Flask API in Data Science: Deploying Machine Learning Models as Web Services

Technical Article | DataCraft Portfolio

✓ By Timothée Nkwar

Published: November 01, 2025

Generated: 2026-04-07T03:16:20.968615-07:00

Introduction

Data scientists often train powerful machine learning models in Jupyter notebooks, but to deliver real-world impact, these models must be **deployed** and made accessible to applications, users, or teams. Modern Python web frameworks like **FastAPI** and **Flask** enable data scientists to transform trained models into **RESTful APIs** that can receive input data and return predictions in real time.

This article explores how to serve machine learning models using modern frameworks and practices, covering foundational concepts, practical implementation, code examples, deployment strategies, and best practices. Whether you're predicting customer churn, classifying images, or forecasting sales, these frameworks provide simple yet scalable ways to productionize your models with type safety and high performance.

Modern Frameworks for Model Serving

Today's best choices for creating **model inference APIs** are **FastAPI** and **Flask**, each with distinct advantages:

FastAPI (Modern Choice)

- **Async/Await**: Handle concurrent requests efficiently.
- **Type Hints**: Built-in validation with Pydantic.
- **Auto Documentation**: Swagger UI and ReDoc generated automatically.
- **High Performance**: ASGI-based, faster than traditional WSGI.
- **Easy Integration**: Works seamlessly with `scikit-learn`, `TensorFlow`, `PyTorch`, and `XGBoost`.

Flask (Proven Simplicity)

- **Lightweight**: Minimal boilerplate code.
- **Flexible**: Full control over request/response handling.
- **Rapid Prototyping**: Build a working API in minutes.
- **Production-Ready**: With Uvicorn, Gunicorn, and Docker.

Real-World Use: Companies like Netflix, Airbnb, and modern startups use FastAPI for new APIs and Flask for legacy systems. Uber, Dropbox, and Spotify rely on high-performance model serving APIs.

Core Concepts of Flask Model APIs

RESTful Endpoints

A Flask API exposes **HTTP endpoints** that clients (web apps, mobile apps, or scripts) can call:

```
POST /predict
Content-Type: application/json

{ "sepal_length": 5.1, "sepal_width": 3.5, ... }
```

```
{
  "prediction": "setosa",
  "confidence": 1.0
}
```

Model Loading Strategy

The trained model (e.g., `.pkl`, `.h5`) is loaded **once at startup** to avoid latency:

```
model = joblib.load('model/model.pkl')  # Global variable
```

Input Validation with Pydantic

Always validate incoming JSON using type hints and schemas to prevent errors or security issues. Modern frameworks like FastAPI use **Pydantic** for automatic validation and serialization.

Implementation: Iris Classification API

We'll build a modern API that predicts Iris flower species using a Random Forest model. Both FastAPI and Flask examples are provided.

Project Structure

```
flask-iris-api/
├── app.py
├── model/
│   └── iris_model.pkl
├── utils/
│   └── validate.py
├── requirements.txt
└── Dockerfile
```

Model Training (One-Time)

```
# train.py
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
import joblib

iris = load_iris()
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(iris.data, iris.target)

joblib.dump(model, 'model/iris_model.pkl')
print("Model saved!")
```

`utils/validate.py` – Pydantic Validation (Modern Approach)

```
from pydantic import BaseModel, Field
from typing import Optional

class IrisInput(BaseModel):
    sepal_length: float = Field(..., ge=0, le=10, description="Sepal length in cm")
    sepal_width: float = Field(..., ge=0, le=10, description="Sepal width in cm")
    petal_length: float = Field(..., ge=0, le=10, description="Petal length in cm")
    petal_width: float = Field(..., ge=0, le=10, description="Petal width in cm")

    class Config:
        schema_extra = {
            "example": {
                "sepal_length": 5.1,
                "sepal_width": 3.5,
                "petal_length": 1.4,
                "petal_width": 0.2
            }
        }

class PredictionResponse(BaseModel):
    prediction: str
    confidence: float
    input: IrisInput
```

