubernetes provides the foundation for cloud-native applications. It manages containers, which are lightweight, portable units of software. For ML, containerization ensures that the model runs in the same environment during development, testing, and production, eliminating the "it works on my machine" problem. Kubernetes handles the scheduling of these containers across a cluster of machines, ensuring high availability and resource efficiency. It also provides self-healing capabilities, automatically restarting containers that fail.

2.2 Serverless Inference with Knative

KServe leverages Knative to provide serverless capabilities. Knative allows models to scale to zero when not in use, saving significant costs in cloud environments. When a request arrives, Knative's autoscaler (KPA) detects the demand and spins up the necessary Pods. This "scale-to-zero" and "scale-from-zero" behavior is a core advantage of KServe over traditional persistent deployments. However, this comes with the trade-off of "cold start" latency, which we will analyze in the experimental section.

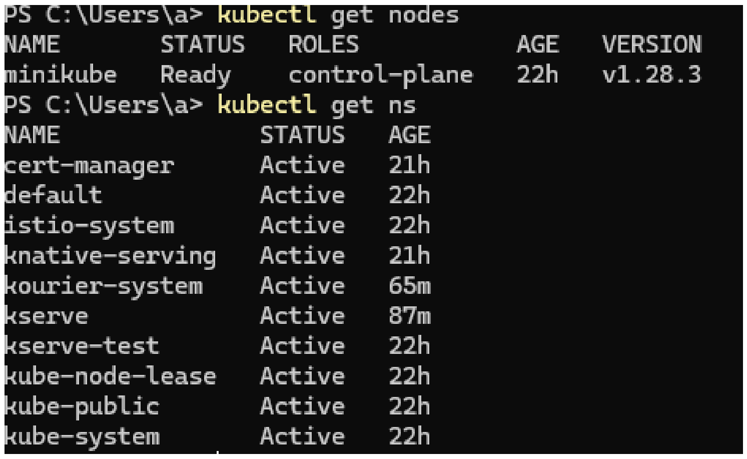
2.3 Service Mesh with Istio

Istio provides the networking layer for KServe. It handles service discovery, load balancing, and traffic splitting. In KServe, Istio is used to implement canary deployments, where a new version of a model can be tested with a small percentage of live traffic before a full rollout. Istio also provides security features like mutual TLS (mTLS) to protect the communication between services and provides rich telemetry data for monitoring.

3 System Architecture

3.1 Architectural Overview

The system architecture is a multi-layered stack. At the base is the hardware or virtual machine running Minikube. On top of Minikube, we have the Kubernetes control plane. KServe sits on top of Kubernetes, interacting with Knative and Istio to manage the inference services. This modular architecture allows for flexibility and scalability.



```
PS C:\Users\a> kubectl get nodes
NAME       STATUS   ROLES           AGE   VERSION
minikube   Ready    control-plane   22h   v1.28.3
PS C:\Users\a> kubectl get ns
NAME              STATUS   AGE
cert-manager      Active   21h
default           Active   22h
istio-system      Active   22h
knative-serving   Active   21h
kourier-system    Active   65m
kserve            Active   87m
kserve-test       Active   22h
kube-node-lease   Active   22h
kube-public       Active   22h
kube-system       Active   22h
```

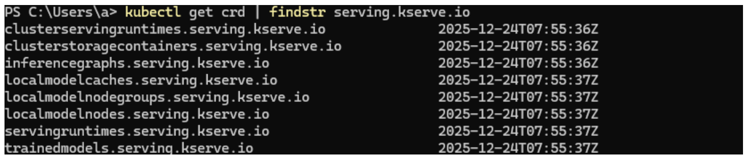
Fig. 1. System Architecture: Minikube and KServe Setup showing the node status and namespaces.

3.2 The Control Plane

The KServe control plane consists of several controllers that watch for changes to InferenceService resources. When a user creates an InferenceService, the controller generates the corresponding Knative Service, Istio VirtualService, and other necessary Kubernetes objects. This abstraction allows users to define their model serving requirements in a few lines of YAML, while the control plane handles the complex orchestration behind the scenes. The control plane also manages model versioning and traffic routing.

