# An Engineering Guide to Fine-Tuning and Deploying Large Language Models: LoRA, QLoRA, and KServe

## Section 1: Foundational Concepts: A Comparative Analysis of LoRA and QLoRA

The advent of Large Language Models (LLMs) has marked a significant milestone in artificial intelligence, yet their immense scale presents substantial challenges for practical application. This section provides a detailed analysis of two leading parameter-efficient fine-tuning techniques, LoRA and QLoRA, establishing the theoretical and practical foundations necessary for their effective implementation.

### 1.1 The Imperative for Parameter-Efficient Fine-Tuning (PEFT)

The conventional method for adapting a pre-trained model to a specific downstream task is full fine-tuning. This process involves updating all, or a significant portion, of the model's parameters using a task-specific dataset.1 For LLMs, which can comprise billions of parameters, this approach is often computationally and financially prohibitive.3

The primary challenges associated with full fine-tuning are threefold. First, it demands enormous memory resources. The GPU must hold not only the model weights but also the gradients, optimizer states, and activations for every trainable parameter, often requiring hundreds of gigabytes of VRAM. Second, for each specialized task, a full copy of the fine-tuned model must be stored, leading to significant storage costs. Third, updating all parameters on a smaller, task-specific dataset carries a high risk of overfitting, where the model memorizes the training data at the expense of generalization, and catastrophic forgetting, where the model loses its pre-trained knowledge.1

Parameter-Efficient Fine-Tuning (PEFT) has emerged as a critical paradigm to address these challenges. PEFT methods enable the adaptation of large pre-trained models by fine-tuning only a small fraction of their parameters, typically less than 1%.4 The majority of the model's original weights are "frozen" and left unchanged. This approach drastically reduces the computational and storage overhead, making it feasible to fine-tune massive models on consumer-grade or single-node enterprise hardware.6 By training fewer parameters, PEFT also mitigates the risk of overfitting and yields lightweight, portable adapters that can be easily shared and deployed for various tasks.3

### 1.2 The LoRA Mechanism: Low-Rank Adaptation in Detail

Low-Rank Adaptation (LoRA) is a prominent and widely adopted additive PEFT technique that achieves remarkable efficiency without introducing inference latency.8 The core principle of LoRA is to freeze the pre-trained model weights and inject a small number of new, trainable parameters into the model's architecture.3

Instead of directly updating the original weight matrix W of a layer, LoRA posits that the change in weights during adaptation, denoted as ΔW, has a low "intrinsic rank." This means the update can be effectively represented by the product of two much smaller matrices. LoRA decomposes this update into two low-rank matrices, A and B, such that ΔW=B⋅A. During fine-tuning, only these new matrices, A and B, are trained, while the original weight matrix W remains frozen.9 The modified forward pass for a layer becomes

h=Wx+(B⋅A)x.

This low-rank decomposition is the source of LoRA's parameter efficiency. For a weight matrix W of size d×d, the update matrices A and B would have dimensions d×r and r×d respectively, where r is the rank and r≪d. The number of trainable parameters is reduced from d2 to 2⋅d⋅r. This reduction leads to significant benefits, including:

* **Fewer Trainable Parameters:** Drastically reduces the number of parameters that require gradient computation and optimizer state storage.1
* **Faster Training:** With fewer parameters to update, the training process is accelerated compared to full fine-tuning.3
* **Reduced Memory Usage:** The smaller number of trainable parameters leads to a lower memory footprint on the GPU.3
* **Lower Overfitting Risk:** Training a smaller set of parameters helps prevent overfitting, especially on limited datasets.3

The application of LoRA is controlled via a LoraConfig object in the Hugging Face PEFT library, which is governed by several key hyperparameters 9:

* r: The rank of the update matrices. This is the most critical hyperparameter, directly controlling the number of trainable parameters. Lower ranks result in smaller adapters but may capture less task-specific information.5
* lora\_alpha: A scaling factor that modulates the magnitude of the LoRA update. The final update is scaled by rα​. This parameter is often set to twice the value of r.5
* target\_modules: A list of the specific modules within the model architecture (e.g., attention query and value projections) where the LoRA adapters will be injected.9

