MLOps Project

Mathias Goris & Arthur Hofman

## GitHub Repository

<https://github.com/MathiasGoris2440/MLOps-Project>

## Explanation of our project

This project builds upon a machine learning case we explored during a previous course. The context involves a luxury hotel in the Bahamas screening **Smurf guests** for admission. Smurfs are notorious for generating both high profits and significant property damages during their stays.

### Data context

The original dataset includes records for 5000 past Smurf guests. For each guest, the dataset contains:

* **Profit made** during their last stay (outcome\_profit)
* **Whether damage was caused** (outcome\_damage\_inc)
* **Damage cost** (outcome\_damage\_amount)
* A variety of features like previous stay history, hotel facility usage, demographic data, and staff behavior scores

### Our approach

To simplify the project and improve focus on the model and deployment pipeline, we used a **cleaned version** of the dataset.

* **Model Used:** GradientBoostingRegressor from scikit-learn
* **Reason for Choice:** This model yielded the best performance in our earlier analysis of this dataset. We reused the best hyperparameters identified during that project.
* **Preprocessing:** The cleaned dataset already handled all of the missing and anomalous data. We ensured correct feature scaling and converted categorical variables where necessary before training.

## Task 1 – Cloud AI Services (Azure Machine Learning)

To train and register our machine learning model in the cloud, we used **Azure Machine Learning** services.

### Resource Setup

We first created a dedicated Azure resource group and workspace:

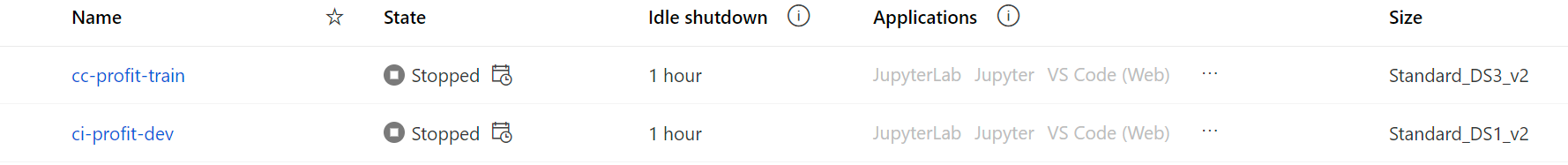
* **Resource Group:** smurf-profit-ml-gr
* **Workspace:** profit-model-ws

A white rectangular object with black lines

AI-generated content may be incorrect.

Within this workspace, we provisioned two compute resources:

* cc-profit-train: Dedicated for running Azure ML pipelines and training jobs
* ci-profit-dev: Used for development and experimentation with Jupyter notebooks



### Data Assets

We uploaded our **cleaned dataset** as a registered data asset inside the Azure Machine Learning workspace. This dataset serves as the input for the training pipeline.

A screenshot of a computer

AI-generated content may be incorrect.

### Development Workflow

For a more flexible development experience, we transitioned to **Visual Studio Code** using the **Azure CLI**. This allowed us to define and control the training pipeline programmatically.

Key configuration and training files:

* **smurf-train-env.yaml** and **conda.yaml**: Define the Python environment for training (dependencies, versions)
* **train\_regressor.yaml**: Defines the training component, including inputs, outputs, and environment
* **train.py**: Contains the training logic using GradientBoostingRegressor
* **pipeline.yaml**: Defines the pipeline which runs train.py and registers the trained model as an Azure ML model artifact

Using these YAML and Python files, we were able to submit the pipeline directly from the command line via the Azure CLI, enabling automated model training and registration. This process is illustrated in the image below.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

### GitHub Integration

To enable GitHub Actions to interact with our Azure ML workspace (for automated training and deployment), we created an App Registration:

* **App Registration Name:** github-ml-pipeline

This registration provides secure credentials that allow GitHub to authenticate against Azure for running ML workflows from CI/CD pipelines.

A screenshot of a computer

AI-generated content may be incorrect.

## Task 2 Deploy the trained model using Kubernetes, using appropriate microservices

This document details the implementation and deployment of a machine learning model using a microservices architecture on a Kubernetes cluster. The primary goal was to create a robust, scalable, and maintainable system, separating concerns into distinct services: a FastAPI API for model inference, a basic web frontend for user interaction, and an NGINX server acting as a dedicated reverse proxy for the frontend and internal API routing.

