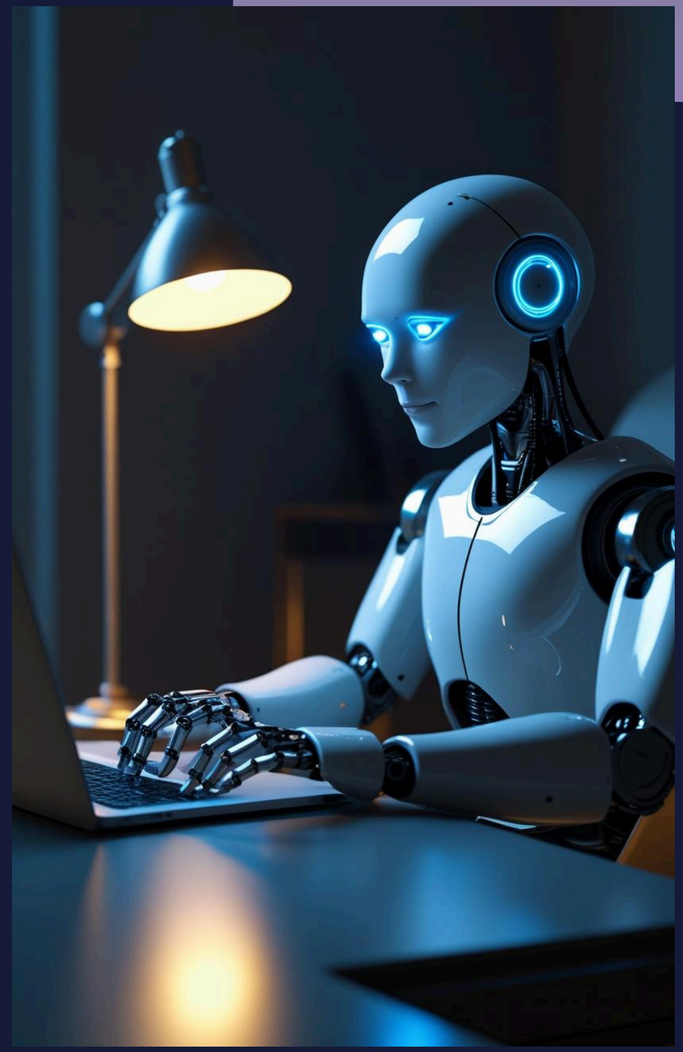
MLOps u praksi: Primena, serviranje i održavanje modela

Fashion MNIST klasifikacija - end-to-end MLOps pipeline



Marko Mrđa E9 4/2023
31.07.2025.

Agenda i kontekst

1. Fashion MNIST klasifikacija sa kompletnim MLOps workflow-om
2. Primena modela, serviranje i monitoring, održavanje modela
3. Demonstracija teorije kroz praktičnu implementaciju

Primena modela

- Primena modela je proces stavljanja obučenog modela u produkciju
- Most između eksperimentalne faze i stvarnog sveta

Delovi arhitekture primene modela

- Cloud vs Edge computing
- Statička vs Dinamička primena
- REST API vs gRPC
- Online vs Batch vs Streaming

Implementacija

```python
@app.route('/predict', methods=['POST'])
def predict():
    client_ip = request.remote_addr
    logger.info(f"Prediction request from {client_ip}")

    try:
        if 'file' not in request.files:
            logger.warning(f"No file in request from {client_ip}")
            return jsonify({'error': 'No file provided'}), 400

        file = request.files['file']
        logger.debug(f"Processing file: {file.filename}")

        img = Image.open(file).convert('RGB')
        transform = get_request_data_transformations()
        img = transform(img).unsqueeze(0)

        with torch.no_grad():
            output = model(img)
            _, predicted = torch.max(output, 1)

        output_lbl = output_label(predicted)

        logger.info(f"Successful prediction: {output_lbl}")
        return jsonify({'predicted_class': output_lbl})

    except Exception as e:
        logger.error(f"Prediction error: {str(e)}", exc_info=True)
        return jsonify({'error': 'Internal server error'}), 500
```

