Industrial Project 2026-04-16

End-to-End YOLO Key Detection System

End-to-end key detection using YOLO, FastAPI, Docker, Hugging Face Spaces, and Azure Container Apps for production-ready inference deployment.

No cover image available.

Project Snapshot

DATE
2026-04-16

CATEGORY
Industrial Project

SHORT DESCRIPTION
End-to-end key detection using YOLO, FastAPI, Docker, Hugging Face Spaces, and Azure Container Apps for production-ready inference deployment.

TAGS / TECH STACK
Computer Vision MLOps

Project Links

Hugging Face Space Project Link Project Link Project Link Demo Video GitHub Repository

Full Project Content

> Building an AI system that doesn't just **detect keys** — but packages training, inference, APIs, and cloud deployment into a production-oriented computer vision workflow.

GitHub Repository: [dranubhaparashar/End-to-End-YOLO-Key-Detection-System-Training-API-Deployment-and-Azure-Container-Apps]

> Live demo: [Hugging Face Space]

Vision

Most computer vision projects stop at model training.

This project focuses on the full operational chain required to make a detection model usable in practice:

- Prepare and organize the dataset
- Train a YOLO-based detector
- Package the trained model for inference
- Expose predictions through a REST API
- Containerize the service with Docker
- Deploy it to Azure Container Apps
- Provide a reusable deployment runbook for scale-out operations

---

Project Attributes

Attribute	Description
`problem-statement`	Key detection models are often trained in isolation without a deployable, reproducible inference workflow.
`primary-objective`	Build an end-to-end key detection pipeline that connects model training, inference serving, and production deployment.
`core-technologies`	YOLO, FastAPI, Docker, Azure Container Apps, Azure Container Registry, Hugging Face Spaces.
`repository-scope`	Dataset assets, training outputs, API implementation, containerization artifacts, and deployment documentation.

Attribute	Description
`runtime-interface`	REST-based inference service with health check and prediction endpoints.
`deployment-target`	Azure Container Apps for production-style hosting of the inference service.
`demo-surface`	Public Hugging Face Space for quick validation and showcase.
`key-capabilities`	Key detection, image upload inference, JSON detection output, containerized deployment, and operational reproducibility.
`security-controls`	API-key-based request validation in the prediction endpoint and environment-driven runtime configuration.
`production-focus`	Container-first packaging, parameterized runtime config, cloud deployment flow, and runbook-oriented documentation.

Post Files

```
src/content/posts/
└── end-to-end-yolo-key-detection/
    ├── cover.png
    └── index.mdx

GitHub Repo: dranubhaparashar/End-to-End-YOLO-Key-Detection-System-Training-API-Deployment-and-Azure-Container-Apps/
├── dataset/
│   └── EV_Keys_2/
├── gis-key-detection-func/
│   ├── app.py
│   ├── Dockerfile
│   ├── requirements.txt
│   └── best.pt
├── runs/
│   └── detect/
│       └── train14/
└── README.md
```

Why This Matters

Note A computer vision model becomes useful only when it is **servable, testable, and deployable**.

Important This project covers the missing middle between **training notebook outputs** and **production-ready inference delivery**.

Tip The same structure can be reused for other object detection use cases with minimal changes to the dataset and model artifacts.

Warning Model quality alone is not sufficient — deployment, runtime packaging, security, and observability matter just as much.

Caution Any public demo should avoid exposing production secrets, internal endpoints, or unrestricted inference access.

What This System Does

```
flowchart TD
    A[Dataset: EV_Keys_2] --> B[YOLO Training Pipeline]
    B --> C[Training Outputs: runs/detect/train14]
    C --> D[Model Artifact: best.pt]

    D --> E[FastAPI Inference Service]
    E --> F[Docker Image Build]
    F --> G[Azure Container Registry]
    G --> H[Azure Container Apps]

    I[Client / Consumer] --> J[POST /predict]
    H --> J
    J --> E
    E --> K[YOLO Inference]
    K --> L[JSON Response: classes, confidence, bbox_xyxy]

    M[GET /health] --> H
    H --> N[Runtime Status, Model Path, Threshold]
```