### Overall Architecture and Communication Flow

The solution is designed with a clear separation of concerns, where each component (frontend, backend API, and a dedicated NGINX proxy) operates as a distinct microservice. All these microservices are deployed within a local Kubernetes cluster managed by k3d.

The communication flow for a user interacting with the application is as follows:

1. **User Request (Browser):** The user accesses the application via their web browser at <http://localhost/>.
2. **External Ingress (Traefik):** This request first hits the Ingress Controller (Traefik, bundled with k3d), which is exposed on the host's port 80 via k3d's port mapping.
   1. The Ingress, configured via *k8s/ingress.yaml*, acts as the cluster's main entry point and routes traffic based on URL paths.
   2. Requests to the root path (/) are routed to the Frontend Service.
   3. Requests to */api/predict* or */api/docs* are routed to the FastAPI Service.
3. **Frontend Service (NGINX Pod):** If the request is for the frontend, it reaches the smurf-frontend-service, which directs it to an NGINX pod.
   1. NGINX serves the static HTML, CSS, and JavaScript files to the user's browser.
4. **Frontend API Call (Browser to NGINX Proxy):** Once the frontend is loaded in the browser, when the user submits the form, the JavaScript sends a POST request to /api/predict. This request goes back through the browser, hits the Ingress (Traefik) again, and is routed directly to the FastAPI Service as defined by the Ingress rules.
   1. *(Note: While NGINX can also proxy API calls, in our current setup, the Ingress directly routes /api calls to FastAPI, bypassing the frontend's NGINX pod for API specific paths. This is a valid configuration, effectively making Traefik the primary API gateway for external requests.)*
5. **FastAPI Service (Backend Pod):** The API request reaches the smurf-fastapi-service, which load-balances it to an available FastAPI pod.
6. **Model Inference:** Inside the FastAPI pod, the application loads the pre-trained model.pkl and performs the prediction based on the input data.
7. **Response:** The prediction result is sent back through the FastAPI Service, Ingress, and finally to the user's browser, where the frontend displays it.

A screenshot of a computer program

AI-generated content may be incorrect.

### FastAPI API Endpoint

The FastAPI application serves as the core backend, exposing an API endpoint for model inference.

* **Purpose:** To encapsulate the machine learning model and provide a standardized HTTP interface for predictions. This allows the model to be consumed by various clients (like our frontend) without them needing to understand the underlying model logic or dependencies.
* **Implementation:**
  + The *inference/main.py* file defines the FastAPI application.
  + The model (*model.pkl*) is loaded directly into the FastAPI application's memory when the application starts, ensuring efficient inference during requests.
  + A *POST* endpoint, */api/predict*, is defined, which expects a JSON payload matching the *SmurfFeatures* Pydantic schema. Upon receiving data, it performs the prediction and returns the result.
  + **CORS Middleware:** *CORSMiddleware* is enabled to allow cross-origin requests from the frontend.
  + API Docs: FastAPI's interactive documentation (Swagger UI) and OpenAPI schema are explicitly configured at /api/docs and /api/openapi.json respectively. This aligns with the Ingress routing, making the API documentation accessible via the main Ingress.

*# inference/main.py snippet*

A blue screen with white text

AI-generated content may be incorrect.

# ... model loading ...

A screenshot of a computer

AI-generated content may be incorrect.

# FastAPI API Documentation Screenshot

A blue and white rectangular object

AI-generated content may be incorrect.

# Frontend

A simple, user-friendly web frontend was developed to allow users to input data and receive predictions from the FastAPI model.

* **Purpose:** To provide a graphical interface for user interaction, abstracting the complexities of the API calls.
* **Implementation:**
  + **HTML** (*frontend/index.html*): Provides the basic structure, including a form with input fields for each feature in the *SmurfFeatures* schema, a submit button, and areas to display the result or error messages. Tailwind CSS is used for minimal styling.
  + **JavaScript** (*frontend/script.js*):
    - Dynamically generates all 44 input fields based on the *SmurfFeatures* schema, ensuring consistency and making the form maintainable. It uses user-friendly labels derived from the schema field names and specifies appropriate input types (e.g., *number* for floats and 0/1 for booleans).
    - Listens for the form submission.
    - Gathers input data from the form fields, parsing values to the correct types (*float* or *boolean*).
    - Sends an asynchronous POST request to the /api/predict endpoint using the fetch API.
    - Handles the API response, displaying the prediction or an error message.

