Machine Learning Frameworks for Deployment and Experiment Tracking: Gradio, BentoML, MLflow, and DagHub

Executive Summary

Machine learning model development is only half the battle—deploying models effectively, tracking experiments, and making models accessible are equally crucial. **Gradio**, **BentoML**, **MLflow**, and **DagHub** are four powerful frameworks that address different aspects of the ML lifecycle. This comprehensive guide explores what each framework does, when to use them, and how they complement each other in modern MLOps workflows.

Key Takeaways:

- **Gradio** excels at rapid UI creation for ML demos with minimal code
- **BentoML** provides production-grade model serving with advanced optimization
- **MLflow** offers comprehensive experiment tracking and model registry capabilities
- **DagHub** simplifies MLflow deployment with zero-configuration remote servers and Git integration

1. Gradio Framework: Interactive ML Interfaces Made Simple

1.1 What is Gradio?

Gradio is an open-source Python library designed to create interactive web interfaces for machine learning models with minimal code[1] [2]. It transforms ML models into shareable web applications in just a few lines of Python, eliminating the need for HTML, CSS, or JavaScript knowledge[3].

Core Philosophy: Focus on your model, not the interface. Gradio automatically generates user-friendly UIs that work on any device with a browser[1].

1.2 Key Features

- **Minimal Code Requirements:** Create functional demos with 3-5 lines of code[1] [4]
- **Multiple Input Types:** Supports text, images, audio, video, and more[3]
- **Automatic API Generation:** Creates RESTful APIs automatically[4]
- **Shareable Links:** Generate public URLs for instant sharing[4]
- **Hugging Face Integration:** Deploy directly to Hugging Face Spaces[5]
- **Real-Time Updates:** Supports live streaming for real-time applications[6]

1.3 When to Use Gradio

Ideal Scenarios: [7] [8]

- Quick model demonstrations and prototyping
- Sharing models with non-technical stakeholders
- Educational purposes and tutorials
- Research paper demonstrations
- Internal testing and validation

Not Recommended For:

- Production-scale deployments requiring high throughput
- Complex multi-model orchestration
- Applications requiring extensive backend logic
- Enterprise systems with strict security requirements

1.4 Gradio Code Example: Image Classification

Here's a complete example of building an image classification interface:

```
import gradio as gr
import tensorflow as tf
import numpy as np

# Load pre-trained model
model = tf.keras.applications.MobileNetV2(weights="imagenet")

def classify_image(image):
    """Classify an image and return top predictions"""
    # Preprocess image
    image = tf.image.resize(image, (224, 224))
    image = np.expand_dims(image, axis=0)
    image = tf.keras.applications.mobilenet_v2.preprocess_input(image)

    # Make predictions
    predictions = model.predict(image)

    # Decode and format results
    decoded = tf.keras.applications.mobilenet_v2.decode_predictions(predictions, top=3)[
    return {label: float(prob) for (_, label, prob) in decoded}

# Create Gradio interface - just one line!
interface = gr.Interface(
    fn=classify_image,
    inputs=gr.Image(),
    outputs=gr.Label(num_top_classes=3),
    title="Image Classification Demo",
    description="Upload an image to classify it using MobileNetV2"
)
```

```
# Launch the interface
interface.launch(share=True)  # share=True creates a public link
```

What This Does: [9] [10] [11]

1. Loads a pre-trained MobileNetV2 model
2. Defines a prediction function that processes images
3. Creates an interactive UI with drag-and-drop image upload
4. Displays top 3 predictions with confidence scores
5. Generates a shareable public URL

1.5 Gradio vs. Streamlit

While both create web interfaces for ML models, they serve different purposes [12] [6] [8]:

Aspect	Gradio	Streamlit
Primary Focus	ML model demos	Data dashboards and apps
Learning Curve	Easier, more intuitive	Steeper, more features
Customization	Limited but sufficient	Extensive customization
Best For	Quick prototypes, model showcasing	Complex data applications
ML Integration	Excellent (built specifically for ML)	Good (general-purpose)

Recommendation: Use Gradio for ML model interfaces, Streamlit for data-heavy applications requiring complex layouts [6] [8].

