

# **HomeLens CA**

Predict the median house values in California (1990) using  
block-level information

Author: Matteo Gianferrari

# Contents

<b>1.Introduction.....</b>	<b>2</b>
1.1.Project Description.....	2
1.2.Design Choices.....	2
1.3.Project Objectives.....	3
<b>2.Tools.....</b>	<b>3</b>
2.1.Git.....	3
2.2.GitHub.....	3
2.2.1.GitHub Actions.....	4
2.2.2.GitHub Packages.....	4
2.3.PyCharm.....	5
2.3.1.UI Microservice.....	5
2.3.2.Model Microservice.....	6
2.3.3.Pylint.....	6
2.3.4.Pytest.....	7
2.4.Google Colab.....	7
2.5.MLflow.....	8
2.5.1.MLflow Experiments.....	8
2.5.2.MLflow Model Registry.....	10
2.6.Docker.....	11
2.6.1.Hadolint.....	11
2.6.2.Docker-compose.....	11
<b>3.Continuous Integration.....</b>	<b>12</b>
3.1.UI Pipeline.....	12
3.2.Model Pipeline.....	13
3.2.1.REST API Pipeline.....	13
3.2.2.MLflow Pipeline.....	13
<b>4.Continuous Deployment.....</b>	<b>14</b>
4.1.UI Pipeline.....	14
4.2.Model Pipeline.....	15
4.3.Render.....	16
<b>5.Monitoring.....</b>	<b>16</b>
<b>6.Possible Improvements.....</b>	<b>17</b>
<b>7.Conclusions.....</b>	<b>17</b>

# 1.Introduction

HomeLens CA is a system designed to **predict median house values** in California (1990) based on block-level information. By leveraging modern **MLOps** practices and **microservice architecture**, a data scientist can rapidly **experiment with models** while maintaining a basic, robust, and scalable infrastructure crafted similarly to what leading technology companies deploy in real-world scenarios.

## 1.1.Project Description

HomeLens CA comprises two core microservices that work together to deliver end-to-end functionality:

1. **UI Microservice** (A Gradio-based interface providing a user-friendly frontend. This service allows users to input relevant housing features and receive a predicted median house value without requiring extensive technical background).
2. **Model Microservice** (A REST API powered by an MLflow-registered model, exposed via the FastAPI framework).

By decoupling the prediction logic from the UI, this architecture promotes **flexibility, maintainability, and scalability**. The model can be easily updated or replaced, ensuring that improvements in prediction quality flow seamlessly into production.

## 1.2.Design Choices

During the development of HomeLens CA, several key tools and platforms were selected based on their alignment with best practices and industry standards:

- **Google Colab** (A free, browser-based environment with optional high-end GPU runtimes for model training and experimentation, eliminating the need for local hardware).
- **PyCharm** (A feature-rich IDE offering integrated plugins for version control, testing, and Docker support—all of which help streamline the development process).
- **GitHub Actions** (Used for Continuous Integration and Continuous Deployment workflows, automating tasks such as code linting, unit testing, Docker builds, and image pushes to GitHub Packages).
- **MLflow** (A robust platform for experiment tracking and model lifecycle management. This project hosts MLflow via DagsHub to record metrics, track versions, and register the champion model).
- **Render** (A simple, free hosting solution where each microservice is defined as a web service. Render's ability to accept web hooks fits easily into the CD pipeline, ensuring rapid re-deployment after successful builds).

## 1.3. Project Objectives

Below are reported the main goal followed during the development of the project:

1. **Establish a Production-like Environment** (HomeLens CA showcases essential MLOps components—version control, automated testing, containerization, and model registry—to mimic the basic workflows of large tech companies).
2. **Enable Efficient Deep Learning Research** (major part of the time can be focused on designing and refining neural network architectures, offloading tasks like environment setup, container deployment, and model versioning to automated pipelines).
3. **Facilitate Easy Scaling and Future Expansion** (By adopting microservices and modular tools, the project can scale horizontally, switch hosting providers when needed, and incorporate new services based on business requirements).

## 2. Tools

The development of a robust **MLOps infrastructure** relies on a carefully selected set of tools that streamline the entire machine learning lifecycle—from code development to continuous integration and deployment (CI/CD), and finally to research and experimentation. By leveraging these tools, we can establish a **basic production-like environment** suitable for large technology companies, ensuring scalability, reliability, and reproducibility.

### 2.1. Git

**Git** is a widely adopted distributed version control system that tracks changes in the codebase and simplifies collaboration among team members. In this project, a local Git repository is maintained, which is synchronized with a remote repository on **GitHub**.

```
matteo@dellmatteo:~/Projects/homelens-ca$ git status
On branch main
Your branch is up to date with 'origin/main'.
```

To avoid committing unnecessary files—such as environment variables—into the repository, a *.gitignore* file is included. This file was adapted from a similar Python-based project and customized to meet the specific needs of this project.