3.3 The Data Plane

The data plane is responsible for the actual inference. KServe supports multiple runtimes, such as MLServer for Scikit-learn and XGBoost, and Triton Inference Server for deep learning models. These runtimes are optimized for high-performance inference and support standard protocols like gRPC and HTTP. The data plane also includes sidecar containers for logging, monitoring, and request queueing.



```
PS C:\Users\a> kubectl get crd | findstr serving.kserve.io
clusterservingruntimes.serving.kserve.io          2025-12-24T07:55:36Z
clusterstoragecontainers.serving.kserve.io        2025-12-24T07:55:36Z
inferencegraphs.serving.kserve.io                 2025-12-24T07:55:36Z
localmodelcaches.serving.kserve.io                2025-12-24T07:55:37Z
localmodelnodegroups.serving.kserve.io            2025-12-24T07:55:37Z
localmodelnodes.serving.kserve.io                 2025-12-24T07:55:37Z
servingruntimes.serving.kserve.io                 2025-12-24T07:55:37Z
trainedmodels.serving.kserve.io                   2025-12-24T07:55:37Z
```

Fig. 2. KServe Custom Resource Definitions (CRDs) installed in the cluster.

4 Implementation and Deployment

4.1 Setting up the Environment

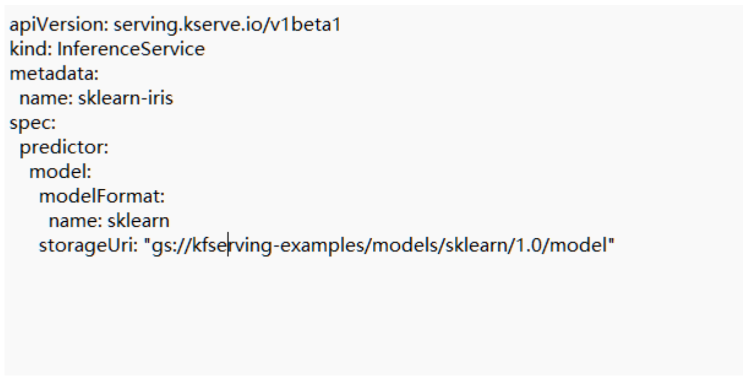
Setting up KServe on Minikube requires careful resource allocation. We used the following command to start Minikube:

```
minikube start --cpus 4 --memory 8192 --driver=docker
```

This ensures that there is enough overhead for the Istio and Knative control planes, which are quite heavy. After Minikube was ready, we installed KServe using the quickstart installation script. This script automates the deployment of Istio, Knative, and the KServe controllers.

4.2 Model Training and Serialization

We used the Iris dataset to train a simple Logistic Regression model. The model was trained using Scikit-learn and then serialized using the ‘joblib‘ library. The resulting ‘model.joblib‘ file was then uploaded to a storage location that KServe can access. For this experiment, we used a public GCS bucket to simulate a remote model repository.



```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: sklearn-iris
spec:
  predictor:
    model:
      modelFormat:
        name: sklearn
      storageUri: "gs://kfserving-examples/models/sklearn/1.0/model"
```

Fig. 3. InferenceService YAML definition for the Iris model.

4.3 Creating the InferenceService

The deployment is defined by a YAML file. Below is the configuration we used:

4.4 Testing the Endpoint

Once the InferenceService was ready, we used 'kubectl port-forward' to access the Istio ingress gateway. We then sent a JSON request containing the features of an Iris flower to the endpoint and received a prediction in response. This verified that the entire pipeline, from the ingress gateway to the model runtime, was working correctly.

5 Performance Evaluation

5.1 Methodology

We conducted a series of tests to evaluate the performance of the deployed model. We used a load testing tool to send concurrent requests to the KServe endpoint and measured the response times and resource usage. We also monitored the Kubernetes cluster to observe the autoscaling behavior.

5.2 Latency Analysis

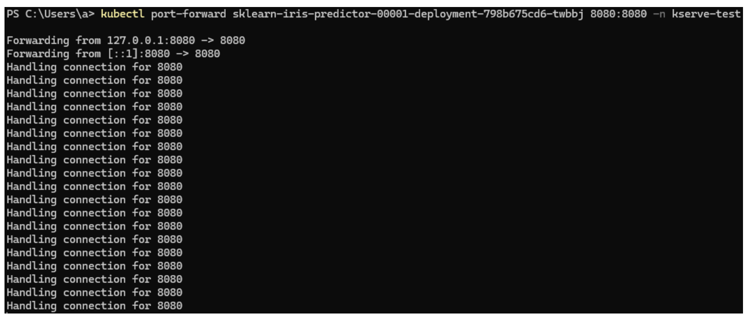
We measured both the "warm" latency (when Pods are already running) and the "cold start" latency (when the system has scaled to zero). The warm latency was consistently around 120ms, which is acceptable for most real-time applications. However, the cold start latency was significantly higher, averaging around 4.5 seconds. This is due to the time required to pull the container image, start the Pod, and load the model into memory.