### 1.3 The QLoRA Extension: Introducing Quantization for Unprecedented Efficiency

QLoRA (Quantized Low-Rank Adaptation) is not a fundamentally different adaptation method but rather an extension that applies the LoRA technique to a base model whose weights have been quantized.3 This combination achieves a new level of memory efficiency, making it possible to fine-tune extremely large models on a single GPU.2

The primary innovation of QLoRA is the quantization of the large, frozen base model. The pre-trained weights, typically stored in 16-bit (BFloat16/FP16) or 32-bit (FP32) precision, are compressed into a 4-bit data type.1 This single change reduces the memory required to load the base model by a factor of 4 to 8. The LoRA adapters themselves, however, are trained in a higher precision (e.g., BFloat16) to maintain performance.17

The success of QLoRA relies on several key technical innovations introduced in its foundational paper to preserve model performance despite the aggressive quantization 16:

* **4-bit NormalFloat (NF4):** QLoRA introduces a new 4-bit data type called NormalFloat. This data type is information-theoretically optimal for data that follows a zero-centered normal distribution, a common characteristic of neural network weights. NF4 provides a more accurate representation of the weight distribution compared to standard 4-bit integer or floating-point formats, which minimizes quantization error.16
* **Double Quantization (DQ):** To further reduce the memory footprint, QLoRA employs a technique called Double Quantization. The initial quantization process requires storing "quantization constants" (e.g., scaling factors) for each block of weights. DQ treats these constants as inputs for a second round of quantization, compressing them from 32-bit to 8-bit precision. This reduces the memory overhead of the quantization metadata by approximately 0.4 bits per model parameter.16
* **Backpropagation Through a Quantized Model:** A critical aspect of QLoRA is how gradients are handled. While the base model's weights are stored in 4-bit NF4 format, they are not used for computation in this low precision. During the forward and backward passes, the frozen 4-bit weights are de-quantized on-the-fly to a higher-precision computation data type, such as BFloat16. Gradients are then calculated and backpropagated through these de-quantized weights. However, these gradients are only used to update the parameters of the high-precision LoRA adapters. The base model weights remain frozen and are never updated, staying in their 4-bit format in GPU memory throughout the process.16

### 1.4 A Quantitative Comparison: Performance, Memory, and Speed

The decision between LoRA and QLoRA represents a fundamental engineering trade-off between resource constraints and training velocity. QLoRA is not an inherently superior version of LoRA; rather, it is a powerful optimization that enables fine-tuning in environments where it would otherwise be impossible due to hardware limitations.

The primary, first-order effect of QLoRA is a dramatic reduction in GPU memory usage. Compared to standard LoRA using BFloat16 precision, QLoRA can reduce peak memory usage by up to 75%.8 This memory saving enables two significant second-order benefits. First, it allows for the fine-tuning of much larger models on the same hardware. For instance, the original QLoRA paper demonstrated fine-tuning a 65B parameter model on a single 48GB GPU, a task infeasible with standard LoRA.16 Second, for a given model size, the freed memory allows for the use of significantly larger batch sizes. On an NVIDIA A100 40G GPU, QLoRA allows for a recommended batch size of 24 for the openLLaMA-7B model, whereas standard LoRA is limited to a batch size of 2.19

However, this efficiency comes at a cost. The on-the-fly de-quantization of the base model weights during every forward and backward pass introduces computational overhead.8 This overhead is the direct cause of QLoRA's primary drawback: slower training speed. Analyses show that standard LoRA can be approximately 66% faster than QLoRA, and other experiments report a 30-50% increase in training time when using QLoRA.8

This leads to a crucial third-order implication: when memory is not the primary bottleneck, standard LoRA is the more effective choice. For smaller models on enterprise-grade hardware where the full model and training components fit comfortably in VRAM, LoRA offers faster iteration cycles and avoids any potential for accuracy degradation due to quantization errors.20 The value proposition of QLoRA becomes dominant only when memory constraints make standard LoRA impractical or impossible.