`app.py` – Modern FastAPI Implementation

```python
from fastapi import FastAPI, HTTPException
from contextlib import asynccontextmanager
import joblib
import os
from utils.validate import IrisInput, PredictionResponse
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

MODEL_PATH = 'model/iris_model.pkl'
model = None

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Load model at startup
    global model
    if not os.path.exists(MODEL_PATH):
        raise FileNotFoundError(f"Model not found: {MODEL_PATH}")
    model = joblib.load(MODEL_PATH)
    logger.info("Model loaded successfully")
    yield
    # Cleanup on shutdown
    logger.info("Shutting down...")

app = FastAPI(
    title="Iris Classification API",
    description="Predict Iris flower species using machine learning",
    version="2.0",
    lifespan=lifespan
)

@app.get("/")
async def home():
    return {
        "message": "Iris Classification API v2.0",
        "endpoint": "POST /predict",
        "docs": "/docs"
    }

@app.post("/predict", response_model=PredictionResponse)
async def predict(input_data: IrisInput):
    try:
        features = [
            input_data.sepal_length,
            input_data.sepal_width,
            input_data.petal_length,
            input_data.petal_width
        ]

        prediction = model.predict([features])[0]
        probability = model.predict_proba([features])[0].max()
        species = ['setosa', 'versicolor', 'virginica'][prediction]
```

```
        logger.info(f"Prediction: {species} ({probability:.4f})")

        return PredictionResponse(
            prediction=species,
            confidence=round(probability, 4),
            input=input_data
        )
    except Exception as e:
        logger.error(f"Prediction error: {str(e)}")
        raise HTTPException(status_code=500, detail="Prediction failed")

@app.get("/health")
async def health():
    return {"status": "healthy", "model": "loaded"}
```

Alternative: Flask Implementation (Traditional)

```python
from flask import Flask, request, jsonify
import joblib
import os
from utils.validate import IrisInput, PredictionResponse
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app = Flask(__name__)

# Load model at startup
MODEL_PATH = 'model/iris_model.pkl'
if not os.path.exists(MODEL_PATH):
    raise FileNotFoundError(f"Model not found: {MODEL_PATH}")

model = joblib.load(MODEL_PATH)
logger.info("Model loaded successfully")

@app.route('/')
def home():
    return jsonify({
        "message": "Iris Classification API",
        "endpoint": "POST /predict"
    })

@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json()
        if not data:
            return jsonify({"error": "No input data"}), 400

        # Validate with Pydantic
        input_data = IrisInput(**data)
        features = [input_data.sepal_length, input_data.sepal_width,
                    input_data.petal_length, input_data.petal_width]

        prediction = model.predict([features])[0]
        probability = model.predict_proba([features])[0].max()
        species = ['setosa', 'versicolor', 'virginica'][prediction]

        return jsonify({
            "prediction": species,
            "confidence": round(probability, 4)
        })
    except ValueError as e:
        return jsonify({"error": str(e)}), 400
    except Exception as e:
        logger.error(f"Error: {str(e)}")
        return jsonify({"error": "Internal server error"}), 500

@app.route('/health')
```

```
def health():
    return jsonify({"status": "healthy", "model": "loaded"})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Testing the API

Using `curl`

```
curl -X POST http://localhost:5000/predict \
  -H "Content-Type: application/json" \
  -d '{
    "sepal_length": 6.7,
    "sepal_width": 3.0,
    "petal_length": 5.2,
    "petal_width": 2.3
  }'
```

Response:

```
{
  "prediction": "virginica",
  "confidence": 0.98,
  "input": { ... }
}
```

Using Python