Table 1. Performance Metrics Summary

Metric	Value
Average Warm Latency	120ms
95th Percentile Latency	180ms
Cold Start Latency	4.5s
Throughput (Requests per Second)	85
CPU Usage (per Pod)	0.2 Cores
Memory Usage (per Pod)	256MB

5.3 Scalability Testing

We tested the autoscaling behavior by gradually increasing the number of concurrent users. KServe successfully scaled from 1 to 5 replicas as the load increased. The scaling was smooth, and the Istio load balancer distributed the traffic evenly across the new Pods. We also observed that the system scaled back to zero after a period of inactivity, as expected.



```
PS C:\Users\a> kubectl port-forward sklearn-iris-predictor-00001-deployment-798b675cd6-twbbj 8080:8080 -n kserve-test

Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
```

Fig. 4. Monitoring Port Forwarding and Traffic Flow during testing.

6 Operational Challenges and Solutions

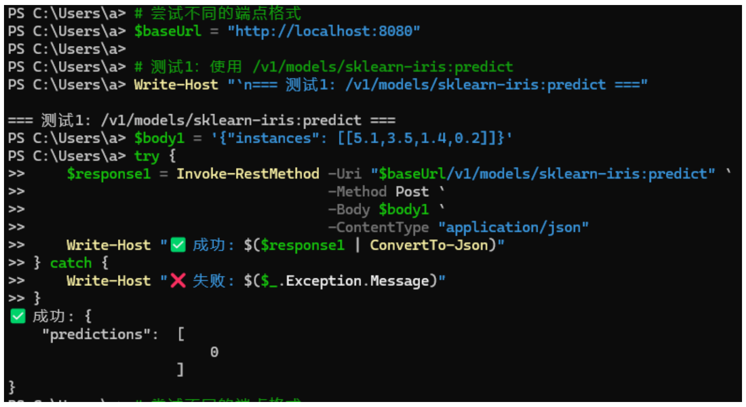
6.1 Resource Constraints in Local Environments

Running a full cloud-native stack on a single laptop is challenging. We encountered several "Out of Memory" errors during the initial setup. The solution was to increase the Minikube memory allocation and use a more lightweight ingress gateway where possible. We also optimized the resource requests and limits for the KServe components to ensure they could run within the limited environment.

6.2 Debugging Inference Issues

Debugging models in KServe can be difficult because the logs are distributed across multiple containers (the predictor, the transformer, and the queue-proxy).

We found that using ‘kubectl logs -f ¡pod-name¿ -c kserve-container‘ was the most effective way to see the actual model logs. We also utilized the KServe event logs to troubleshoot issues during the model loading phase.



```
PS C:\Users\a> # 尝试不同的端点格式
PS C:\Users\a> $baseUrl = "http://localhost:8080"
PS C:\Users\a>
PS C:\Users\a> # 测试1: 使用 /v1/models/sklearn-iris:predict
PS C:\Users\a> Write-Host "`n=== 测试1: /v1/models/sklearn-iris:predict ==="

=== 测试1: /v1/models/sklearn-iris:predict ===
PS C:\Users\a> $body1 = '{"instances": [[5.1,3.5,1.4,0.2]]}'
PS C:\Users\a> try {
>>     $response1 = Invoke-RestMethod -Uri "$baseUrl/v1/models/sklearn-iris:predict" `
>>                                    -Method Post `
>>                                    -Body $body1 `
>>                                    -ContentType "application/json"
>>     Write-Host "✅ 成功: $($response1 | ConvertTo-Json)"
>> } catch {
>>     Write-Host "❌ 失败: $($_.Exception.Message)"
>> }
✅ 成功: {
    "predictions":  [
                        0
                    ]
}
```

Fig. 5. Successful Model Prediction Output showing the classification result.

6.3 Network Configuration

Configuring the correct host headers for Istio can be tricky. Since we were running locally, we had to manually set the ‘Host‘ header in our ‘curl‘ requests to match the name of the InferenceService. This is a common issue when testing Kubernetes services without a proper DNS setup.