The following table provides a quantitative summary of the trade-offs between LoRA and QLoRA.

| Metric | LoRA | QLoRA | Rationale & Citations |
| --- | --- | --- | --- |
| **Peak GPU Memory** | Higher | ~75% smaller | QLoRA's primary benefit comes from quantizing the base model to 4-bit precision. 3 |
| **Training Speed** | Faster (~66%) | Slower | QLoRA incurs overhead from de-quantizing base model weights during each forward/backward pass. 8 |
| **Cost Efficiency** | Up to 40% less expensive | More expensive | If training time is the dominant cost factor, LoRA's speed makes it more cost-effective. 19 |
| **Max Batch Size** | Lower | Significantly Higher | Reduced memory footprint allows for much larger batch sizes on the same hardware. 19 |
| **Max Sequence Length** | Lower | Higher | Lower memory consumption per token allows for processing longer sequences before encountering OOM errors. 19 |
| **Accuracy** | Potentially higher | Similar | Both methods offer comparable accuracy improvements. LoRA may have a slight edge by avoiding quantization errors. 10 |

## Section 2: Practical Implementation: Merging LoRA Adapters with the Base Model

After successfully fine-tuning a model using LoRA or QLoRA, the resulting artifacts consist of the original, frozen base model and a small set of trained adapter weights. While this modular separation is highly efficient for training and storage, it is suboptimal for production inference. This section provides a practical guide on merging the adapter weights with the base model to create a single, performance-optimized artifact for deployment.

### 2.1 Rationale for Merging: From Training Efficiency to Inference Performance

During inference, if the base model and LoRA adapters are loaded separately, the computation for each forward pass involves an extra step. The model must fetch weights from both the frozen base layers and the corresponding adapter layers and combine their outputs on the fly.9 This dynamic composition, while flexible, introduces a small but measurable latency overhead to each inference request.

To eliminate this latency and streamline the deployment process, the adapter weights can be mathematically merged directly into the base model's weights. The Hugging Face PEFT library provides the merge\_and\_unload() function specifically for this purpose. This function calculates the final weight matrix W′ by adding the learned LoRA update ΔW to the original weight matrix W, such that W′=W+ΔW.9

The result of this operation is a standard, standalone model that is functionally identical to a fully fine-tuned model. It no longer contains separate adapter layers and does not require any PEFT-specific logic to run. This merged model can be deployed using any standard inference server, and because the weight addition is done offline, there is no additional latency during inference.1 This process marks a critical transition from a flexible training artifact to an optimized, immutable production artifact.

### 2.2 Code Walkthrough: The merge\_and\_unload() Process

The following Python script demonstrates the complete process of loading a base model and a LoRA adapter, merging them, and saving the final artifact using the Hugging Face transformers and peft libraries.

Step 1: Install and Import Libraries

First, ensure the necessary libraries are installed and import the required classes.

Python

# Install required libraries  
#!pip install transformers peft accelerate torch  
  
import torch  
from transformers import AutoModelForCausalLM, AutoTokenizer  
from peft import PeftModel  
  
# [22]

Step 2: Load the Base Model in High Precision

It is critical to load the base model in a high-precision data type, such as torch.bfloat16 or torch.float16. This practice prevents precision loss during the merging operation, which is essential for maintaining the fine-tuned model's accuracy.22

Python

# Define model and adapter identifiers  
base\_model\_id = "mistralai/Mistral-7B-Instruct-v0.2"  
adapter\_id = "ebinzack15/mistral-7b-finance-v1.x\_adaptor\_only" # Replace with your adapter path  
  
# Load the base model  
base\_model = AutoModelForCausalLM.from\_pretrained(  
 base\_model\_id,  
 torch\_dtype=torch.bfloat16,  
 device\_map="auto"  
)  
  
# Load the tokenizer  
tokenizer = AutoTokenizer.from\_pretrained(base\_model\_id)  
  
# [22]

Step 3: Load the PEFT Model

Next, use the PeftModel.from\_pretrained() method to apply the trained adapter weights onto the loaded base model. This creates a PeftModel object that combines the two components.

Python

# Load the LoRA adapter onto the base model  
peft\_model = PeftModel.from\_pretrained(base\_model, adapter\_id)  
  
# [14, 22, 38]

Step 4: Execute the Merge and Unload

Call the merge\_and\_unload() function on the PeftModel object. This function performs the weight addition in-place and returns the merged base model, now stripped of the adapter layers.