Primene modela – zaključci

1. REST API - odličan izbor, jednostavan za integraciju
2. Docker kontejnerizacija omogućila konzistentno okruženje

Serviranje i monitoring

- Runtime okruženje za primenu modela na stvarne podatke
- Kontinuirano praćenje performansi u produkciji
- Ključna svojstava: bezbednost, jednostavnost primene, garancija valjanosti, lakoća oporavka

Izazovi u stvarnom svetu

- Greške u sistemu: Neizbežne, moraju se prihvatiti i upravljati njima
- Promene u podacima: Performanse se menjaju tokom vremena
- Ljudska priroda: Nepredvidivi, iracionalni korisnici

Garancija valjanosti modela

- Automatska evaluacija svakog novog modela
- Samo bolji modeli zamenjuju postojeće
- Čuvanje svih verzija u Azure Blob Storage za rollback

Implementacija

```python
# Rate limiting
limiter = Limiter(app=app, default_limits=["200 per day", "50 per hour"])

# IP tracking u svakom endpoint-u
client_ip = request.remote_addr
logger.info(f"Request from {client_ip}")

# Hot-swapping modela
@app.route('/reload_model', methods=['POST'])
@limiter.limit("1 per 5 minutes")
def reload_model():
    logger.info("Model reload requested")
    try:
        global model
        model = load_model()
        logger.info("Model reloaded successfully")
        return jsonify({'message': 'Model reloaded successfully'})
    except Exception as e:
        logger.error(f"Model reload failed: {str(e)}")
        return jsonify({'error': 'Model reload failed'}), 500

# Fallback strategija
try:
    model = get_best_model()  # Azure
    logger.info("Model loaded from Azure Blob Storage")
except Exception as e:
    logger.error(f"Azure loading failed: {e}")
    logger.warning("Falling back to local storage...")
    model = torch.load(local_path)
    logger.info(f"Model loaded locally from: {local_path}")
```

Serviranja i monitoringa modela – zaključci

1. Fallback na lokalno skladištenje povećava pouzdanost u slučaju greške skadišta
2. Izazov: Balansiranje između bezbednosti i pristupačnosti
3. IP tracking i centralizovani logger poboljšao bezbednosno praćenje API-ja

Održavanje modela

- Kontinuiran proces ažuriranja nakon produkcije
- Dva ključna pitanja: KADA i KAKO ažurirati
- Faktori za odlučivanje: frekvencija grešaka, dostupnost podataka, vreme treninga, troškovi, vrednost za korisnike

Šabloni primene

- Kompleksni sistemi zahtevaju sofisticiranu infrastrukturu
- Schedulers, Orchestrators, Storage, Compute
- Cloud vs On-premises trade-off-ovi

Prefect workflow

```python
@flow(name="fashion_mnist_training", retries=1)
def training_pipeline() -> Tuple[float, float]:
    logger = get_run_logger()
    logger.info("Starting training pipeline")

    try:
        device = setup_device()
        train_loader, test_loader = load_training_data()
        best_params = optimize_hyperparameters(train_loader, test_loader, device)
        model, optimizer = initialize_model(best_params, device)
        trained_model = train(model, train_loader, optimizer, device)
        val_loss, val_accuracy = evaluate(trained_model, test_loader, device)
        save_model(trained_model, val_accuracy)

        logger.info("Pipeline completed successfully")
        return val_loss, val_accuracy

    except Exception as e:
        logger.error(f"Pipeline failed: {str(e)}")
        raise


training_pipeline.serve(
    name="daily-fashion-mnist-training",
    cron="0 2 * * *"  # Runs at 2 AM UTC daily
)
```

Infrastruktura

docker-compose.yml

- Postgres (Prefect database)
- Prefect-server (Orchestration)
- App (Flask API + Worker + Training)