# Frontend Application Screenshot

A screenshot of a computer

AI-generated content may be incorrect.

### Production-Ready Webserver/Reverse Proxy (NGINX)

A dedicated NGINX server is deployed to serve the static frontend assets and act as an internal reverse proxy, fulfilling the requirement for a production-ready web server.

* **Purpose:**
  + **Efficient Static File Serving:** NGINX is highly optimized for serving static content (HTML, CSS, JavaScript files) with high performance and low resource consumption, which is more efficient than having the FastAPI application serve these.
  + **Application-Level Routing/Proxying:** It provides a flexible layer for routing requests within the application's domain. In our setup, it routes all non-API requests to the frontend files.
* **Implementation:**
  + **NGINX Configuration** **(***frontend/nginx.conf***):** A custom NGINX configuration is used:
    - The *location /* block serves *index.html* and other static files from */usr/share/nginx/html*.
    - The *location /api/predict* block (and */api/docs*, */api/openapi.json*) proxies requests directly to http://smurf-fastapi-service:80/. This routes API calls from the frontend to the FastAPI backend, ensuring communication.
  + **NGINX Dockerfile (***frontend.Dockerfile***):** A lightweight NGINX base image is used. This Dockerfile copies the custom *nginx.conf* and all frontend static files into the NGINX web root.
* **Relationship with Ingress (Traefik):**
  + **The Kubernetes Ingress (Traefik)** acts as the cluster-edge reverse proxy/load balancer. It handles incoming traffic from outside the Kubernetes cluster (e.g., from *localhost* on your machine) and routes it to the appropriate Kubernetes Service.
  + Our **deployed NGINX** is an application-level web server/reverse proxy running inside the cluster.
  + In the current *k8s/ingress.yaml* configuration, Traefik routes traffic directly to the *smurf-frontend-service* (which serves the NGINX frontend) for paths matching */* and directly to the *smurf-fastapi-service* for paths matching */api*. This demonstrates how both an external Ingress and an internal application-specific proxy can coexist and complement each other.

### Deployment on Kubernetes

The entire microservices architecture is deployed and managed on a local Kubernetes cluster using *k3d*. Kubernetes orchestrates the containers, provides service discovery, load balancing, and external access.

* **Deployments (***k8s/deployment.yaml***,** *k8s/frontend-deployment.yaml***):**
  + Define how many replicas of each application (FastAPI backend, NGINX frontend) should run (*replicas: 1*).
  + Specify the Docker image to use (e.g., *dutchg/smurf-fastapi:latest*).
  + Crucially, they include *nodeSelector* fields to ensure pods are scheduled on specific nodes (see "Special Kubernetes Setup" below).
* **Services (***k8s/service.yaml***,** *k8s/frontend-service.yaml***):**
  + Provide stable network identities (IP addresses and DNS names) for groups of pods. This is essential because pod IPs are dynamic.
  + The *smurf-fastapi-service* and *smurf-frontend-service* enable other components (like the Ingress or the NGINX proxy) to reliably connect to the respective backend and frontend pods, even if pods are recreated or scaled.
  + They act as internal load balancers, distributing traffic across all healthy pods that match their *selector*.
  + Both are *ClusterIP* types, meaning they are only accessible from within the cluster.
* **Ingress (***k8s/ingress.yaml***):**
  + Manages external access to services within the cluster, typically for HTTP/HTTPS traffic.
  + It defines rules that map incoming hostnames and URL paths to specific Kubernetes Services.
  + Our *smurf-ingress* routes *localhost/* to the *smurf-frontend-service* and *localhost/api* to the *smurf-fastapi-service*. This allows users to access both the frontend and API documentation via the standard *localhost* URL.

# "Special" Kubernetes Setup: Node Selection for Service Isolation

A key aspect of this deployment, beyond standard Kubernetes practices, is the deliberate separation of application components onto dedicated worker nodes within the k3d cluster.

* **What was done:**

1. The k3d cluster was created with two dedicated agent (worker) nodes: *--agents 2*.
2. These agent nodes were explicitly labeled using kubectl label node commands:

* *k3d-k3s-default-agent-0* was labeled *node-type=frontend-node*.
* *k3d-k3s-default-agent-1* was labeled *node-type=backend-node*.