1.6 Deployment Options

Gradio offers multiple deployment paths [1] [4]:

1. **Local Development:** `interface.launch()` starts a local server
2. **Public Links:** `interface.launch(share=True)` creates temporary public URLs
3. **Hugging Face Spaces:** One-click deployment to Hugging Face infrastructure [5]
4. **Custom Servers:** Can be embedded in Flask/FastAPI applications

2. BentoML: Production-Grade Model Serving

2.1 What is BentoML?

BentoML is a unified framework for building production-ready AI applications and deploying ML models at scale [13] [14]. It bridges the gap between model development and production deployment by providing standardized packaging, serving optimization, and deployment automation [15].

Core Philosophy: Make production deployment as simple as writing Python code, without sacrificing performance or flexibility [13] .

2.2 Key Features

- **Framework Agnostic:** Supports scikit-learn, PyTorch, TensorFlow, XGBoost, Hugging Face, and more [13] [14]
- **Performance Optimization:** 100x throughput improvement via micro-batching [16] [17]
- **Automatic API Generation:** Creates REST and gRPC APIs automatically [14] [18]
- **Containerization:** Built-in Docker image generation [14] [19]
- **Model Versioning:** Local model store with version tracking [14] [15]
- **Multi-Model Serving:** Orchestrate multiple models in a single service [18]

2.3 BentoML Architecture

BentoML follows a structured workflow [19] [18] :

1. **Train Model:** Use any ML framework
2. **Save to Model Store:** BentoML maintains a local model repository
3. **Define Service:** Create API endpoints with decorators
4. **Build Bento:** Package everything into a deployable unit
5. **Containerize:** Generate Docker images automatically
6. **Deploy:** Deploy to any cloud platform

2.4 BentoML Code Example: Complete Workflow

Step 1: Train and Save Model

```
import bentoml
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# Train model
X, y = load_iris(return_X_y=True)
model = RandomForestClassifier()
model.fit(X, y)

# Save to BentoML model store
bentoml.sklearn.save_model(
    "iris_classifier",  # Model name
    model,
    signatures={
        "predict": {"batchable": True, "batch_dim": 0}
    }
)
```

Step 2: Create Service Definition

Create `service.py`:

```
import bentoml
import numpy as np
from bentoml.io import NumpyNdarray

# Load the model
iris_model = bentoml.models.get("iris_classifier:latest")

# Define the service
@bentoml.service(
    resources={"cpu": "2", "memory": "500MiB"},
    traffic={"timeout": 10}
)
class IrisClassifier:

    def __init__(self):
        # Load model into runner for optimized inference
        self.model = bentoml.sklearn.get("iris_classifier:latest")

    @bentoml.api(
        input=NumpyNdarray(),
        output=NumpyNdarray()
    )
    def classify(self, input_data: np.ndarray) -&gt; np.ndarray:
        """Classify iris species"""
        return self.model.predict(input_data)
```

Step 3: Serve Locally for Testing

```
# Start the service locally
bentoml serve service:IrisClassifier --reload

# Service runs at http://localhost:3000
# Automatic Swagger UI at http://localhost:3000
```

Step 4: Build Bento for Deployment

Create `bentofile.yaml`:

```
service: "service:IrisClassifier"
labels:
  owner: "ml-team"
  project: "iris-classification"
include:
  - "*.py"
python:
  packages:
    - scikit-learn
    - numpy
```

Build the Bento:

```
# Package everything
bentoml build

# Containerize
bentoml containerize iris_classifier:latest

# Run Docker container
docker run -p 3000:3000 iris_classifier:latest
```

2.5 BentoML Performance Optimization

Micro-Batching: [14] [17]