### 2.2. GitHub

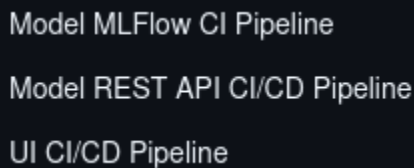
GitHub is a widely adopted remote platform for hosting and managing Git repositories. By leveraging GitHub, the development processes are streamlined, it maintains clear

documentation of changes, and integrates additional services such as **GitHub Actions** and **GitHub Packages**. This integration helps implement continuous integration and delivery (CI/CD) practices..

### 2.2.1. GitHub Actions

GitHub Actions is the automation tool used in this project to **manage and orchestrate all workflows**, from building and testing code to deploying production-ready images. Workflow files stored in the repository define these processes, making them easily configurable and repeatable.

To enable advanced functionalities—such as updating the MLflow Model Registry or pushing Docker images to GitHub Packages—**repository settings** must be configured to grant GitHub Actions “read and write” permissions. Furthermore, **secrets** (e.g., usernames, tokens, and API keys) are stored securely in GitHub’s repository settings to ensure that sensitive information remains protected and that workflows can authenticate with external services without exposing credentials in the codebase.



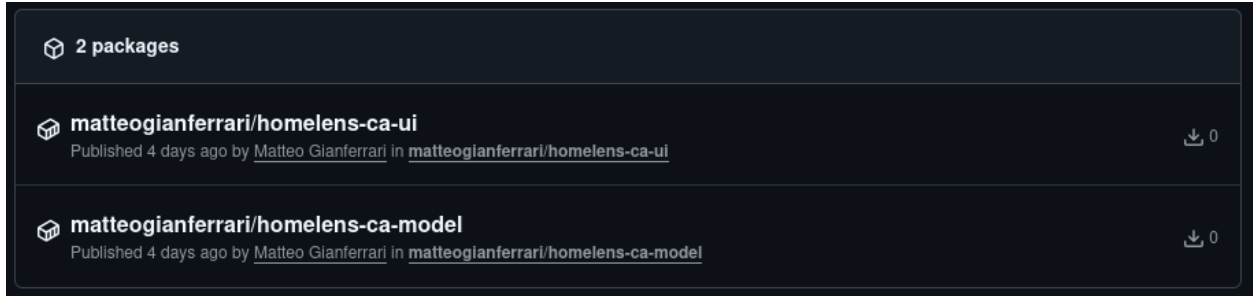
Model MLFlow CI Pipeline  
Model REST API CI/CD Pipeline  
UI CI/CD Pipeline

More details about the pipelines, including triggers and stages, are discussed in the dedicated pipeline chapters.

### 2.2.2. GitHub Packages

GitHub Packages is an integrated platform for hosting and managing software packages, including **Docker images**. By consolidating source code and associated packages within GitHub, the project benefits from centralized control over the entire software supply chain.

In this project, both microservice images are built and published to GitHub Packages, providing a **public registry** that can be easily pulled into local development environments or deployed to production hosting platforms. When combined with GitHub Actions, the result is a streamlined, end-to-end DevOps workflow that includes code versioning, continuous integration, and automated deployment.



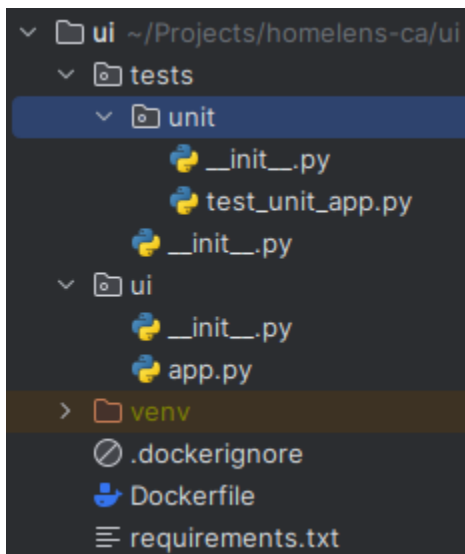
## 2.3.PyCharm

PyCharm serves as the primary environment for creating and maintaining the project's source code. It offers advanced features such as intelligent code completion, debugging tools, built-in version control integration, and seamless management of virtual environments. These capabilities streamline the development process on both the UI and Model microservices while maintaining code quality and consistency.

### 2.3.1.UI Microservice

The UI microservice leverages **Gradio**, a Python library that simplifies building and sharing user-friendly **web interfaces** for Deep Learning models. This approach allows anyone to interact with the model predictions through a browser without requiring extensive technical knowledge.

