

# **Flask-based Machine Learning Model** **Deployment API**

Developed by: Mukunda

B.Tech – Artificial Intelligence & Data Science

# **1. Objective**

To design and implement a Flask-based REST API that deploys a pre-trained Machine Learning model for real-time predictions.

The project demonstrates how an ML model can be integrated into a production-style API, following clean design and deployment practices.

## **Key Highlights**

Framework: Flask (Python 3.11)

Model: Logistic Regression (trained on Iris dataset)

Functionality: Predicts flower species based on four numeric input features.

Endpoints Implemented:

/health – Check model readiness

/predict – Predict for a single input

/batch\_predict – Predict for multiple inputs with IDs

Deployment: Dockerized for easy execution and portability.

## **Outcome**

A fully functional, validated, and containerized ML API application that can accept user input, perform predictions, and return structured JSON responses.

The design follows standard production patterns such as configuration-based model loading, error handling, and authentication.

## **Scalability and Extensibility**

The architecture is designed to scale easily — new models, endpoints, or validation schemas can be added without changing the core structure. The use of configuration files, centralized error handling, and modular imports ensures future enhancements can be implemented with minimal refactoring.

# **2. System Architecture and Core Components**

## **Overview**

The application follows a clean, modular Flask architecture designed for clarity, maintainability, and scalability. Each core functionality—configuration management, model operations, API routing, input validation, authentication, and error handling—is isolated into its own module. This separation of concerns enables independent updates, easier debugging, and consistent code organization. The architecture ensures that the API remains lightweight while adhering to production-level standards such as configuration-driven initialization, centralized exception handling, and secure API key-based access. Overall, the design makes the service easy to extend, test, and deploy across different environments.

## Component Summary

Component / Folder	Description
config/	Contains settings.yaml used to store API key, model path, and CORS settings.
app/model.py	Loads the pre-trained Iris model and provides methods for single and batch predictions.
app/routes.py	Defines REST endpoints (/health, /predict, /batch_predict).
app/schemas.py	Defines Pydantic models for input validation and type checking.
app/auth.py	Implements API key-based authentication for request protection.
app/errors.py	Centralized exception handling and structured logging for debugging and API stability.
Dockerfile	Defines containerization steps to build and run the API in an isolated environment.

## Application Flow

1. **Client sends a request** → /predict or /batch\_predict.
2. **Request validated** by schemas.py using Pydantic models.
3. **Authentication** handled by auth.py (API key validation).
4. **Model prediction** executed through model.py.
5. **Response** returned as structured JSON with prediction label and probability.

```
[Client Request]
  ↓
[Flask Routes (/predict, /batch_predict)]
  ↓
[Validation + Auth Layer]
  ↓
[Iris Model (Logistic Regression)]
  ↓
[JSON Response with Prediction]
```

### **3. Iris Classification Model (Logistic Regression)**

#### **Dataset Used**

**Dataset Name:** Iris Dataset (UCI Machine Learning Repository)

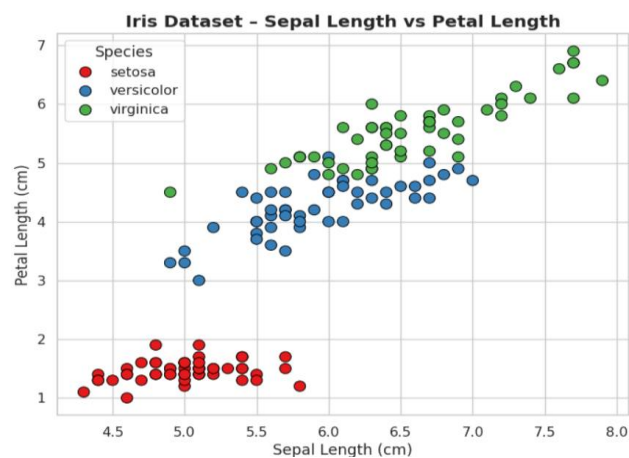
**Samples:** 150 flower observations across 3 species

#### **Features:**

- Sepal Length
- Sepal Width
- Petal Length
- Petal Width

#### **Target Classes:**

- Setosa
- Versicolor
- Virginica



#### **Algorithm Chosen**

- **Model Type:** Logistic Regression (multi-class classification using scikit-learn).
- **Reason for Selection:**
  - Simple yet effective for small structured data.
  - Interpretable model with stable performance.
  - Fast to train and lightweight for deployment.