Python

# Merge the adapter weights and unload the adapter  
merged\_model = peft\_model.merge\_and\_unload()  
  
# [9, 22, 38]

Step 5: Save the Merged Artifacts

Finally, save the merged model and its tokenizer to a new directory using the standard save\_pretrained() method. The resulting directory contains a regular Hugging Face model that can be loaded and used for inference without the PEFT library.

Python

# Define an output directory for the merged model  
output\_dir = "merged\_mistral\_finance\_model"  
  
# Save the merged model and tokenizer  
merged\_model.save\_pretrained(output\_dir)  
tokenizer.save\_pretrained(output\_dir)  
  
# Optional: Push to Hugging Face Hub  
# merged\_model.push\_to\_hub("your-hf-username/merged-mistral-finance-model")  
# tokenizer.push\_to\_hub("your-hf-username/merged-mistral-finance-model")  
  
# [22, 38]

### 2.3 Best Practices and Nuances

While the merging process is straightforward, several nuances must be considered to ensure optimal results and avoid common pitfalls.

* **Irreversibility:** The merge\_and\_unload() operation is destructive to the PeftModel object. Once the adapters are merged and unloaded, they cannot be easily detached or managed separately. It is advisable to treat the merged model as a final, immutable artifact for deployment.9
* **Precision is Paramount:** The importance of performing the merge in high precision cannot be overstated. Loading the base model in bfloat16 or float32 ensures that the addition of the adapter weights is performed with maximum fidelity, preserving the nuances learned during fine-tuning.22
* **Quantization Workflow:** A common source of error is improper handling of quantization. Attempting to merge high-precision adapter weights directly into a 4-bit or 8-bit quantized base model can degrade performance, as it may involve naive quantization of the adapter or de-quantization of the base model, both of which introduce errors.23 The recommended workflow is to:
  1. Load the base model in high precision (e.g., bfloat16).
  2. Apply and merge the adapter weights in high precision.
  3. If a quantized model is required for deployment, perform post-training quantization on the final, merged model as a separate, controlled step. This decouples the two sources of potential error and preserves the integrity of the merged model.22
* **Advanced Multi-Adapter Merging:** While merge\_and\_unload() is ideal for a single adapter, more advanced techniques exist for merging multiple adapters trained on different tasks. Methods like TIES (TrIm, Elect, and Merge) and DARE (Drop And REscale) use sophisticated algorithms involving sparsification, sign-based consensus, and random pruning to resolve interference between different task vectors, enabling the creation of powerful multi-task models from specialized adapters.24

## Section 3: Deployment Strategies on KServe: A Technical Deep Dive

KServe is a highly scalable, standards-based model serving platform built on Kubernetes. It provides a robust solution for deploying LLMs in production environments by abstracting away the complexities of networking, autoscaling, and server configuration.26 KServe utilizes a primary Custom Resource Definition (CRD) called

InferenceService to declaratively manage model deployments.27 It has native support for serving Hugging Face models, making it an excellent choice for deploying the merged model artifacts created in the previous section.26

This section presents two distinct, production-grade strategies for deploying a merged LLM on KServe: using a self-managed Persistent Volume Claim (PVC) and pulling the model directly from the Hugging Face Hub.

### 3.2 Deployment via Persistent Volume Claim (PVC): The Controlled Environment Approach

This strategy involves storing the model artifacts on a Kubernetes Persistent Volume and configuring the InferenceService to load the model from this volume. This approach is ideal for organizations with strict security requirements, air-gapped environments, or the need for a clear audit trail of model provenance, as it avoids dependencies on external services at runtime.

Step 1: Create the Persistent Volume Claim (PVC)

A PVC is a request for storage within the Kubernetes cluster. The following YAML manifest defines a PVC named my-llm-models-pvc with a request for 50 GiB of storage, which is sufficient for a merged 7B parameter model. The ReadWriteMany access mode allows the volume to be mounted by multiple pods simultaneously, which can be useful for populating the volume and for serving runtimes that might scale to multiple replicas.30

YAML

# pvc-definition.yaml  
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
 name: "my-llm-models-pvc"  
spec:  
 accessModes:  
 - ReadWriteMany  
 resources:  
 requests:  
 storage: 50Gi

Apply this manifest using kubectl apply -f pvc-definition.yaml.