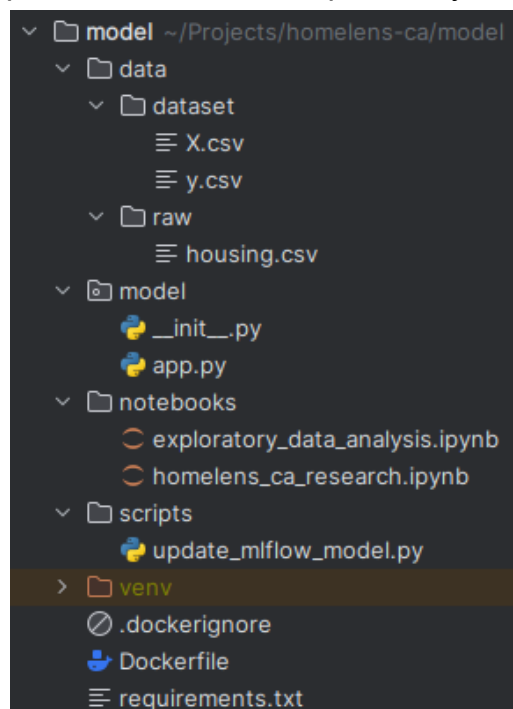
The image reported below shows how the UI microservice source code is structured. By structuring the code in this manner, the UI microservice remains modular, testable, and easy to maintain, whether running locally or deployed in production.



### 2.3.2. Model Microservice

The Model microservice exposes a basic **REST API** through **FastAPI**. Normally, a hosted MLflow instance could automatically generate an API for the champion model, but since this project uses MLflow's free tier, implementing the API manually with FastAPI becomes necessary.

The image reported below shows how the Model microservice source code is structured. With this setup, the Model microservice can be run or containerized to serve predictions, either independently or as part of a larger application ecosystem.



### 2.3.3. Pylint

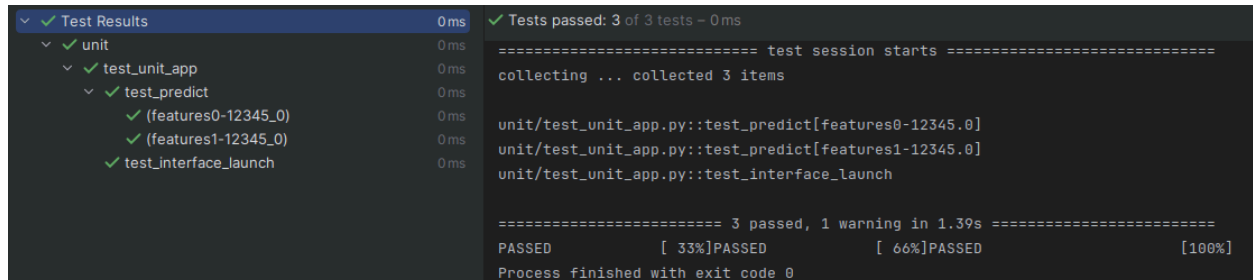
To enforce **coding standards** and identify potential issues early, Pylint is used throughout the development cycle for both microservices. A shared `.pylintrc.toml` configuration file ensures consistent style and linting rules across the entire codebase.

```
(venv) matteo@dellmatteo:~/Projects/homelens-ca/ui$ pylint --rcfile ../.pylintrc.toml ui/
-----
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)

(venv) matteo@dellmatteo:~/Projects/homelens-ca/model$ pylint --rcfile ../.pylintrc.toml model/
-----
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
```

### 2.3.4. Pytest

Pytest is employed for **unit testing**, particularly within the UI microservice, though it can be extended to other components as well. Tests verify functions such as the prediction logic and interface initialization.



```
Test Results 0ms
├── unit 0ms
│   ├── test_unit_app 0ms
│   │   ├── test_predict 0ms
│   │   │   ├── (features0-12345_0) 0ms
│   │   │   ├── (features1-12345_0) 0ms
│   │   └── test_interface_launch 0ms
└── 3 tests passed: 3 of 3 tests - 0ms

===== test session starts =====
collecting ... collected 3 items

unit/test_unit_app.py::test_predict[features0-12345_0]
unit/test_unit_app.py::test_predict[features1-12345_0]
unit/test_unit_app.py::test_interface_launch

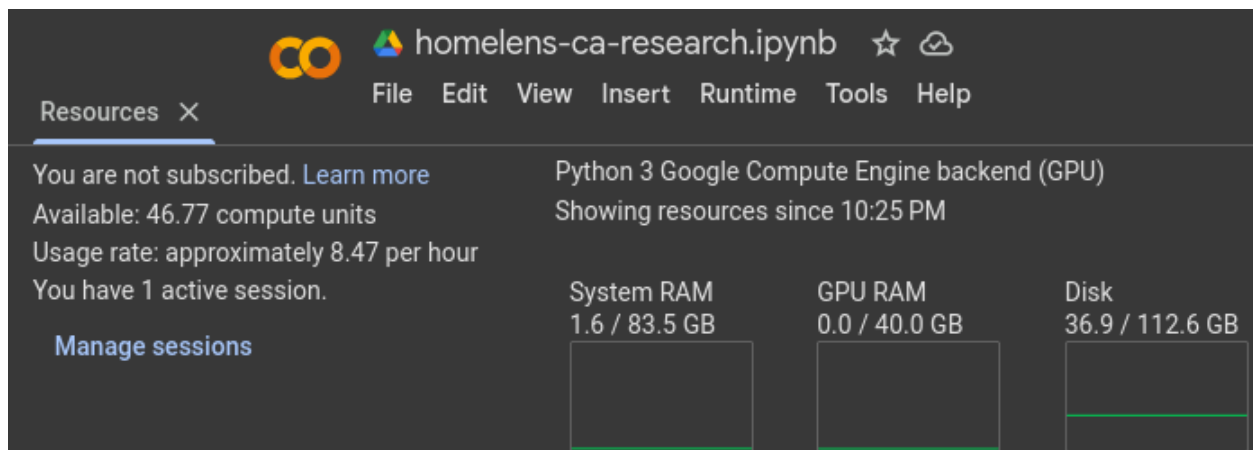
===== 3 passed, 1 warning in 1.39s =====
PASSED [ 33%] PASSED [ 66%] PASSED [100%]
Process finished with exit code 0
```