Runtime Request Flow

```
sequenceDiagram
    autonumber
    participant U as User / Client
    participant API as FastAPI Service
    participant M as YOLO Model
    participant IMG as Uploaded Image

    U->>API: POST /predict + file + x-api-key
    API->>IMG: Read multipart image payload
    API->>M: Run predict(image, conf=CONF_THRESHOLD)
    M-->>API: Detection boxes, classes, confidence
    API-->>U: JSON response with detections
```

Core Capabilities

- YOLO-based object detection for keys
- FastAPI-based REST inference service
- Health endpoint for runtime verification
- API-key protected prediction endpoint
- Dockerized inference packaging
- Cloud deployment to Azure Container Apps
- Public demo surface through Hugging Face Spaces
- Reusable technical documentation and runbook structure

Bento Overview

Capability	Description
Dataset Layer	Stores labeled key images and YOLO training inputs
Training Layer	Produces trained weights and evaluation outputs
Inference Layer	Loads `best.pt` and exposes predictions via FastAPI
Runtime Layer	Uses environment variables for model path, API key, and threshold
Container Layer	Packages the API into a reproducible Docker image
Cloud Layer	Hosts the service on Azure Container Apps
Demo Layer	Exposes a public-facing validation surface through Hugging Face

---

Detection and Serving Stack

Model Details

- Model family: YOLO
- Task: object detection
- Input type: uploaded RGB image
- Output type: detection list with class, class ID, confidence, and bounding box coordinates

API Details

- `GET /` → service status message
- `GET /health` → runtime health and model metadata
- `POST /predict` → inference endpoint for uploaded images

Runtime Configuration

- `MODEL_PATH`
- `API_KEY`
- `CONF_THRESHOLD`

Inference Service

```
from fastapi import FastAPI, UploadFile, File, Header, HTTPException
from ultralytics import YOLO
from PIL import Image
from io import BytesIO
from pathlib import Path
import os

app = FastAPI(title="YOLO Key Detection API")

MODEL_PATH = os.getenv("MODEL_PATH", "/app/best.pt")
API_KEY = os.getenv("API_KEY", "change-this-key")
CONF_THRESHOLD = float(os.getenv("CONF_THRESHOLD", "0.25"))

_model = None

def get_model() -> YOLO:
    global _model
    if _model is not None:
        return _model

    model_file = Path(MODEL_PATH)
```

```
    if not model_file.exists():
        raise RuntimeError(f"Model file not found: {model_file}")

    _model = YOLO(str(model_file))
    return _model

@app.get("/health")
def health():
    model_file = Path(MODEL_PATH)
    return {
        "status": "ok",
        "model_path": str(model_file),
        "model_exists": model_file.exists(),
        "confidence_threshold": CONF_THRESHOLD,
    }

@app.post("/predict")
async def predict(
    file: UploadFile = File(...),
    x_api_key: str | None = Header(default=None),
):
    if API_KEY and x_api_key != API_KEY:
        raise HTTPException(status_code=401, detail="Invalid API key")

    image = Image.open(BytesIO(await file.read())).convert("RGB")
    model = get_model()
    results = model.predict(image, conf=CONF_THRESHOLD, verbose=False)
    result = results[0]

    detections = []
    if result.boxes is not None:
        names = result.names
        for box in result.boxes:
            cls_id = int(box.cls.item())
            conf = float(box.conf.item())
            x1, y1, x2, y2 = [float(v) for v in box.xyxy[0].tolist()]
            detections.append({
                "class": names.get(cls_id, str(cls_id)) if isinstance(names, dict) else
names[cls_id],
                "class_id": cls_id,
                "confidence": conf,
                "bbox_xyxy": [x1, y1, x2, y2],
            })

    return {
        "filename": file.filename,
        "image_width": image.width,
```

```
        "image_height": image.height,
        "detections": detections,
    }
```

Dependency Stack

```
fastapi==0.115.12
uvicorn[standard]==0.34.0
python-multipart==0.0.20
pillow==11.2.1
ultralytics==8.3.146
```