Step 2: Transfer Model Artifacts to the PVC

To get the model files onto the PVC, a common method is to create a temporary "helper" pod that mounts the volume. Once this pod is running, the model files can be copied into it from a local machine.

First, create a pod that mounts the PVC:

YAML

# helper-pod.yaml  
apiVersion: v1  
kind: Pod  
metadata:  
 name: "pvc-access-pod"  
spec:  
 containers:  
 - name: main  
 image: ubuntu  
 command: ["/bin/sh", "-ec", "sleep 3600"]  
 volumeMounts:  
 - name: "my-pvc-storage"  
 mountPath: "/mnt/models"  
 volumes:  
 - name: "my-pvc-storage"  
 persistentVolumeClaim:  
 claimName: "my-llm-models-pvc"

Apply this with kubectl apply -f helper-pod.yaml. After the pod is running, copy the merged model directory into it:

Bash

kubectl cp./merged\_mistral\_finance\_model pvc-access-pod:/mnt/models/

This command copies the local merged\_mistral\_finance\_model directory to the /mnt/models path inside the pod, which corresponds to the root of the PVC.30

Step 3: Define and Deploy the InferenceService

The final step is to create the InferenceService manifest. The critical configuration is the storageUri field, which uses the pvc:// scheme to point KServe to the model's location on the persistent volume.32

YAML

# isvc-pvc.yaml  
apiVersion: "serving.kserve.io/v1beta1"  
kind: "InferenceService"  
metadata:  
 name: "merged-model-from-pvc"  
spec:  
 predictor:  
 model:  
 modelFormat:  
 name: huggingface  
 storageUri: "pvc://my-llm-models-pvc/merged\_mistral\_finance\_model"  
 resources:  
 requests:  
 cpu: "4"  
 memory: "16Gi"  
 limits:  
 cpu: "4"  
 memory: "16Gi"

Deploy the service with kubectl apply -f isvc-pvc.yaml. KServe's controller will now provision the necessary resources, and the model server will load the model from the specified PVC path.

### 3.3 Deployment from the Hugging Face Hub: The Agile Approach

This strategy leverages KServe's built-in runtime support for pulling models directly from the Hugging Face Hub. The model server automatically downloads the specified model repository upon pod startup. This method is exceptionally well-suited for rapid development, iteration, and continuous deployment workflows where models are managed and versioned on the Hub.

Step 1 (Optional): Create an Authentication Secret

If the model is hosted in a private Hugging Face repository, KServe requires an access token to authenticate. This token should be stored securely as a Kubernetes Secret.

YAML

# hf-secret.yaml  
apiVersion: v1  
kind: Secret  
metadata:  
 name: hf-secret  
type: Opaque  
stringData:  
 hf\_api\_token: "YOUR\_HUGGING\_FACE\_READ\_TOKEN\_HERE"

Create the secret using kubectl apply -f hf-secret.yaml.34

Step 2: Define and Deploy the InferenceService

The InferenceService manifest for this approach uses the hf:// scheme in the storageUri field, followed by the model's repository ID on the Hub.29 To use the authentication token, the secret is mounted as an environment variable (

HF\_TOKEN) in the predictor's container. The KServe Hugging Face runtime is designed to automatically detect and use this environment variable for authentication.34

YAML

# isvc-hub.yaml  
apiVersion: "serving.kserve.io/v1beta1"  
kind: "InferenceService"  
metadata:  
 name: "merged-model-from-hub"  
spec:  
 predictor:  
 model:  
 modelFormat:  
 name: huggingface  
 storageUri: "hf://your-hf-username/merged-mistral-finance-model"  
 resources:  
 requests:  
 cpu: "4"  
 memory: "16Gi"  
 limits:  
 cpu: "4"  
 memory: "16Gi"  
 env:  
 - name: HF\_TOKEN  
 valueFrom:  
 secretKeyRef:  
 name: hf-secret  
 key: hf\_api\_token

Deploy this service with kubectl apply -f isvc-hub.yaml. Upon startup, the KServe pod will use the provided token to download the model from the private repository and load it into the server.