BentoML automatically batches incoming requests to maximize GPU/CPU utilization:

```
@bentoml.service
class OptimizedService:

    @bentoml.api(
        input=NumpyNdarray(),
        output=NumpyNdarray(),
        batchable=True,  # Enable batching
        batch_dim=0,
        max_batch_size=32,
        max_latency_ms=100
    )
    def predict(self, input_data: np.ndarray) -&gt; np.ndarray:
        return self.model.predict(input_data)
```

This configuration:

- Batches up to 32 requests together
- Waits maximum 100ms before processing
- Results in 10-100x throughput improvement

2.6 Deployment Targets

BentoML supports deployment to [13] [17]:

- **Kubernetes:** Native support with Helm charts
- **AWS:** ECS, Lambda, SageMaker
- **Google Cloud:** Cloud Run, GKE
- **Azure:** Container Apps, AKS
- **BentoCloud:** Managed deployment platform

3. MLflow: Complete ML Lifecycle Management

3.1 What is MLflow?

MLflow is an open-source platform for managing the complete machine learning lifecycle, including experimentation, reproducibility, deployment, and model registry[20] [21]. It provides a centralized system for tracking experiments, versioning models, and facilitating team collaboration[22].

Core Philosophy: Provide a unified platform that works with any ML library, algorithm, or deployment tool[23].

3.2 MLflow Components

MLflow consists of four main components[21] [23]:

1. **MLflow Tracking:** Log and query experiments (parameters, metrics, artifacts)
2. **MLflow Projects:** Package ML code for reproducibility
3. **MLflow Models:** Deploy models to various serving environments
4. **MLflow Model Registry:** Centralized model versioning and lifecycle management

3.3 Key Features

- **Experiment Tracking:** Log parameters, metrics, and artifacts automatically[21] [23]
- **Model Registry:** Version control and staging for models[20] [22]
- **Model Lineage:** Track which experiments produced which models[22]
- **Multi-Framework Support:** Works with any ML framework[23]
- **Collaboration:** Team-based model management[21] [22]
- **Deployment Integration:** Deploy to multiple platforms[24]

3.4 MLflow Code Example: Experiment Tracking

Basic Experiment Tracking

```
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load data
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Set tracking URI (optional, defaults to local)
```

```
mlflow.set_tracking_uri("http://localhost:5000")

# Create or set experiment
mlflow.set_experiment("iris-classification")

# Start MLflow run
with mlflow.start_run():

    # Define hyperparameters
    params = {
        "n_estimators": 100,
        "max_depth": 5,
        "random_state": 42
    }

    # Log parameters
    mlflow.log_params(params)

    # Train model
    model = RandomForestClassifier(**params)
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Calculate and log metrics
    accuracy = accuracy_score(y_test, y_pred)
    mlflow.log_metric("accuracy", accuracy)
    mlflow.log_metric("test_samples", len(y_test))

    # Log the model
    mlflow.sklearn.log_model(
        model,
        "model",
        registered_model_name="iris_classifier"
    )

    # Log additional artifacts
    import matplotlib.pyplot as plt
    plt.figure()
    plt.hist(y_pred)
    plt.savefig("predictions.png")
    mlflow.log_artifact("predictions.png")

    print(f"Model accuracy: {accuracy:.3f}")
    print(f"Run ID: {mlflow.active_run().info.run_id}")
```

Auto-Logging

MLflow provides automatic logging for popular frameworks[25]:

```
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
```

```
# Enable auto-logging
mlflow.sklearn.autolog()

# Train your model normally - MLflow logs everything automatically!
model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)

# Parameters, metrics, and model are logged automatically
```