1. The Kubernetes Deployment manifests (*k8s/deployment.yaml* and *k8s/frontend-deployment.yaml*) were updated to include a *nodeSelector* in their *spec.template.spec* section:
   * + *smurf-fastapi* deployment specifies *node-type: backend-node*.
     + *smurf-frontend deployment* specifies *node-type: frontend-node*.

* **Why this is "Special" / Beneficial:**
  + **Resource Isolation:** In a larger cluster, this ensures that, for instance, a CPU-intensive model inference task in the FastAPI backend doesn't starve the frontend NGINX process for resources if they were on the same node.
  + **Independent Scaling:** While not fully demonstrated with *replicas > 1* here, in a production scenario, you could scale your *frontend-node* pool or *backend-node* pool independently based on the specific load requirements of each service.
  + **Failure Domains:** If a "backend-node" fails, it primarily affects backend services, potentially allowing the frontend to remain available and display a graceful error message.
  + **Specialized Hardware:** This pattern is crucial if certain services require specific hardware (e.g., GPUs for machine learning models), allowing you to schedule them only on nodes equipped with those resources.
  + **Clearer Operations:** It provides a clearer operational picture of where each logical service is running, simplifying troubleshooting and capacity planning.

This deliberate node separation, even in a local k3d environment, showcases an understanding of best practices for microservice deployment in Kubernetes, enabling better resource management, scalability, and resilience.

## Task 3: Automation – GitHub Actions, Pipeline Control, and Deployment Strategy

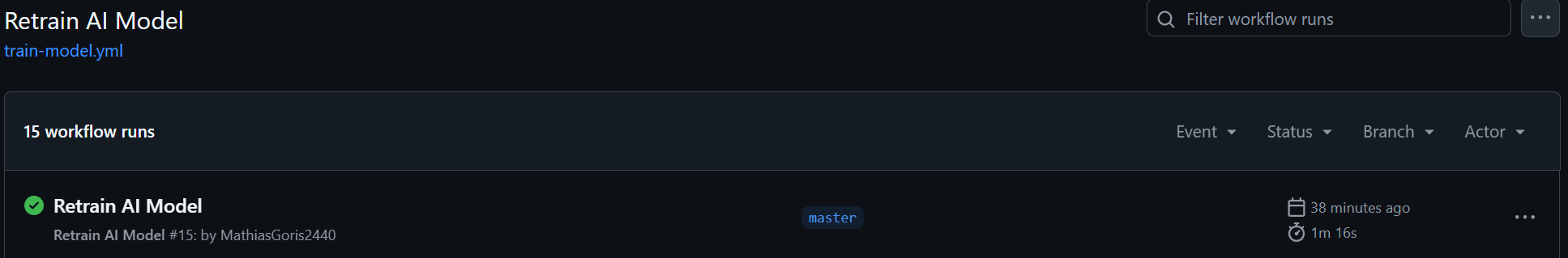
We implemented a CI/CD pipeline using **GitHub Actions** and a **self-hosted runner** to automate model training, deployment, and frontend/backend updates. The automation is divided into three main workflows:

### 1. train-model.yml – Automatic Model Retraining

**Trigger**:  
This workflow is triggered when any code changes occur in the profit-predictor/\*\* directory. This includes updates to the training script, pipeline definition, or model logic.

**Purpose**:

* Submits a training job to Azure ML using a defined pipeline (pipeline.yaml).
* Uses the registered dataset (cleaned\_dataset) and trains the model on the cc-profit-train compute.
* The trained model is automatically registered in the Azure ML model registry as smurf-regressor.



### 2. update-frontend-api-config.yml – Redeploy on Code or Config Changes

**Trigger**:  
Triggered automatically on changes to:

* Frontend (frontend/\*\*)
* Kubernetes config files (k8s/\*\*)
* Inference backend (inference/\*\*)
* Dockerfiles (Dockerfile, frontend.Dockerfile)

**Purpose**:

* Rebuilds Docker images for both FastAPI and the frontend.
* Pushes updated images to Docker Hub.
* Applies Kubernetes manifests (deployment, service, ingress).
* Triggers a rollout restart for updated deployments.