## Section 4: Strategic Recommendations: Selecting the Optimal Fine-Tuning and Deployment Pathway

The preceding sections have provided a deep technical dive into the mechanisms of LoRA/QLoRA and the practicalities of KServe deployment. This final section synthesizes this information into a strategic framework to guide practitioners in selecting the optimal end-to-end workflow based on their specific constraints, goals, and operational environment.

### 4.1 Decision Framework for Fine-Tuning: LoRA vs. QLoRA

The selection between LoRA and QLoRA is fundamentally dictated by the primary resource constraint: available GPU memory or available time for experimentation and training.

The decision process can be simplified into a clear sequence. First, identify the primary bottleneck for the fine-tuning task. Is it the inability to load the base model and its training components into VRAM, or is it the need to iterate through experiments as rapidly as possible?

If the bottleneck is VRAM, QLoRA is the necessary and superior choice. This scenario arises when using consumer-grade GPUs (e.g., NVIDIA RTX series) or when attempting to fine-tune very large models (e.g., 33B parameters or more) on enterprise hardware that still has memory limitations.16 In these cases, QLoRA's 4-bit quantization is the enabling technology that makes the task feasible.

Conversely, if sufficient VRAM is available—for example, when fine-tuning a 7B model on an 80GB NVIDIA A100 GPU—then standard LoRA is the more effective option. With memory constraints removed, LoRA's primary advantage is its training speed, which can be over 60% faster than QLoRA. This accelerates the experimental cycle and reduces computational costs. Furthermore, it completely avoids any risk of performance degradation, however small, that might be introduced by the quantization process.19

| Scenario | Recommended Method | Rationale |
| --- | --- | --- |
| **VRAM is the primary constraint** (e.g., consumer GPUs, very large models) | **QLoRA** | Enables fine-tuning that would otherwise be impossible. Allows for larger batch sizes to improve training stability. 16 |
| **Training speed is the primary constraint** (e.g., enterprise GPUs, smaller models) | **LoRA** | Significantly faster training cycles, leading to quicker iteration and lower computational cost. Avoids any potential for quantization-induced accuracy loss. 19 |

### 4.2 Decision Framework for KServe Deployment: PVC vs. Hugging Face Hub

The choice of deployment strategy on KServe is a trade-off between operational control and development velocity, often influenced by an organization's security posture and MLOps maturity.

The decision hinges on several key factors. First is the network environment. In an air-gapped or network-restricted setting, deploying from the Hugging Face Hub is not an option, making the PVC-based approach mandatory.30

Second is the requirement for security and governance. The PVC method provides a high degree of control and a clear audit trail. Model artifacts can be stored in a secure, internal registry, scanned for vulnerabilities, and then programmatically transferred to the PVC. This creates a hermetically sealed path from artifact to deployment. The Hub-based method introduces a dependency on an external service and requires managing access tokens within the cluster.35

Third is operational overhead versus iteration speed. The PVC approach entails more operational steps: provisioning and managing the lifecycle of the persistent volume and implementing a CI/CD process to update the model files on it. In contrast, the Hub approach is operationally simpler, as KServe's runtime handles the model download. This simplicity translates to much faster iteration cycles; a data scientist can push a new model version to the Hub, and a simple update to the InferenceService manifest is all that is required to deploy it.35

| Factor | PVC Deployment | Hugging Face Hub Deployment |
| --- | --- | --- |
| **Security & Control** | **High.** Full control over model artifacts. Suitable for air-gapped and highly regulated environments. | **Moderate.** Relies on an external service. Requires secure management of access tokens. |
| **Operational Overhead** | **Higher.** Requires management of PVC lifecycle and a process for model file transfer. | **Lower.** KServe runtime and Hugging Face Hub handle model storage and retrieval. |
| **Iteration Speed** | **Slower.** Updating a model requires a multi-step process to update the volume's contents. | **Faster.** A simple update to the InferenceService YAML can deploy a new model version. |
| **Network Dependency** | **None** at inference time. The model is already on the cluster's storage. | **High** dependency on network connectivity to the Hub during pod startup or scaling events. |
| **Primary Use Case** | Production deployments in secure, regulated, or offline environments. Serving large, static models. | Rapid prototyping, development, and CI/CD-driven production workflows in cloud-native environments. |