Running the tests locally and in the CI pipeline ensures that changes do not break existing functionality. In the future, integration tests may be added to provide **end-to-end validation** of interactions between microservices.

## 2.4. Google Colab

Google Colab is a browser-based environment that allows users to develop and run **Jupyter notebooks** without setting up a local server. It offers a free tier of compute resources and supports high-end GPU runtimes for more computationally intensive tasks, such as **training deep learning models**. By leveraging Colab, a researcher can focus on developing and refining model architectures, while DevOps processes handle environment setup and automated workflows.

Google Colab provides on-demand access to GPU resources and reports real-time usage statistics for system RAM, GPU RAM, and Disk capacity. This makes **scaling up computational resources** straightforward for experiments that require additional power. If the free tier's limits are reached, users can purchase compute units to accommodate larger or more complex training workloads.





In this project, a template notebook—named *homelens-ca-research.ipynb*—has been created to streamline model experimentation. By inserting the necessary credentials (e.g., GitHub tokens and MLflow secrets) and updating only the model and hyperparameter sections, it is possible to quickly begin training new architectures without manually configuring dependencies. This setup allows to **concentrate time** on **deep learning model innovation** while relying on DevOps automation for environment provisioning and continuous integration tasks.

All model artifacts, parameters, and metrics are automatically logged to **MLflow**, facilitating easy tracking and versioning of experiments. More details about MLflow are discussed in the dedicated paragraph.

## 2.5. MLflow

MLflow is an open-source platform designed to streamline the machine learning lifecycle by providing tools for experiment tracking and model packaging. In this project, MLflow is hosted for free on **DagsHub**, a platform that integrates closely with **GitHub** repositories. This setup allows **automatic synchronization** of code, data, and experiment logs, which simplifies collaboration and version control for machine learning research and development.



By leveraging the free tier of DagsHub, the project gains access to key MLflow features such as **Experiments** (for tracking runs and comparing metrics) and the **Model Registry** (for storing and versioning production-ready models). These capabilities ensure a standardized process for developing, evaluating, and updating models over time.

### 2.5.1. MLflow Experiments

The Experiments feature in MLflow organizes the various model training runs into logical groups, enabling to track parameters, metrics, and artifacts for each run. In this project, experiments fall into two main categories:

1. **Pipeline Experiment:** Focuses on developing the data pre-processing pipeline.
2. **Research Experiments:** Manages a larger set of runs, including various neural network architectures and hyperparameter configurations.



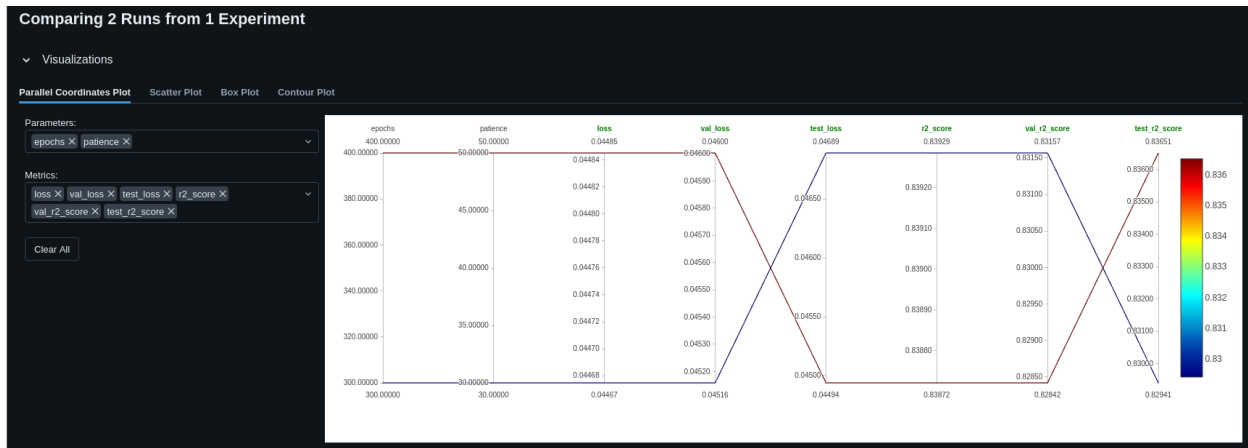
MLflow provides visualizations (e.g., loss curves,  $R^2$  scores, and other performance metrics) to compare different runs and gain deeper insights into model behavior. For instance, it's possible to plot training and validation **loss** side by side to **detect overfitting** or evaluate learning progression over time.