## Training Details

- **Split:** 80% training / 20% testing.
- **Tool Used:** scikit-learn.
- **Accuracy Achieved:** ~96% on test data.
- **Evaluation Metrics:** Accuracy and confusion matrix.
- **Model Artifact:** Saved as iris\_logreg.joblib under the models/ folder.

## Integration with Flask

Model is loaded **once during application startup** (as per config/settings.yaml).

Flask routes call model.predict\_one() or model.predict\_many() for inference.

Predictions include both the **class label** and the **confidence probability**.

## 4. Flask API Endpoints and Input Validation

### Overview

The application exposes **RESTful endpoints** for health checks and predictions. Each endpoint is secured with an API key and validated using **Pydantic models** before processing requests.

### Endpoints Summary

Endpoint	Method	Description	Authentication	Example Response
/health	GET	Checks service and model readiness	Not Required	{ "status": "ok", "model": "logreg-iris", "model_version": "1.0" }
/predict	POST	Predicts class for a single input record	Required	{ "id": "r1", "label": "setosa", "probability": 0.9784 }
/batch_predict	POST	Predicts for multiple inputs (list) with request IDs	Required	{ "results": [ { "id": "r1", ... }, { "id": "r2", ... } ] }

## Input Validation (via Pydantic)

- Each incoming request body is parsed and validated before reaching the model.
- Validation ensures:
  - Required fields are present.
  - All features are numeric.
  - The feature list has **exactly four values**.
- Invalid inputs trigger structured **400 Bad Request** responses with detailed error messages.

## Authentication

- Implemented using a custom decorator (`require_api_key`) in `app/auth.py`.
- Each prediction request must include the header:  
**x-api-key: CHANGE\_ME**
- If the key is missing or incorrect, the API returns a **401 Unauthorized** error.

## Error Handling

- Centralized in `app/errors.py`.
- Handles invalid requests, missing data, and internal exceptions.
- Returns consistent JSON error responses such as:

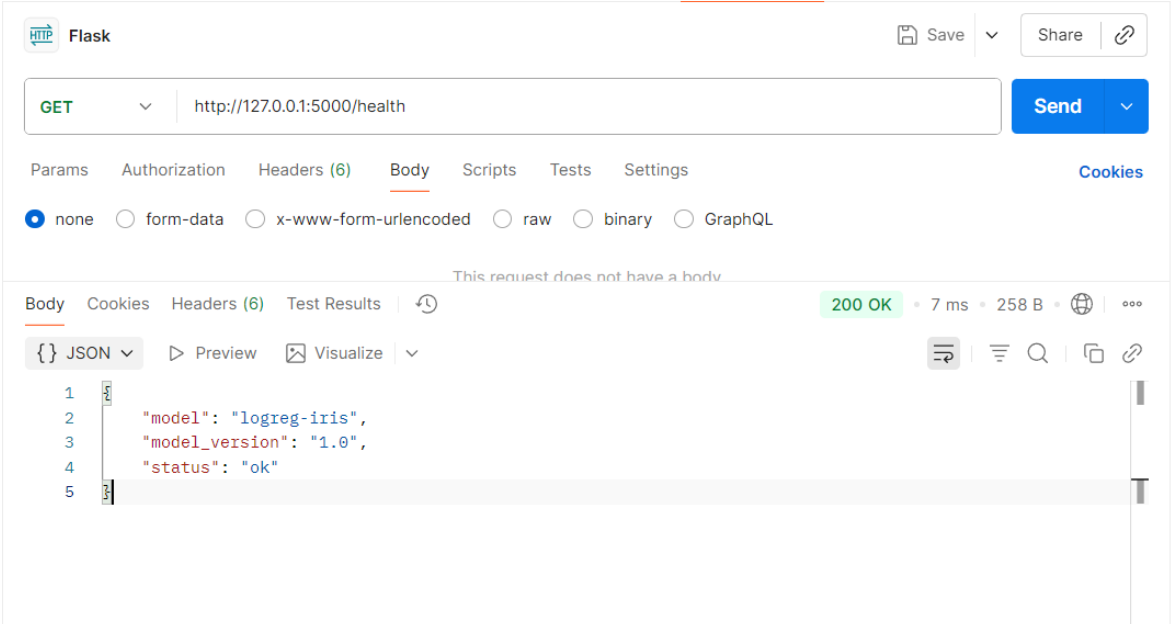
```
Example 400 (invalid features length)
{
  "error": "Invalid input",
  "code": 400,
  "details": "features must contain exactly 4 numeric values"
}
```

## 5. Demo Results (Postman, Tests, Docker)

Below are the actual outputs captured while testing the deployed Flask ML API:

- 5.1 Postman – /health (GET, 200) → Confirms model readiness and version.