3.5 MLflow Model Registry

The Model Registry provides centralized model management[20] [22]:

```
import mlflow
from mlflow.tracking import MlflowClient

client = MlflowClient()

# Register a model
model_uri = "runs:/abc123/model"
model_name = "iris_classifier"

mlflow.register_model(
    model_uri=model_uri,
    name=model_name
)

# Transition model to production
client.transition_model_version_stage(
    name=model_name,
    version=1,
    stage="Production"
)

# Add model alias
client.set_registered_model_alias(
    name=model_name,
    alias="champion",
    version="1"
)

# Load model from registry
model = mlflow.pyfunc.load_model(
    f"models:/{model_name}@champion"
)
```

3.6 MLflow UI

MLflow provides a built-in web UI for visualizing experiments[21] [23]:

```
# Start MLflow UI
mlflow ui --port 5000
```

```
# Access at http://localhost:5000
```

UI Features:

- Compare multiple runs side-by-side
- Visualize metrics over time
- Search and filter experiments
- Download artifacts
- Manage model registry

4. DagHub: Collaborative MLOps Platform

4.1 What is DagHub?

DagHub is a Git-based platform for data scientists that provides zero-configuration remote MLflow servers, data versioning, and team collaboration [26] [27]. It integrates MLflow tracking with Git workflows, eliminating the DevOps overhead of setting up remote tracking servers [27].

Core Philosophy: Make MLflow accessible without infrastructure complexity while maintaining Git-based version control [26] [27].

4.2 Key Features

- **Zero-Configuration MLflow Server:** Automatic MLflow server with every repository [27]
- **Git Integration:** Track code, data, and experiments together [28] [29]
- **Team Access Control:** Built-in authentication and permissions [27]
- **Remote Storage:** Cloud-based artifact storage [27]
- **Data Versioning:** DVC integration for large files [28]
- **Free Tier:** Generous free hosting for public projects [27]

4.3 DagHub Setup and Integration

Step 1: Create DagHub Repository

1. Sign up at https://dagshub.com
2. Create a new repository or connect existing GitHub repo
3. DagHub automatically provisions an MLflow server at: `https://dagshub.com/&lt;username&gt;/&lt;repo-name&gt;.mlflow`

Step 2: Configure Local Environment

```
import dagshub
import mlflow

# Initialize DagHub integration
dagshub.init(
    repo_owner="your-username",
    repo_name="your-project",
    mlflow=True
)

# MLflow is now configured to use DagHub's remote server
# All mlflow commands will log to DagHub automatically

with mlflow.start_run():
    mlflow.log_param("learning_rate", 0.01)
    mlflow.log_metric("accuracy", 0.95)
    # ... rest of your training code
```

Step 3: Alternative Configuration

```
import mlflow
import os

# Set environment variables
os.environ['MLFLOW_TRACKING_URI'] = 'https://dagshub.com/username/repo.mlflow'
os.environ['MLFLOW_TRACKING_USERNAME'] = 'your-username'
os.environ['MLFLOW_TRACKING_PASSWORD'] = 'your-token'  # Get from DagHub settings

# Use MLflow normally
mlflow.set_tracking_uri('https://dagshub.com/username/repo.mlflow')
```

4.4 DagHub vs. Self-Hosted MLflow

Aspect	DagHub	Self-Hosted MLflow
Setup Time	Instant (zero config)	Hours to days
Infrastructure	Fully managed	Manual setup required
Access Control	Built-in	Must configure
Cost	Free tier available	Infrastructure costs
Git Integration	Native	Manual setup
Maintenance	None required	Ongoing DevOps work

4.5 DagHub with Data Version Control (DVC)

DagHub integrates DVC for versioning large datasets [28] [29]:

```
# Initialize DVC
dvc init

# Add large data files
dvc add data/large_dataset.csv

# Configure DagHub remote
dvc remote add origin https://dagshub.com/username/repo.dvc

# Push data to DagHub
dvc push

# Track with Git
git add data/large_dataset.csv.dvc .dvc/config
git commit -m "Add dataset"
git push
```

5. Framework Comparison and Integration

5.1 When to Use Each Framework

Use Gradio When: [7] [8]