Similarly, the  **$R^2$  score** plots help assess a model's **predictive power** against validation data. By comparing results across multiple runs, it's possible to systematically identify optimal configurations and make data-driven decisions.



It is also possible to select multiple runs and compare them using the dedicated function. Comparing model performances allows to create complex plots with different kinds of metrics.



## 2.5.2. MLflow Model Registry

The Model Registry in MLflow serves as a **centralized repository** for storing and versioning models. Each registered model entry maintains a history of versions, complete with associated metadata such as training parameters and performance metrics. Models stored in the registry can be pulled and loaded via **MLflow's Python API**, facilitating easy integration into other services or microservices.

In this project:

- **Production Model:** The currently deployed or “champion” model is tagged as **@champion**.
- **Production Pipeline:** The currently deployed or “champion” pipeline is tagged as **@champion**.

homelens-ca-model

Created Time: 2025-01-24 19:41:51 Last Modified: 2025-01-25 15:03:01

> Description [Edit](#)

> Tags

> Versions [Compare](#)

New model registry UI ☐

Version	Registered at	Created by	Tags	Aliases	Description
<input checked="" type="radio"/> Version 2	2025-01-25 15:03:01		<a href="#">Add</a>	<a href="#">@champion</a> <a href="#">Edit</a>	
<input checked="" type="radio"/> Version 1	2025-01-24 20:02:40		<a href="#">Add</a>	<a href="#">Add</a>	

1

When a new model version outperforms the existing champion, the CI pipeline can **promote it to production** by updating the **@champion** tag. Details on how the CI pipeline automates these updates and promotions can be found in the dedicated chapter.

## 2.6.Docker

Docker is a **containerization platform** that packages applications and their dependencies into lightweight, portable containers. In this project, Docker is used to build images for both the **UI** and **Model** microservices (as described in Section 2.3: PyCharm for each microservice's respective Dockerfile). By containerizing these services, the project ensures consistent behavior across different environments—local machines, or cloud platforms. This approach enables rapid deployment, isolation of services, and simplified scalability, making Docker an ideal choice for modern **MLOps workflows**.

### 2.6.1.Hadolint

Hadolint is a linter specifically designed to check **Dockerfiles** for best practices, potential errors, and security vulnerabilities. It helps maintain a clean, efficient, and secure build process. In the project, a shared `.hadolint.yml` configuration file standardizes linting rules across both microservices.

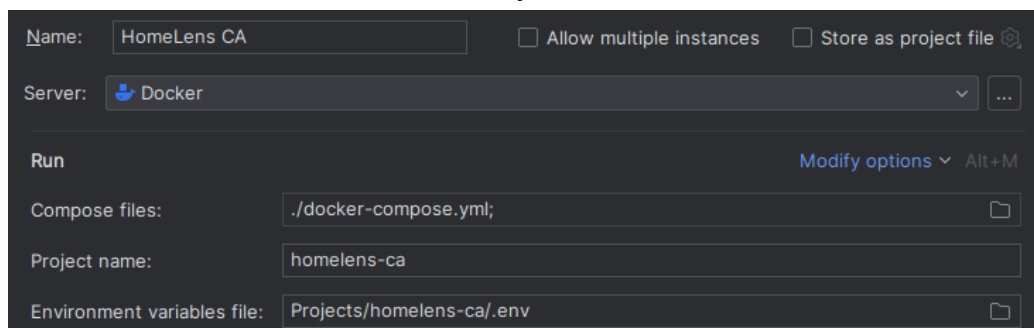
This step is included in **CI pipelines** to ensure Dockerfiles remain compliant with standards over time.

### 2.6.2.Docker-compose

In order to run the system locally with both microservices, the project relies on docker-compose. This tool **orchestrates multiple containers**, enabling them to communicate via a dedicated network.

Running the entire system with `docker-compose up --build` creates both microservices and links them through the **app-network**. This local, containerized environment mirrors a production-like setup.

Docker is perfectly integrated with PyCharm, which allows the creation of a run configuration for Docker-compose, where it's possible to specify the `.env` file containing all the environment variables used by the two microservices.



## 3. Continuous Integration

Continuous Integration (CI) is a development practice where developers integrate code into a shared repository frequently, triggering automated builds and tests. In this project, CI pipelines handle core tasks like linting Python code and Dockerfiles, unit testing, containerizing microservices, and updating MLflow. By leveraging **GitHub Actions**, these processes run automatically whenever changes are pushed to the GitHub repository, aligning with the overall **MLOps** strategy.