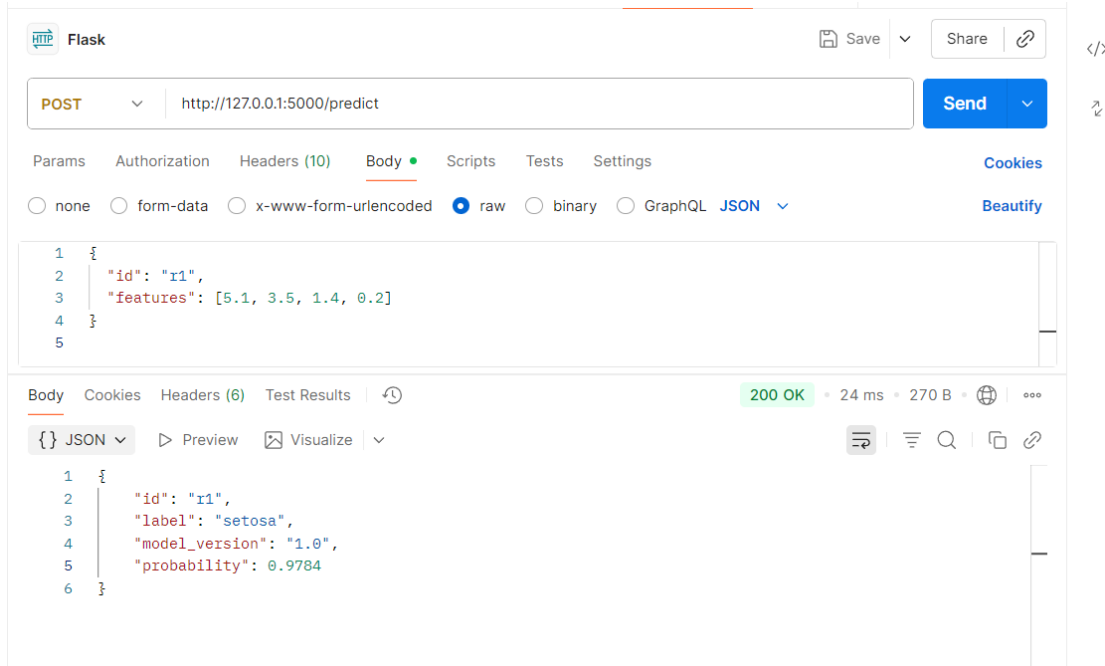
```
(C:\AI Intern\flask-ml-api\flaskml) C:\AI Intern\flask-ml-api>curl http://127.0.0.1:5000/health
{"model": "logreg-iris", "model_version": "1.0", "status": "ok"}
```



The screenshot shows a Postman client interface for a GET request to `http://127.0.0.1:5000/health`. The response is a 200 OK status with a response time of 7 ms and a body size of 258 B. The response body is displayed in JSON format: `{ "model": "logreg-iris", "model_version": "1.0", "status": "ok" }`.

- 5.2 Postman – /predict (POST, 200) → Returns predicted class for a single input.