```python
import requests

url = "http://localhost:5000/predict"
data = {
    "sepal_length": 5.9,
    "sepal_width": 3.0,
    "petal_length": 5.1,
    "petal_width": 1.8
}

print(requests.post(url, json=data).json())
```

Deployment Strategies

Method	Use Case	Command
Local (FastAPI)	Development	`fastapi dev app.py`
Local (Flask)	Development	`python app.py`
Production (FastAPI)	High performance	`uvicorn app:app --host 0.0.0.0 --port 5000 --workers 4`
Production (Flask)	Traditional	`gunicorn -w 4 app:app`
Docker	Containerized	See Dockerfile below
Cloud	Serverless	Railway, Render, GCP Cloud Run, AWS Lambda

Dockerfile (FastAPI with Uvicorn)

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "5000", "--workers", "4"]
```

requirements.txt (FastAPI)

```
fastapi==0.104.1
uvicorn[standard]==0.24.0
scikit-learn==1.3.2
joblib==1.3.2
pydantic==2.5.0
python-multipart==0.0.6
```

requirements.txt (Flask)

```
flask==3.0.0
gunicorn==21.2.0
scikit-learn==1.3.2
joblib==1.3.2
pydantic==2.5.0
```

Advanced Features

Health Check Endpoint

```
@app.route('/health')
def health():
    return jsonify({"status": "healthy", "model": "loaded"})
```

Batch Prediction

```
# Accept list of inputs
features_list = [validate_input(item) for item in data]
predictions = model.predict(features_list)
```

Structured Logging

```
import logging
from pythonjsonlogger import jsonlogger

logger = logging.getLogger()
logHandler = logging.StreamHandler()
formatter = jsonlogger.JsonFormatter()
logHandler.setFormatter(formatter)
logger.addHandler(logHandler)
logger.setLevel(logging.INFO)

# Log prediction with context
logger.info("prediction_made", extra={
    "species": species,
    "confidence": probability,
    "input_hash": hash(tuple(features))
})
```

Business Impact

- **Faster Decision-Making**: Real-time predictions for apps.
- **Scalability**: Serve thousands of requests/second.
- **Team Collaboration**: Non-data scientists can use models via API.
- **Integration**: Connect to dashboards, mobile apps, or CRM.

Case Study: A retail company reduced fraud by 40% using a Flask-deployed XGBoost model.

Challenges

- **Cold Start**: Model loading delay on serverless platforms.

- **Input Validation**: Prevent malformed data crashes.
- **Model Drift**: Performance degrades over time.
- **Security**: Protect API with authentication (JWT, API keys).
- **Scalability**: Use load balancers for high traffic.

Best Practices (2024+)

1. **Use FastAPI for new projects** – Better performance, type safety, auto-documentation.
2. **Use Uvicorn** – Modern ASGI server, faster than Gunicorn.
3. **Implement Pydantic validation** – Type hints with automatic validation.
4. **Load model once at startup** – Use lifespan context managers.
5. **Add type hints** – Leverage Python 3.10+ features.
6. **Implement `/health` endpoint** – Essential for orchestration (Kubernetes, Docker).
7. **Use structured logging** – JSON logs for better observability.
8. **API versioning** – `/v1/predict`, `/v2/predict` for backward compatibility.
9. **Monitor latency, errors, and model drift** – Use Prometheus metrics.
10. **Containerize with Docker** – Use Python 3.12+ slim images.
11. **Add authentication** – JWT tokens or API keys for production.
12. **Document with OpenAPI** – FastAPI generates it automatically.

Conclusion

Modern frameworks like **FastAPI** and **Flask** enable data scientists to deploy machine learning models as **reliable, fast, and maintainable APIs**. FastAPI's async support, automatic validation, and auto-documentation make it the recommended choice for new projects, while Flask remains ideal for simpler use cases.

By following this guide, you've learned:

- How to structure a modern ML API (FastAPI + Pydantic)
- How to validate input with type hints and Pydantic models

- How to use Uvicorn for high-performance deployment
- How to containerize with Docker using Python 3.12
- Best practices for production ML APIs in 2024+

Now, take your own model — whether it's for churn prediction, sentiment analysis, or sales forecasting — and **deploy it today** with FastAPI and Uvicorn.

References

- Flask Documentation: flask.palletsprojects.com
- scikit-learn: scikit-learn.org
- joblib: joblib.readthedocs.io
- Gunicorn: gunicorn.org
- Docker: docker.com