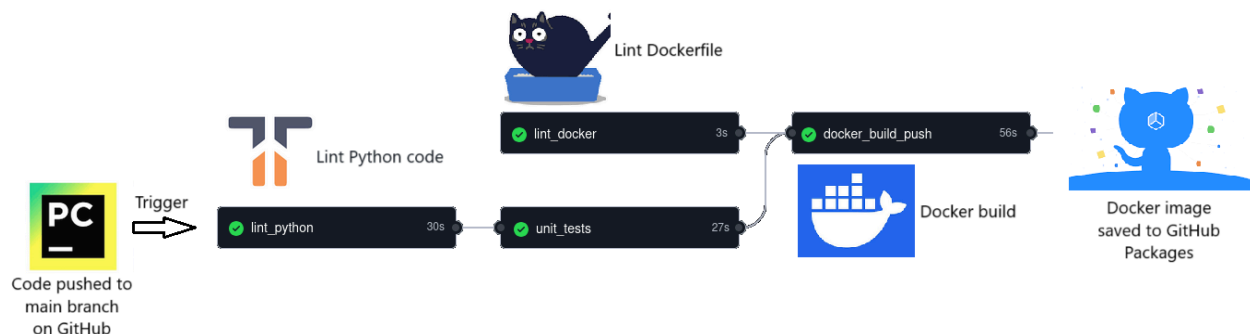
### 3.1. UI Pipeline

This pipeline runs under two conditions:

- **Push to main** for the UI microservice code (i.e., changes detected in the *ui* folder).
- **Manual triggers** for debugging (in this case the built image is not pushed to GitHub Packages).

The diagram below shows the high-level stages that occur when the pipeline is triggered:

1. From **PyCharm** push to main on GitHub (the committed changes related to the *ui* folder are pushed on the main branch).
2. **Lint Python Code** (Pylint checks the code against best practices and coding standards).
3. **Lint Dockerfile** (Hadolint checks the Dockerfile in the *ui* directory for potential errors and best-practice violations).
4. **Unit Tests** (Pytest runs on the UI microservice tests to confirm functionality).
5. **Docker Build & Push** (If the above steps succeed, a Docker image is built and tagged. The image is then pushed to GitHub Packages, unless it was a manual debug trigger, in which case no image push is performed).



Only the linting jobs can be executed in **parallel**, all the **other steps depend** on the previous one. Thus, errors are caught early, and only a successful build proceeds to the deployment phase.

## 3.2.Model Pipeline

The Model microservice has two **separate pipelines** to cover distinct workflows; one for the **REST API** and one for the **MLflow** updates triggered by research notebooks.

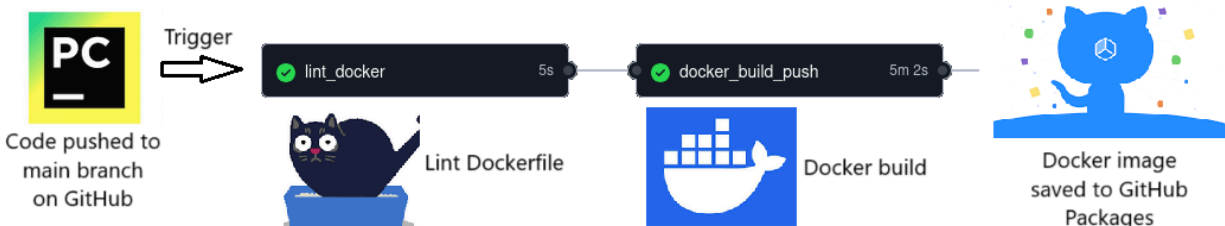
### 3.2.1.REST API Pipeline

This pipeline runs under two conditions:

- **Push to main** for the Model microservice code (i.e., changes detected in the *model* folder excluding notebooks).
- **Manual triggers** for debugging (in this case the built image is not pushed to GitHub Packages).

The diagram below shows the high-level stages that occur when the pipeline is triggered:

1. From **PyCharm** push to main on GitHub (the committed changes related to the *model* folder are pushed on the main branch).
2. **Lint Dockerfile** (Hadolint checks the Dockerfile in the *model* directory for potential errors and best-practice violations).
3. **Docker Build & Push** (If the above step succeeds, a Docker image is built and tagged. The image is then pushed to GitHub Packages, unless it was a manual debug trigger, in which case no image push is performed).



All the **steps depend** on the previous one. Thus, errors are caught early, and only a successful build proceeds to the deployment phase.