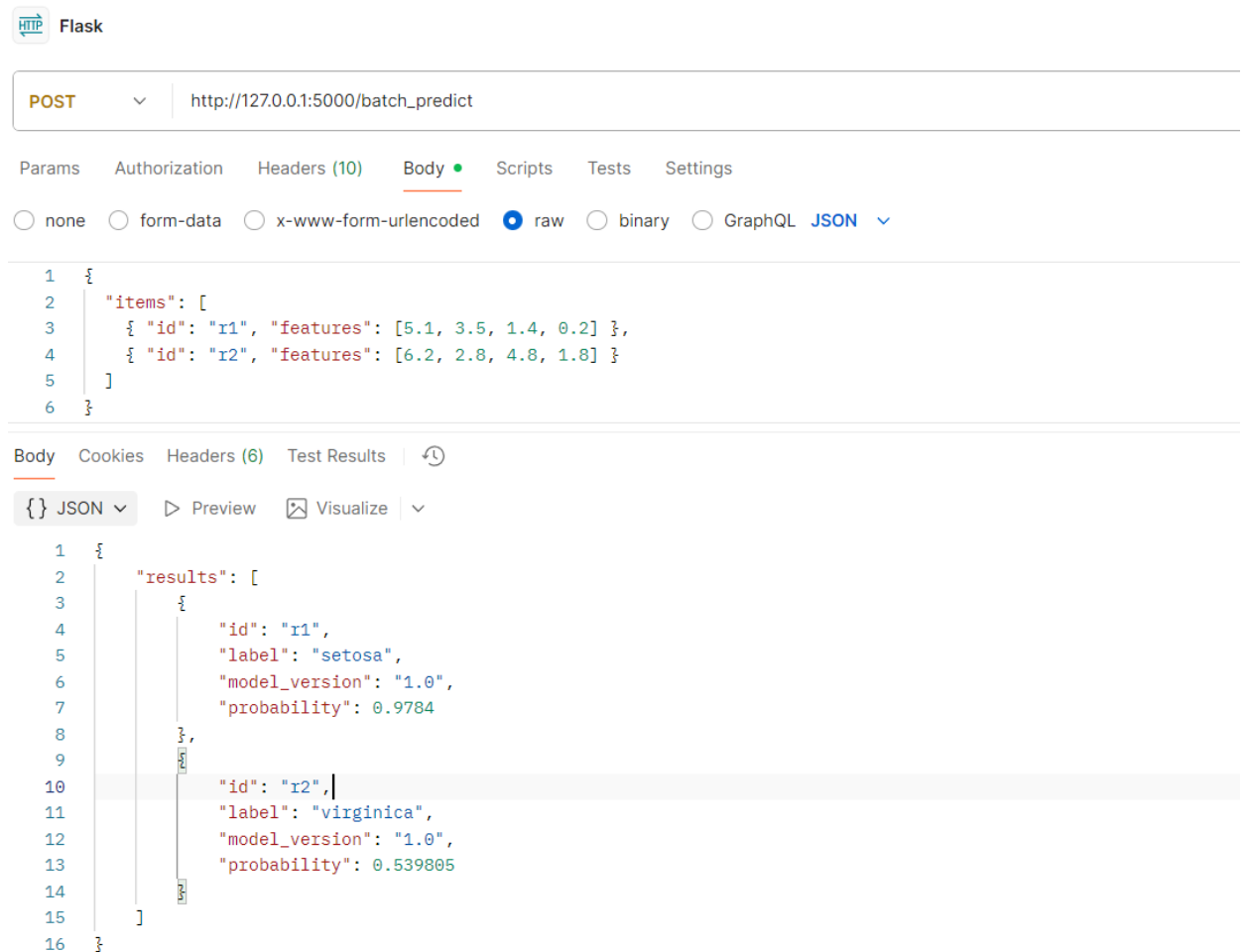
```
(C:\AI Intern\flask-ml-api\flaskml) C:\AI Intern\flask-ml-api>curl -X POST http://127.0.0.1:5000/predict ^
More? -H "Content-Type: application/json" ^
More? -H "x-api-key: CHANGE_ME" ^
More? -d {"id": "r1", "features": [5.1, 3.5, 1.4, 0.2]}
{"id": "r1", "label": "setosa", "model_version": "1.0", "probability": 0.9784}
```



The screenshot shows a Postman client interface for a POST request to `http://127.0.0.1:5000/predict`. The request body is a JSON object: `{ "id": "r1", "features": [5.1, 3.5, 1.4, 0.2] }`. The response is a 200 OK status with a response time of 24 ms and a body size of 270 B. The response body is displayed in JSON format: `{ "id": "r1", "label": "setosa", "model_version": "1.0", "probability": 0.9784 }`.

- 5.3 Postman – /batch\_predict (POST, 200)→ Returns predictions for multiple inputs with request IDs.

```
(C:\AI Intern\flask-ml-api\flaskml) C:\AI Intern\flask-ml-api>curl -X POST http://127.0.0.1:5000/batch_predict -H "Content-Type: application/json" -H "x-api-key: CHANGE_ME" -d '{"items":[{"id":"r1","features":[5.1,3.5,1.4,0.2]},{"id":"r2","features":[6.2,2.8,4.8,1.8]}]}'
{"results":[{"id":"r1","label":"setosa","model_version":"1.0","probability":0.9784},{"id":"r2","label":"virginica","model_version":"1.0","probability":0.539805}]}
```



The screenshot shows the Postman interface for a POST request to `http://127.0.0.1:5000/batch_predict`. The request body is a JSON object with two items. The response body is a JSON object with two results, each containing an id, label, model\_version, and probability.