Containerization Layer

```
FROM python:3.11-slim

ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PORT=8000 \
    MODEL_PATH=/app/best.pt \
    API_KEY=change-this-key \
    CONF_THRESHOLD=0.25

WORKDIR /app

RUN apt-get update && apt-get install -y --no-install-recommends \
    libgl1 libglib2.0-0 && \
    rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir --index-url https://download.pytorch.org/whl/cpu torch
torchvision && \
    pip install --no-cache-dir -r requirements.txt

COPY app.py .
COPY best.pt .

EXPOSE 8000
CMD ["sh", "-c", "uvicorn app:app --host 0.0.0.0 --port ${PORT}"]
```

Deployment Path

```
flowchart LR
    A[Local Dev / Training Outputs] --> B[Inference Service Code]
    B --> C[Docker Build]
    C --> D[Container Image]
    D --> E[Azure Container Registry]
    E --> F[Azure Container Apps]
    F --> G[External HTTPS Endpoint]
    G --> H[Health Check / Predict Requests]
```

Runtime Example

```
# local virtual environment
python -m venv .venv
source .venv/bin/activate

# install dependencies
pip install -r requirements.txt

# run API locally
uvicorn app:app --host 0.0.0.0 --port 8000

# health check
curl http://localhost:8000/health

# prediction request
curl -X POST "http://localhost:8000/predict" \
  -H "x-api-key: change-this-key" \
  -F "file=@sample_key.jpg"
```

Azure Deployment Pattern

- Build a Docker image for the inference service
- Push the image to Azure Container Registry
- Deploy or update the Azure Container App
- Expose external ingress on the inference port
- Validate runtime through `/health`
- Execute inference through `/predict`

```
az login
az acr login --name <ACR_NAME>

docker build -t gis-key-detection:latest .
```

```
docker tag gis-key-detection:latest <ACR_NAME>.azurecr.io/gis-key-detection:latest
docker push <ACR_NAME>.azurecr.io/gis-key-detection:latest

az containerapp create \
  --name <CONTAINER_APP_NAME> \
  --resource-group <RESOURCE_GROUP> \
  --environment <CONTAINER_APPS_ENV> \
  --image <ACR_NAME>.azurecr.io/gis-key-detection:latest \
  --target-port 8000 \
  --ingress external \
  --registry-server <ACR_NAME>.azurecr.io
```

Public Demo Surface

> **Important** A public Hugging Face Space provides a lightweight demo surface while the primary production deployment pattern remains Azure Container Apps.

Demo Link

- AnubhaParashar / Key-detection

Demo Video

Embedded media: Open video

Engineering Value

This project is not just about detecting a visual object.

It demonstrates how to take a trained model and make it:

- reproducible
- containerized
- callable through a stable API
- deployable in cloud infrastructure
- portable across demo and production surfaces
- documented for reuse by other engineers

Current Strengths

- Clear separation between training artifacts and inference service
- Simple API contract for health and prediction

- Parameterized runtime settings
- CPU-compatible deployment path
- Public demo and cloud deployment story in the same project
- Technical runbook orientation rather than notebook-only experimentation

---

Next Improvements

- Add formal evaluation metrics and benchmark summary
- Introduce CI/CD for build and deployment automation
- Add model versioning and artifact registry integration
- Add structured logging and request tracing
- Add batch inference mode
- Add request/response schema docs with OpenAPI examples
- Move secrets to managed secret storage for production environments
- Add load and concurrency validation for Azure-hosted runtime

---

Key Innovation

> **Important** This project connects **computer vision model development** with **API engineering** and **cloud deployment discipline**.

It turns:

Model Training → Inference Service → Containerized Deployment

rather than stopping at experimentation.

Conclusion

This repository shows how an object detection solution can evolve from a YOLO training workflow into a deployable engineering asset.

It combines:

- dataset organization
- training outputs
- FastAPI inference serving
- Docker packaging
- Azure Container Apps deployment
- public demo exposure
- technical runbook documentation

Final Thought

> From **training a key detector**
> to **shipping a cloud-ready computer vision service**

> The real value is not only the model — it is the **end-to-end delivery architecture**.

Source: src/content/posts/End-to-End-YOLO-Key-Detection-System-Training-API-Deployment-and-Azure-Container-Apps.md