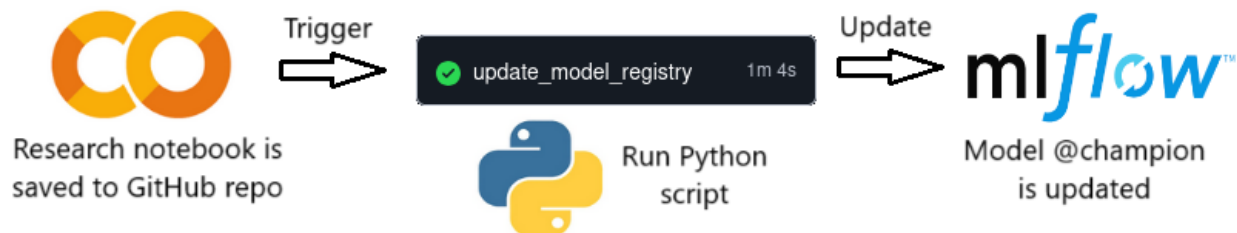
### 3.2.2.MLflow Pipeline

This pipeline activates in the following scenarios:

- **Colab Notebook** changes committed to main (e.g., saving research experiments from a notebook).
- **Manual triggers** for debugging.

For model updates tied to research outputs in Google Colab, the pipeline follows this flow:

1. **Colab Notebook** Saved to GitHub (When a user saves or commits the research notebook).
2. **Run Python Script** (the script contained in the Python file `update_mlflow_model.py` checks if the run ID of the model tagged `@champion` is equal to the run ID of the best model in the experiment list, based on the test  $R^2$  score. If a model is better than the current champion, the new model is elected to production model and tagged).
3. **MLflow Model Registry** Updated (the updated champion model and versioning information are stored in MLflow Model Registry. This ensures that any newly validated model can be promoted seamlessly).



This pipeline automates the **last mile of integrating research** results into production. For more information on MLflow usage and hosting on DagsHub, refer to the MLflow paragraph.

## 4. Continuous Deployment

Continuous Deployment (CD) is the practice of **automatically deploying changes** to production once they pass the required validations—such as linting, unit tests, and container builds—handled by the Continuous Integration (CI) pipelines. In this project, CD pipelines work in tandem with the **CI pipelines** (discussed in the previous chapter), triggering a deployment process upon successful CI completion. This approach ensures that new versions of the UI and Model microservices are **rapidly and consistently made available** to end users, minimizing downtime and manual intervention.

### 4.1. UI Pipeline

This deployment pipeline is activated once the UI CI pipeline (see Section 3.1) completes successfully. Both pipelines reside in the same GitHub Actions workflow file, but the CD pipeline jobs depend on the outcome of the CI pipeline jobs.

As illustrated in the diagram below, the UI CD pipeline consists of the following steps:

1. **CI Completion** (The CI workflow finishes, confirming that the new Docker image is built and pushed to GitHub Packages).

2. **Deploy Step** (A deployment job runs, using a curl command to invoke a web hook on the hosting platform).
3. **Render re-deploy** (The hosting service pulls the newly built image from GitHub Packages and updates the running instance of the UI microservice).



This pipeline structure ensures that any newly validated Docker image for the UI is automatically deployed once it passes all CI checks.

## 4.2.Model Pipeline

Similarly, the Model CD pipeline kicks off when the Model CI pipeline (see Section 3.2.1) finishes successfully, indicating that the new image for the Model microservice is ready for deployment.

The Model microservice deployment flow mirrors the UI pipeline:

1. **CI Completion** (The CI workflow finishes, confirming that the new Docker image is built and pushed to GitHub Packages).
2. **Deploy Step** (A deployment job runs, using a curl command to invoke a web hook on the hosting platform).
3. **Render re-deploy** (The hosting service pulls the newly built image from GitHub Packages and updates the running instance of the Model microservice).



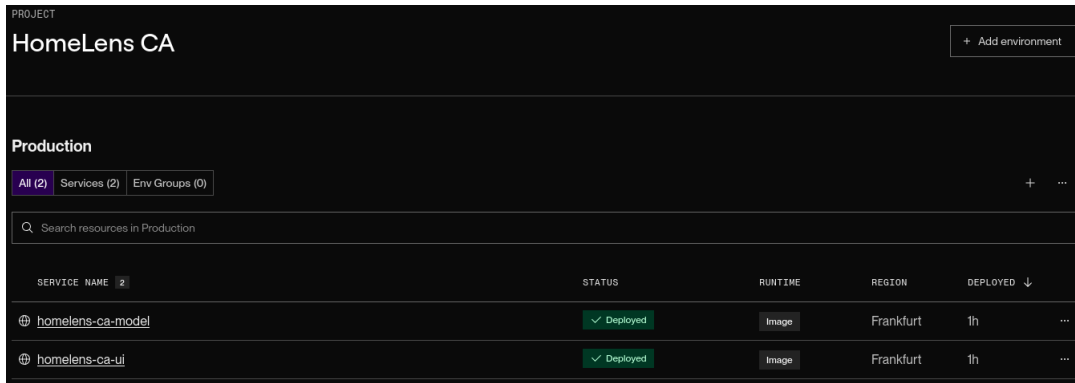
By decoupling the CI and CD pipelines yet **chaining them through success criteria**, the project achieves fully automated releases of both the UI and Model services. This methodology **avoids manual labor** and reduces the time between new code validation and production deployment, reinforcing a robust DevOps strategy.