**Kubernetes Strategy**:  
We chose a **rolling update** strategy via kubectl rollout restart, which:

* Minimizes downtime
* Ensures zero-downtime deployment
* Automatically replaces pods with updated ones

A screenshot of a video game

AI-generated content may be incorrect.

### 3. update-model.yml – Download & Deploy Latest Model

**Trigger**:  
Manually triggered (via workflow\_dispatch) when we want to deploy the **latest trained model**.

**Purpose**:

* Authenticates with Azure ML and fetches the latest version of the registered model smurf-regressor.
* Removes the old model from the inference/ directory.
* Downloads the new model.
* Rebuilds the FastAPI Docker image with the updated model.
* Pushes the image and triggers a rollout restart in Kubernetes.

**Manual Work Left**:

* Currently, this is **not triggered automatically** after training. Linking train-model.yml and update-model.yml could fully automate the lifecycle.

A screen shot of a computer

AI-generated content may be incorrect.

### Version Controlling

**Model Versions**: Managed via Azure ML model registry. Each pipeline run produces a new version (incremental).

**Code Versions**: GitHub is used to track changes in:

* Training pipeline
* Frontend code
* Inference API
* Kubernetes manifests

**Image Versions**: Docker images are always tagged as latest, but history is tracked on Docker Hub.

## Reflection

### Task 1: Implement the Training Process in a Cloud AI Service (Azure ML)

Setting up the training process in Azure ML was a very accessible experience. I was surprised by how little code was needed to get something running. The YAML files were straightforward to write, and Azure’s documentation and interface made it easy to go from experimenting on the website to using the CLI and integrating everything in Visual Studio Code.

The most challenging part of this task was setting up the environment. Getting the dependencies right and making sure the training script could run in a clean, reproducible environment took some trial and error. However, Azure ML provided clear error messages, which made debugging much easier than expected. Each time something failed, the platform gave a good explanation of what went wrong, which helped me fix the issue quickly and keep moving forward.

Overall, this task gave me a solid introduction to cloud-based ML workflows. It showed me how powerful and scalable tools like Azure ML can be, even for smaller experiments or dummy models.

### Task 2: Deploy the Trained Model Using Kubernetes with a Microservice Architecture

Task 2, focused on deploying our model using Kubernetes and a microservices architecture, was a significant learning experience for me. Initially, Kubernetes felt like a daunting, abstract concept. During the labs I had a lot of trouble grasping \*what exactly\* I was doing or what Kubernetes did or was.

However, as we moved through the project, from building the Docker images for our FastAPI backend and NGINX frontend to configuring their Kubernetes manifests, the pieces began to click. Implementing the nodeSelector to logically separate our frontend and backend services onto distinct nodes was a particularly insightful step. It demonstrated the practical application of Kubernetes' scheduling capabilities for resource isolation and enhanced scalability, something that was initially quite abstract.

Seeing the entire system come alive, with the frontend communicating seamlessly with the backend via the Kubernetes network and Ingress, was incredibly rewarding. The iterative process of debugging ErrImagePull issues or FailedScheduling events, and then successfully resolving them, solidified my understanding. By the end of Task 2, I not only understood *what* Kubernetes does but *why* it's so powerful for managing complex, scalable, and resilient applications. It transformed from a confusing theory into a tangible, powerful tool, and frankly, I had a genuinely fun and rewarding time seeing the project run smoothly.

### Task 3: Implement CI/CD with GitHub Actions

Working with GitHub Actions turned out to be much more straightforward than I initially expected. Setting up basic workflows was easy, and the interface made it simple to understand how to trigger different actions based on repository changes.

Connecting GitHub Actions to Azure — including using secrets like credentials and workspace details — was surprisingly smooth. The process was well-documented, and GitHub’s UI made managing secrets intuitive. Just like with Azure ML, the error messages in the GitHub Actions logs were generally very clear, which helped a lot when something went wrong.

The only real challenge I ran into was understanding how the self-hosted runners worked. At first, it was a bit confusing figuring out what should be done and how to separate concerns across different workflows (e.g. retraining the model, redeploying the model, and updating the frontend/API). Once I got the structure right and tested each step, everything started working reliably.

In the end, this task really showed how powerful and flexible GitHub Actions can be. With relatively little effort, I was able to set up full automation — from training to deployment — and I now have a much better understanding of how CI/CD works in a real-world machine learning pipeline.