### 4.3 End-to-End Workflow Synthesis

By integrating the insights from the preceding analyses, a complete, best-practice workflow for moving from fine-tuning to production deployment can be established:

1. **Select Fine-Tuning Method:** Based on the framework in Section 4.1, choose either LoRA for speed when resources permit, or QLoRA for memory efficiency when resources are constrained.
2. **Execute Fine-Tuning:** Run the training process to produce the lightweight LoRA adapter weights as the primary output artifact.
3. **Merge for Inference:** As detailed in Section 2, load the original pre-trained base model in a high-precision format (e.g., BFloat16). Apply the trained adapter weights and execute the merge\_and\_unload() function. This creates a single, standalone, inference-optimized model.
4. **Package and Store Artifact:** Based on the deployment strategy determined in Section 4.2, either push the merged model directory to a repository on the Hugging Face Hub or package it and upload it to a secure internal artifact registry.
5. **Deploy on KServe:** Create and apply the appropriate InferenceService YAML manifest, configuring the storageUri to point to either the Hugging Face Hub repository (hf://...) or the path on the Persistent Volume Claim (pvc://...).

This structured workflow ensures that decisions made at each stage—from fine-tuning to deployment—are deliberate and aligned with the specific technical constraints and operational goals of the project, leading to a robust, efficient, and maintainable production system.