- Building quick demos or prototypes
- Sharing models with non-technical stakeholders
- Creating educational materials
- Testing models interactively during development
- Prioritizing speed over production features

Use BentoML When: [13] [14] [17]

- Deploying models to production
- Requiring high-throughput serving
- Need containerization and cloud deployment
- Managing multiple model versions
- Optimizing inference performance

Use MLflow When: [20] [21] [22]

- Tracking multiple experiments systematically
- Managing model lifecycle and versions
- Collaborating with a team on ML projects

- Requiring model lineage and reproducibility
- Comparing different model iterations

Use DagHub When: [26] [27]

- Want MLflow without DevOps overhead
- Need remote experiment tracking
- Collaborating with distributed teams
- Versioning both code and data
- Requiring Git-based workflows

5.2 Combining Frameworks

These frameworks complement each other excellently [15]:

MLflow + BentoML Pipeline: [15]

```
import mlflow
import bentoml

# Track experiments with MLflow
with mlflow.start_run():
    # Train and log model
    mlflow.sklearn.log_model(model, "model")
    model_uri = mlflow.get_artifact_uri("model")

# Register in MLflow
mlflow.register_model(model_uri, "production_model")

# Import to BentoML for serving
bentoml.mlflow.import_model(
    "production_model",
    model_uri=f"models:/production_model/Production"
)

# Deploy with BentoML
@bentoml.service
class ProductionService:
    model = bentoml.models.get("production_model:latest")

    @bentoml.api
    def predict(self, input_data):
        return self.model.predict(input_data)
```

Gradio + BentoML Pipeline:

```
import gradio as gr
import bentoml

# Load BentoML model
model = bentoml.sklearn.get("iris_classifier:latest")
```

```
# Create Gradio interface for the BentoML model
def predict(input_data):
    return model.predict(input_data)

gr.Interface(
    fn=predict,
    inputs=gr.Slider(0, 10),
    outputs="number"
).launch()
```

5.3 Complete MLOps Workflow

A production-ready workflow combining all frameworks:

1. **Development Phase (Gradio):**
    - Rapidly prototype model interfaces
    - Test with stakeholders
    - Gather feedback quickly
2. **Experiment Tracking (MLflow + DagHub):**
    - Log all experiments to DagHub
    - Compare model performance
    - Track hyperparameters
    - Version control code and data
3. **Model Registration (MLflow):**
    - Register best models
    - Add metadata and descriptions
    - Stage models (Development → Staging → Production)
4. **Production Deployment (BentoML):**
    - Package production model
    - Optimize for performance
    - Containerize and deploy
    - Monitor in production

6. Practical Examples and Best Practices

6.1 Example: End-to-End Image Classification

Phase 1: Development with Gradio

```
import gradio as gr
import tensorflow as tf

model = tf.keras.applications.MobileNetV2(weights='imagenet')

def classify(image):
    # Quick prototype for testing
    image = tf.image.resize(image, (224, 224))
    predictions = model.predict(image)
    return decoded_predictions

# Quick demo for team review
gr.Interface(fn=classify, inputs="image", outputs="label").launch()
```

Phase 2: Experiment Tracking with MLflow + DagHub

```
import mlflow
import dagshub

# Configure DagHub
dagshub.init("username", "image-classifier", mlflow=True)

# Track multiple model variations
for lr in [0.001, 0.01, 0.1]:
    with mlflow.start_run():
        mlflow.log_param("learning_rate", lr)

        # Train model
        model = train_model(learning_rate=lr)

        # Log metrics
        accuracy = evaluate_model(model)
        mlflow.log_metric("accuracy", accuracy)

        # Log model
        mlflow.tensorflow.log_model(model, "model")

# Register best model
best_model_uri = "runs:/best-run-id/model"
mlflow.register_model(best_model_uri, "image_classifier_prod")
```