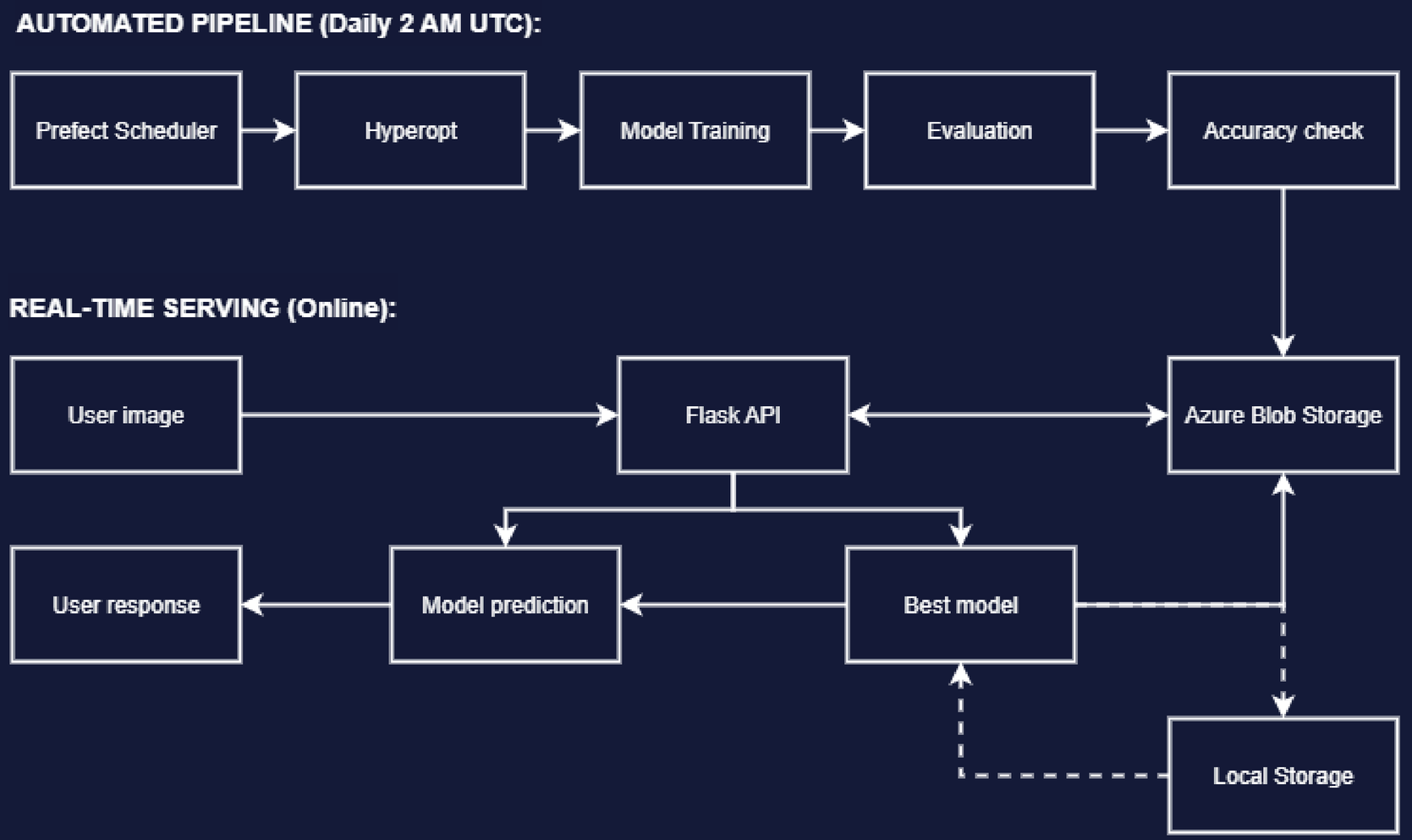
supervisord.conf

- Flask API server
- Prefect worker
- Training pipeline deployment

Održavanje modela – zaključci

1. Prefect omogućava robusnu orkestraciju sa retry logikom
2. Dnevno retreniranje osigurava svežinu modela

End-to-end workflow



Hvala na pažnji!