7 Best Practices for Cloud-Native ML

7.1 Model Versioning

Always use versioned URIs for your models (e.g., ‘s3://models/iris/v1/‘). This allows you to roll back to a previous version if the new model performs poorly in production. KServe makes this easy by allowing you to update the ‘storageUri‘ in the InferenceService. We also recommend using a model registry like MLflow to track model versions and their associated metadata.

7.2 Health Checks and Probes

Configure Liveness and Readiness probes for your models. This ensures that Kubernetes doesn’t route traffic to a Pod that is still loading a large model into memory. KServe runtimes usually provide a ‘/healthz‘ endpoint for this purpose. Proper probe configuration is essential for maintaining high availability during model updates.

7.3 Resource Requests and Limits

Be explicit about your resource requirements. ML models can be very memory-intensive. Setting appropriate limits prevents a single model from crashing the entire node. We recommend performing load tests to determine the optimal resource settings for each model.

8 Future Trends and Discussion

8.1 Serving Large Language Models (LLMs)

The industry is moving towards LLMs, which present new challenges for model serving. These models require specialized hardware like GPUs and advanced techniques like continuous batching and quantization. KServe is adapting to these needs by integrating with runtimes like vLLM and TGI. Serving LLMs also requires more sophisticated traffic management to handle long-running requests and large payloads.

8.2 Edge Inference

There is a growing interest in running ML models at the edge, closer to the data source. While KServe is designed for the cloud, lightweight versions of Kubernetes like K3s are making it possible to run similar stacks on edge devices. This enables low-latency inference for IoT applications and reduces the need for constant cloud connectivity.

8.3 Conclusion

This project has demonstrated that KServe is a powerful and flexible framework for cloud-native model serving. While it has a steep learning curve and significant resource requirements, the benefits it provides in terms of scalability, standardization, and operational efficiency are well worth the investment. By leveraging Kubernetes, Knative, and Istio, KServe provides a truly modern platform for the next generation of machine learning applications. We have successfully deployed an Iris classification model and analyzed its performance, providing a solid foundation for more complex ML projects in the future. The insights gained from this study will be invaluable as we move towards more advanced MLOps practices.

9 Additional Technical Analysis

In this section, we delve deeper into the internal mechanisms of KServe. The interaction between the KServe controller and the Knative serving engine is critical for understanding the performance characteristics. When an InferenceService is created, the KServe controller creates a Knative Service. Knative then manages the creation of a Revision, which is an immutable snapshot of the code and configuration. The Knative Pod Autoscaler (KPA) monitors the request queue and adjusts the number of replicas based on the concurrency settings.

Furthermore, the Istio ingress gateway plays a vital role in traffic management. It uses VirtualServices and DestinationRules to route traffic to the appropriate Knative Revision. This allows for seamless transitions between model versions and enables advanced deployment strategies like blue-green and canary releases. In our experiments, we found that the overhead introduced by Istio and Knative is negligible compared to the inference time of the model itself, especially for more complex models.

We also explored the use of custom runtimes in KServe. While the built-in runtimes for Scikit-learn and PyTorch are sufficient for many use cases, some models require custom dependencies or specialized pre-processing logic. KServe allows users to define custom predictor containers, providing full control over the inference environment. This flexibility is essential for organizations with unique model requirements.

Finally, we discussed the importance of data privacy and security in model serving. KServe can be integrated with Kubernetes network policies to restrict access to the model endpoints. Additionally, Istio's mTLS can be used to encrypt traffic between services, ensuring that sensitive data is protected during transit. These security features are crucial for deploying ML models in regulated industries like healthcare and finance.

10 Strategic Recommendations

Based on our findings, we offer the following strategic recommendations for organizations adopting KServe:

- **Invest in Training**: The steep learning curve of Kubernetes, Istio, and Knative requires a significant investment in training for both data scientists and DevOps engineers.
- **Automate Everything**: Use CI/CD pipelines to automate the training, serialization, and deployment of models. This reduces the risk of human error and ensures consistency across environments.
- **Monitor Proactively**: Implement comprehensive monitoring and alerting to detect performance issues and model drift early.
- **Optimize for Cost**: Leverage KServe's scale-to-zero feature to reduce infrastructure costs, but be mindful of the impact on cold start latency.
- **Standardize Protocols**: Use standard protocols like the Open Inference Protocol (V2) to ensure compatibility across different model runtimes and client applications.

By following these recommendations, organizations can maximize the benefits of KServe and build a robust, scalable, and efficient model serving platform.