Phase 3: Production Deployment with BentoML

```
import bentoml
import mlflow
import numpy as np
```

```
# Import from MLflow
model_uri = "models:/image_classifier_prod/Production"
bentoml.mlflow.import_model(
    "image_classifier",
    model_uri=model_uri
)

# Create production service
@bentoml.service(
    resources={"gpu": "1", "memory": "4Gi"},
    traffic={"timeout": 30}
)
class ImageClassifier:

    model = bentoml.models.get("image_classifier:latest")

    @bentoml.api(
        input=bentoml.io.Image(),
        output=bentoml.io.JSON(),
        batchable=True,
        max_batch_size=32
    )
    async def classify(self, images):
        results = await self.model.predict(images)
        return results
```

6.2 Best Practices

Gradio Best Practices

1. **Keep Interfaces Simple:** Focus on core functionality
2. **Add Examples:** Pre-load sample inputs for users
3. **Use Descriptions:** Document what the model does
4. **Enable Sharing Carefully:** Be mindful of sensitive data
5. **Cache Results:** Use caching for expensive predictions

BentoML Best Practices

1. **Version Models Explicitly:** Use semantic versioning
2. **Configure Resources:** Specify CPU/GPU requirements
3. **Enable Batching:** For high-throughput scenarios
4. **Monitor Performance:** Track latency and throughput
5. **Use Async APIs:** For I/O-bound operations

MLflow Best Practices

1. **Use Descriptive Names:** Clear experiment and run names
2. **Log Everything:** Parameters, metrics, and artifacts
3. **Tag Strategically:** Use tags for organization
4. **Document Models:** Add descriptions in registry
5. **Automate Tracking:** Use autolog when possible

DagHub Best Practices

1. **Commit Regularly:** Keep code and experiments in sync
2. **Use DVC for Large Files:** Don't commit large datasets to Git
3. **Document Experiments:** Use README and notebooks
4. **Set Up Webhooks:** Automate notifications
5. **Manage Permissions:** Control team access appropriately

7. Common Pitfalls and Solutions

7.1 Gradio Pitfalls

Problem: Interface is slow with large files
Solution: Implement preprocessing and compression before processing

Problem: Temporary share links expire
Solution: Deploy to Hugging Face Spaces for permanent hosting

7.2 BentoML Pitfalls

Problem: Docker images are too large
Solution: Use multi-stage builds and exclude unnecessary dependencies

Problem: Model inference is slow
Solution: Enable batching and adjust batch size/latency parameters

7.3 MLflow Pitfalls

Problem: Tracking server runs out of disk space
Solution: Configure artifact storage to cloud (S3, Azure Blob)

Problem: Experiments are disorganized
Solution: Use consistent naming conventions and tags

7.4 DagHub Pitfalls

Problem: Large files in Git repository
Solution: Use DVC for all files > 10MB

Problem: Slow pushes to remote
Solution: Configure parallel uploads in DVC

8. Conclusion and Recommendations

8.1 Framework Selection Matrix

Project Type	Recommended Stack	Reason
Solo Research Project	Gradio + MLflow	Quick prototyping with experiment tracking
Team Research Project	Gradio + MLflow + DagHub	Collaboration with zero DevOps
Production Deployment	BentoML + MLflow	Performance with lineage tracking
Enterprise MLOps	All Four	Complete lifecycle coverage
Quick Demo	Gradio Only	Fastest time to demo
High-Scale API	BentoML Only	Best performance optimization

8.2 Learning Path

Week 1: Start with Gradio

- Learn basic interface creation
- Experiment with different input/output types
- Build 2-3 simple demos

Week 2: Add MLflow

- Set up local MLflow tracking
- Log experiments systematically
- Use MLflow UI to compare runs

Week 3: Integrate DagHub

- Create DagHub repository
- Configure remote tracking
- Practice collaborative workflows

Week 4: Deploy with BentoML

- Package trained models
- Create production services

- Deploy to cloud platform

8.3 Final Thoughts

The modern ML workflow requires multiple specialized tools. Rather than viewing these frameworks as competitors, think of them as complementary pieces of a complete MLOps toolkit:

- **Gradio** makes your models interactive and shareable
- **BentoML** makes them production-ready and scalable
- **MLflow** makes your experiments trackable and reproducible
- **DagHub** makes collaboration effortless

By mastering all four, you'll have the complete skillset to take ML projects from initial prototype to production deployment efficiently and professionally.