#### Works cited

1. Fine-Tuning using LoRA and QLoRA - GeeksforGeeks, accessed September 11, 2025, <https://www.geeksforgeeks.org/deep-learning/fine-tuning-using-lora-and-qlora/>
2. Full Fine-Tuning, LoRA, and qLoRA: A Comparison of Techniques in Model Optimization | by Gokcer Belgusen | Garanti BBVA Teknoloji | Medium, accessed September 11, 2025, <https://medium.com/garantibbva-teknoloji/full-fine-tuning-lora-and-qlora-a-comparison-of-techniques-in-model-optimization-89038d36865f>
3. LoRA vs. QLoRA - Red Hat, accessed September 11, 2025, <https://www.redhat.com/en/topics/ai/lora-vs-qlora>
4. Fine-Tuning LLMs with LoRA and QLoRA: From Confusion to (Kinda) Working Results | by Deepesh Sharma | Medium, accessed September 11, 2025, <https://medium.com/@dsh.2065/fine-tuning-llms-with-lora-and-qlora-from-confusion-to-kinda-working-results-89b348bcce71>
5. Fine-tune Llama 3.1 Ultra-Efficiently with Unsloth - Hugging Face, accessed September 11, 2025, <https://huggingface.co/blog/mlabonne/sft-llama3>
6. PEFT - Hugging Face, accessed September 11, 2025, <https://huggingface.co/docs/transformers/peft>
7. PEFT - Hugging Face, accessed September 11, 2025, <https://huggingface.co/docs/peft/index>
8. Finetuning LLMs with LoRA and QLoRA: Insights from Hundreds of Experiments, accessed September 11, 2025, <https://lightning.ai/pages/community/lora-insights/>
9. LoRA - Hugging Face, accessed September 11, 2025, <https://huggingface.co/docs/peft/main/conceptual_guides/lora>
10. COMPARISON OF LORA, DORA, AND QLORA, accessed September 11, 2025, <http://www.cs.sjsu.edu/faculty/pollett/masters/Semesters/Fall24/alisha/Different_fine_tuning_models.pdf>
11. Difference between QLoRA and LoRA for Fine-Tuning LLMs. | by Sujatha Mudadla | Medium, accessed September 11, 2025, <https://medium.com/@sujathamudadla1213/difference-between-qlora-and-lora-for-fine-tuning-llms-0ea35a195535>
12. LLM Optimization: LoRA and QLoRA - Towards Data Science, accessed September 11, 2025, <https://towardsdatascience.com/llm-optimization-lora-and-qlora/>
13. LoRA vs Full Fine-tuning: An Illusion of Equivalence - arXiv, accessed September 11, 2025, <https://arxiv.org/html/2410.21228v1>
14. LoRA (Low-Rank Adaptation) - Hugging Face LLM Course, accessed September 11, 2025, <https://huggingface.co/learn/llm-course/chapter11/4>
15. QLoRA and 4-bit Quantization - Chris McCormick, accessed September 11, 2025, <https://mccormickml.com/2024/09/14/qlora-and-4bit-quantization/>
16. artidoro/qlora: QLoRA: Efficient Finetuning of Quantized LLMs - GitHub, accessed September 11, 2025, <https://github.com/artidoro/qlora>
17. (LoRA) Fine-Tuning FLUX.1-dev on Consumer Hardware, accessed September 11, 2025, <https://huggingface.co/blog/flux-qlora>
18. Accurate LoRA-Finetuning Quantization of LLMs via Information Retention - arXiv, accessed September 11, 2025, <https://arxiv.org/pdf/2402.05445>
19. LoRA and QLoRA recommendations for LLMs | Generative AI on ..., accessed September 11, 2025, <https://cloud.google.com/vertex-ai/generative-ai/docs/model-garden/lora-qlora>
20. LoRA vs. QLoRA performance comparison · Issue #511 · ml-explore/mlx-examples - GitHub, accessed September 11, 2025, <https://github.com/ml-explore/mlx-examples/issues/511>
21. LoRA - Hugging Face, accessed September 11, 2025, <https://huggingface.co/docs/peft/main/en/conceptual_guides/lora>
22. Merging LoRA Adapters with Base Model - Kaggle, accessed September 11, 2025, <https://www.kaggle.com/code/ebinbt007/merging-lora-adapters-with-base-model>
23. Does merging of based model with LORA weight mandatory for LLAMA2? - Reddit, accessed September 11, 2025, <https://www.reddit.com/r/LocalLLaMA/comments/1b6zxn8/does_merging_of_based_model_with_lora_weight/>
24. Merging Adapters - LoRAX Docs, accessed September 11, 2025, <https://loraexchange.ai/guides/merging_adapters/>
25. Model merging - Hugging Face, accessed September 11, 2025, <https://huggingface.co/docs/peft/developer_guides/model_merging>
26. kserve/kserve: Standardized Distributed Generative and Predictive AI Inference Platform for Scalable, Multi-Framework Deployment on Kubernetes - GitHub, accessed September 11, 2025, <https://github.com/kserve/kserve>
27. KServe Resources - GitHub Pages, accessed September 11, 2025, <https://kserve.github.io/website/docs/concepts/resources>
28. KServe - GitHub Pages, accessed September 11, 2025, <https://kserve.github.io/website/>
29. Hugging Face Runtime Overview - KServe - GitHub Pages, accessed September 11, 2025, <https://kserve.github.io/website/docs/model-serving/generative-inference/overview>
30. Serving AI Models from Kubernetes Persistent Volumes with KServe ..., accessed September 11, 2025, <https://developer.ibm.com/tutorials/awb-use-kubernetes-persistent-volumes-with-kserve-modelmesh/>
31. Persistent Volumes - Kubernetes, accessed September 11, 2025, <https://kubernetes.io/docs/concepts/storage/persistent-volumes/>
32. kserve - PyPI, accessed September 11, 2025, <https://pypi.org/project/kserve/>
33. Announcing KServe v0.15: Advancing Generative AI Model Serving | CNCF, accessed September 11, 2025, <https://www.cncf.io/blog/2025/06/18/announcing-kserve-v0-15-advancing-generative-ai-model-serving/>
34. Kserve | GKE AI Labs, accessed September 11, 2025, <https://gke-ai-labs.dev/docs/tutorials/inference-servers/kserve/>
35. 7 best KServe alternatives in 2025 for scalable model deployment | Blog - Northflank, accessed September 11, 2025, <https://northflank.com/blog/kserve-alternatives>
36. Serving models | opendatahub.io, accessed September 11, 2025, <https://opendatahub.io/docs/serving-models/>
37. Which architecture does Hugging face use for model serving ? Are they using kserve ? : r/mlops - Reddit, accessed September 11, 2025, <https://www.reddit.com/r/mlops/comments/13vxhrm/which_architecture_does_hugging_face_use_for/>