**Request Body:**

```
{
  "items": [
    {
      "id": "r1",
      "features": [5.1, 3.5, 1.4, 0.2]
    },
    {
      "id": "r2",
      "features": [6.2, 2.8, 4.8, 1.8]
    }
  ]
}
```

**Response Body:**

```
{
  "results": [
    {
      "id": "r1",
      "label": "setosa",
      "model_version": "1.0",
      "probability": 0.9784
    },
    {
      "id": "r2",
      "label": "virginica",
      "model_version": "1.0",
      "probability": 0.539805
    }
  ]
}
```

- 5.4 Tests – pytest -q → All tests passed

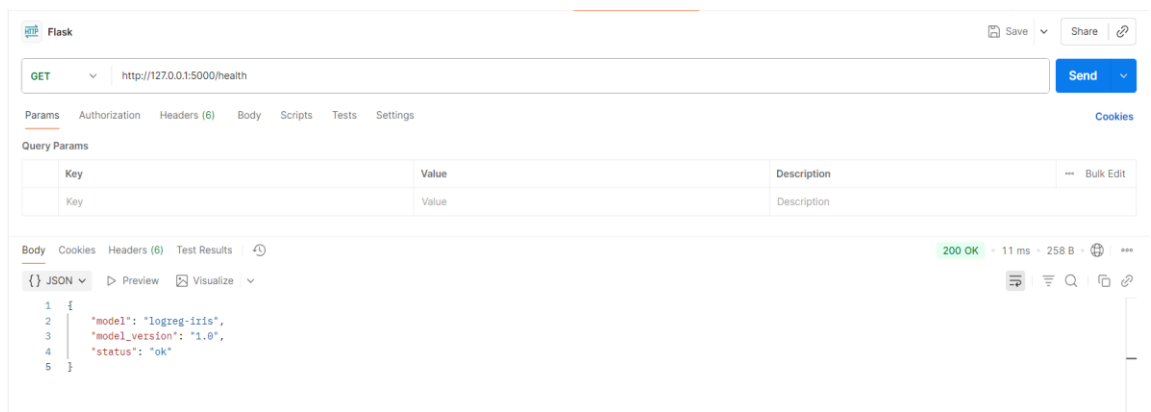
```
(C:\AI Intern\flask-ml-api\flaskml) C:\AI Intern\flask-ml-api>pytest -q
..... [100%]
5 passed in 1.39s
```



- 5.5 Docker – image build + container /health

Docker image successfully built and container verified by testing /health endpoint.

```
(C:\AI Intern\flask-ml-api\flaskml) C:\AI Intern\flask-ml-api>docker build -t flask-ml-api .
[+] Building 11.7s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 353B
=> [internal] load metadata for docker.io/library/python:3.11
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/5] FROM docker.io/library/python:3.11@sha256:c549b348bb8b6518df6dec2e356381a8a5d9333a6a3979de79b714a18a02d255
=> => resolve docker.io/library/python:3.11@sha256:c549b348bb8b6518df6dec2e356381a8a5d9333a6a3979de79b714a18a02d255
=> [internal] load build context
=> => transferring context: 1.80MB
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY requirements.txt .
=> CACHED [4/5] RUN pip install -r requirements.txt
=> CACHED [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:1749b4ea993b7cb3a19a2097b5873103fcaba19d6c7c1bd73dc5a325926805b
=> => exporting config sha256:fd11cc7acc7dd3f3ee7f043dc4c08979714c0b27ac23f0b3ef9d91717098b6385
=> => exporting attestation manifest sha256:d8802348691fa07c8a6a0a390b6cef6d155b29018f317be0efcd75ce3e5657e3a
=> => exporting manifest list sha256:a802e84f9b3428c298405529cfe6ff072c99b403935c54548f22c571a5c2561b
=> => naming to docker.io/library/flask-ml-api:latest
=> => unpacking to docker.io/library/flask-ml-api:latest
```



## 6. How to Run the Application

### Run Locally (Using Conda or Python)

1. Activate your environment:  
`conda activate "C:\\AI Intern\\flask-ml-api\\flaskml"`
2. Navigate to the project directory:  
`cd "C:\\AI Intern\\flask-ml-api"`
3. Start the Flask application:  
`python -m app`
4. Open a browser or Postman and verify:  
<http://127.0.0.1:5000/health>

### Run Using Docker

1. Build the Docker image:  
`docker build -t flask-ml-api .`
2. Run the container:  
`docker run -p 5000:5000 flask-ml-api`
3. Verify in browser or Postman:  
`http://127.0.0.1:5000/health`

## **7. Conclusion**

- Delivered a **Flask-based REST API** that loads a **pre-trained Iris Logistic Regression model** at startup.
- Exposed **/health**, **/predict**, **/batch\_predict** with **Pydantic validation**, **API key auth**, **CORS**, and **centralized error handling**.
- Provided **unit tests** and **Docker** support to simplify verification and deployment.
- The implementation is modular, maintainable, and ready for integration.