References

[30] Krish Naik, "Gradio Library - Interfaces for your Machine Learning Models," YouTube, 2020
[31] Krish Naik, "BentoML Tutorial: Build Production Grade AI Applications," YouTube, 2023
[1] GeeksforGeeks, "Creating Interactive Machine Learning Demos with Gradio," 2024
[13] Gocodeo, "BentoML: Deploying Machine Learning Models Made Simple," 2025
[20] ZenML Documentation, "MLflow Model Registry," 2023
[2] Gradio Documentation, "Quickstart," 2022
[14] GeeksforGeeks, "BentoML: Helping Deploy ML Models," 2025
[21] lakeFS Blog, "MLflow Model Registry: Workflows, Benefits & Challenges," 2025
[3] freeCodeCamp, "How to Build Your AI Demos with Gradio," 2025
[19] Valerio Velardo, "How to Deploy ML Models in Production with BentoML," YouTube, 2022
[23] MLflow Documentation, "MLflow Model Registry," 2021
[5] Cognitive Class, "Bring your Machine Learning model to life with Gradio," 2024
[32] BentoML Blog, "Why Do People Say It's So Hard To Deploy A ML Model," 2023
[22] MLflow Documentation, "MLflow Model Registry," 2024
[4] Gradio Official Website, "Gradio," 2023
[33] BentoML Blog, "Design Considerations For Model Deployment Systems," 2023
[24] MLflow Documentation, "MLflow Tracking," 2024
[34] Gradio Documentation, "Interface," 2022
[35] DataCamp, "How to Deploy LLMs with BentoML," 2025
[36] Microsoft Azure, "Manage models registry with MLflow," 2024
[7] DataCamp, "Building User Interfaces For AI Applications with Gradio," 2024
[37] BentoML Blog, "The Easiest Way To Deploy Machine Learning Models," 2022
[26] DagHub Blog, "Launching DAGsHub integration with MLflow," 2025
[38] LinkedIn, "MLflow vs BentoML Comparison," 2024
[12] MyScale Blog, "Streamlit vs Gradio: The Ultimate Showdown," 2024
[27] DagHub Blog, "Free Remote MLflow Tracking Server," 2023
[16] Slashdot, "Compare BentoML vs. MLflow in 2025," 2025
[6] UnfoldAI, "Streamlit vs Gradio — Choosing the right framework," 2024
[28] Hackmamba, "ML experiment tracking with DagsHub, MLFlow, and DVC," 2022

[15] BentoML Blog, "Building ML Pipelines with MLflow and BentoML," 2023

[8] UIBakery, "Streamlit vs Gradio: The Ultimate Showdown," 2025

[25] DagsHub, "Experiment Tracking for Machine Learning with MLflow," YouTube, 2023

[33-42] Various supporting references and documentation sources

[39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55]