## 4.3.Render

Render is a cloud hosting platform that offers **free tiers** for smaller applications. In this project, each microservice (UI and Model) is deployed on Render as a **separate web service**, pointing to the corresponding Docker image hosted on GitHub Packages. Environment variables—such as API keys or model endpoints—can be configured directly in the Render dashboard, ensuring secure and flexible deployment.

By setting up **web hooks in Render** and invoking them via the CD pipeline (see Sections 4.1 and 4.2), the platform can automatically pull and redeploy the newest Docker images whenever the pipelines successfully complete. This seamless integration **reduces manual effort**, allowing focusing time on **model research** rather than provisioning or maintenance tasks.



PROJECT: HomeLens CA

+ Add environment

Production

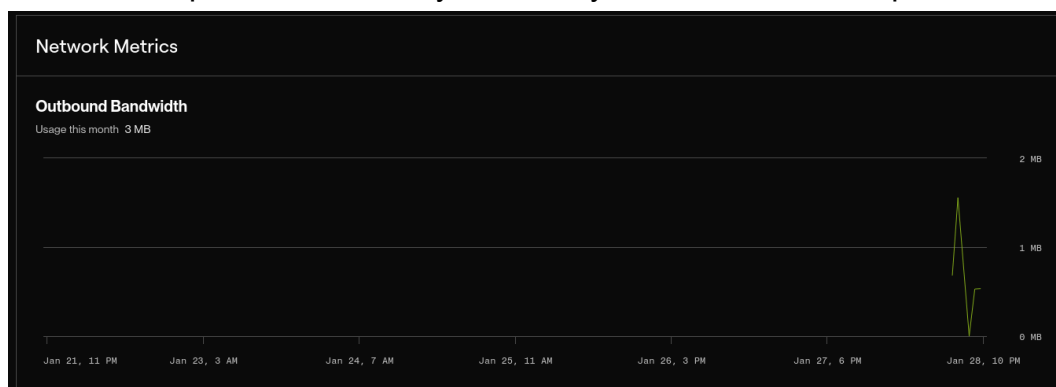
All (2) Services (2) Env Groups (0)

Search resources in Production

SERVICE NAME	STATUS	RUNTIME	REGION	DEPLOYED
homelens-ca-model	✓ Deployed	Image	Frankfurt	1h
homelens-ca-ui	✓ Deployed	Image	Frankfurt	1h

## 5.Monitoring

Monitoring is an important component of any production environment. In this project, the UI and Model microservices are deployed on Render (see Chapter 4.3), which includes **basic monitoring features** in its free tier—such as bandwidth usage. While these default tools offer a starting point, advanced monitoring capabilities can significantly enhance the product's reliability, scalability, and overall user experience.



## 6. Possible Improvements

While the current architecture and tooling provide a **solid foundation** for developing, testing, and deploying microservices, ongoing growth and evolving business requirements may demand further refinement. This chapter outlines potential improvements that could enhance scalability, maintainability, and overall efficiency. By learning from practices employed at **large tech companies**, the project can stay aligned with **industry standards** and remain adaptable to future needs.

Below are reported possible improvements to implement:

- **Branching Model:** To facilitate more complex development workflows, adopting a branching strategy can be beneficial. For instance, adding dedicated development, feature/\*, or bugfix/\* branches allow parallel work without disrupting the main branch.
- **Hosting Migration:** While Render provides a convenient free tier, a business may need a more robust provider for higher traffic demands, compliance requirements, or advanced features. Potential alternatives include AWS, Azure, or GCP, each of which integrates seamlessly with Docker and offers additional managed services (e.g., load balancers, auto-scaling).
- **Kubernetes Adoption:** Many large-scale organizations rely on Kubernetes to orchestrate container deployments. Migrating from Docker Compose to Kubernetes could enable horizontal scaling, rolling updates, and advanced networking configurations. This refactoring would also facilitate monitoring and logging integrations aligned with industry standards.
- **ML Model Monitoring:** For advanced ML workloads, specialized ML monitoring platforms can track drift in data distribution and model performance. Integrating with MLflow Model Registry would further streamline end-to-end model tracking.

## 7. Conclusions

HomeLens CA successfully demonstrates how a basic modern MLOps framework can be applied to a real-world deep learning problem—predicting median house values based on block-level information from 1990 California data. By integrating industry-standard tools such as Google Colab, PyCharm, GitHub Actions, MLflow, and Render, the project lays a basic, but robust foundation template for future projects related to deep learning research.

The chosen microservice architecture separates UI concerns from model logic, promoting flexibility, maintainability, and straightforward updates or replacements of the predictive model. Leveraging these open-source technologies, all of which are widely

used by leading tech companies, further cements best practices for version control, continuous integration, and continuous deployment.

This approach has not only streamlined the development process and minimized manual configuration but also established a scalable pathway for ongoing refinements. Whether adding more sophisticated monitoring, transitioning to larger-scale hosting, or experimenting with new deep learning architectures, the groundwork has been set for future growth.