⁂

1. https://docs.zenml.io/stacks/stack-components/model-registries/mlflow
2. https://lakefs.io/blog/mlflow-model-registry/
3. https://mlflow.org/docs/3.0.1/model-registry/
4. https://mlflow.org/docs/latest/ml/tracking/
5. https://mlflow.org/docs/latest/ml/model-registry/
6. https://www.bentoml.com/blog/building-ml-pipelines-with-mlflow-and-bentoml
7. https://dagshub.com/blog/launching-dagshub-integration-with-databricks-mlflow/
8. https://sourceforge.net/software/compare/BentoML-vs-MLflow/
9. https://github.com/bentoml/BentoML
10. https://github.com/JuliaAI/MLJ.jl/issues/1075
11. https://docs.newrelic.com/docs/mlops/integrations/dagshub-mlops-integration/
12. https://slashdot.org/software/comparison/BentoML-vs-MLflow/
13. https://www.gradio.app/guides/quickstart
14. https://www.freecodecamp.org/news/how-to-build-your-ai-demos-with-gradio/
15. https://www.youtube.com/watch?v=JmCfkpGOE8c
16. https://hackmamba.io/engineering/ml-experiment-tracking-with-dagshub-mlflow-and-dvc/
17. https://dagshub.com/docs/integration_guide/mlflow_tracking/
18. https://www.gradio.app/4.44.1/guides/image-classification-in-tensorflow
19. https://cognitiveclass.ai/courses/bring-your-machine-learning-model-to-life-with-gradio
20. https://www.geeksforgeeks.org/machine-learning/bentoml-helping-deploy-ml-models/
21. https://www.youtube.com/watch?v=HHkmfI_yncc
22. https://www.bentoml.com/blog/ml-requirements
23. https://bentoml.com/blog/why-do-people-say-its-so-hard-to-deploy-a-ml-model-to-production
24. https://www.datacamp.com/tutorial/deploy-llms-with-bentoml
25. https://evidence.dev/learn/gradio-vs-streamlit
26. https://myscale.com/blog/streamlit-vs-gradio-ultimate-showdown-python-dashboards/
27. https://unfoldai.com/streamlit-vs-gradio/
28. https://uibakery.io/blog/streamlit-vs-gradio
29. https://www.youtube.com/watch?v=rJ4_7pnIRmA
30. https://www.geeksforgeeks.org/artificial-intelligence/creating-interactive-machine-learning-demos-with-gradio/
31. https://www.gocodeo.com/post/bentoml-deploying-machine-learning-models-made-simple
32. https://gradio.app

33. https://www.gradio.app/docs/gradio/interface
34. https://learn.microsoft.com/en-us/azure/machine-learning/how-to-manage-models-mlflow?view=azureml-api-2
35. https://www.datacamp.com/tutorial/gradio-python-tutorial
36. https://bentoml.com/blog/the-easiest-way-to-deploy-your-machine-learning-models-in-2022-streamlit-bentoml-dagshub
37. https://www.linkedin.com/posts/neuralgrade_mlflow-vs-bentoml-comparison-activity-7175426772623126528-M2LF
38. https://dagshub.com/blog/free-remote-mlflow-server/
39. https://dagshub.com/docs/feature_guide/experiment_tracking/
40. https://northflank.com/blog/mlflow-alternatives
41. https://www.reddit.com/r/LocalLLaMA/comments/180kvsx/gradio_or_streamlit_for_prototyping_and_why/
42. https://dagshub.com/docs/use_cases/track_ml_experiments/
43. https://www.truefoundry.com/blog/model-deployment-tools
44. https://www.zenml.io/blog/mlflow-alternatives
45. https://www.gradio.app/guides/image-classification-in-pytorch
46. https://docs.bentoml.com/en/latest/build-with-bentoml/services.html
47. https://www.youtube.com/watch?v=K2i-9Gn4XNY
48. https://www.gradio.app/guides/image-classification-with-vision-transformers
49. https://docs.bentoml.com/en/latest/build-with-bentoml/iotypes.html
50. https://towardsdatascience.com/creating-a-simple-image-classification-machine-learning-demo-with-gradioml-361a245d7b50/
51. https://docs.bentoml.com/en/latest/get-started/hello-world.html
52. https://www.youtube.com/watch?v=jOmzWK0CT28
53. https://github.com/JYe9/gradio_img_classfication_demo
54. https://docs.bentoml.com/en/latest/examples/overview.html
55. https://www.youtube.com/watch?v=a8aS3ZYlzDM