# 2b. Tutorial - Train a model in Code Repositories

Before starting this step of the tutorial, you should have completed the modeling project set up. In this tutorial, you can choose to either train a model in a Jupyter® notebook or in Code Repositories. Jupyter® notebooks are recommended for fast and iterative model development whereas code repositories are recommended for production-grade data and model pipelines.

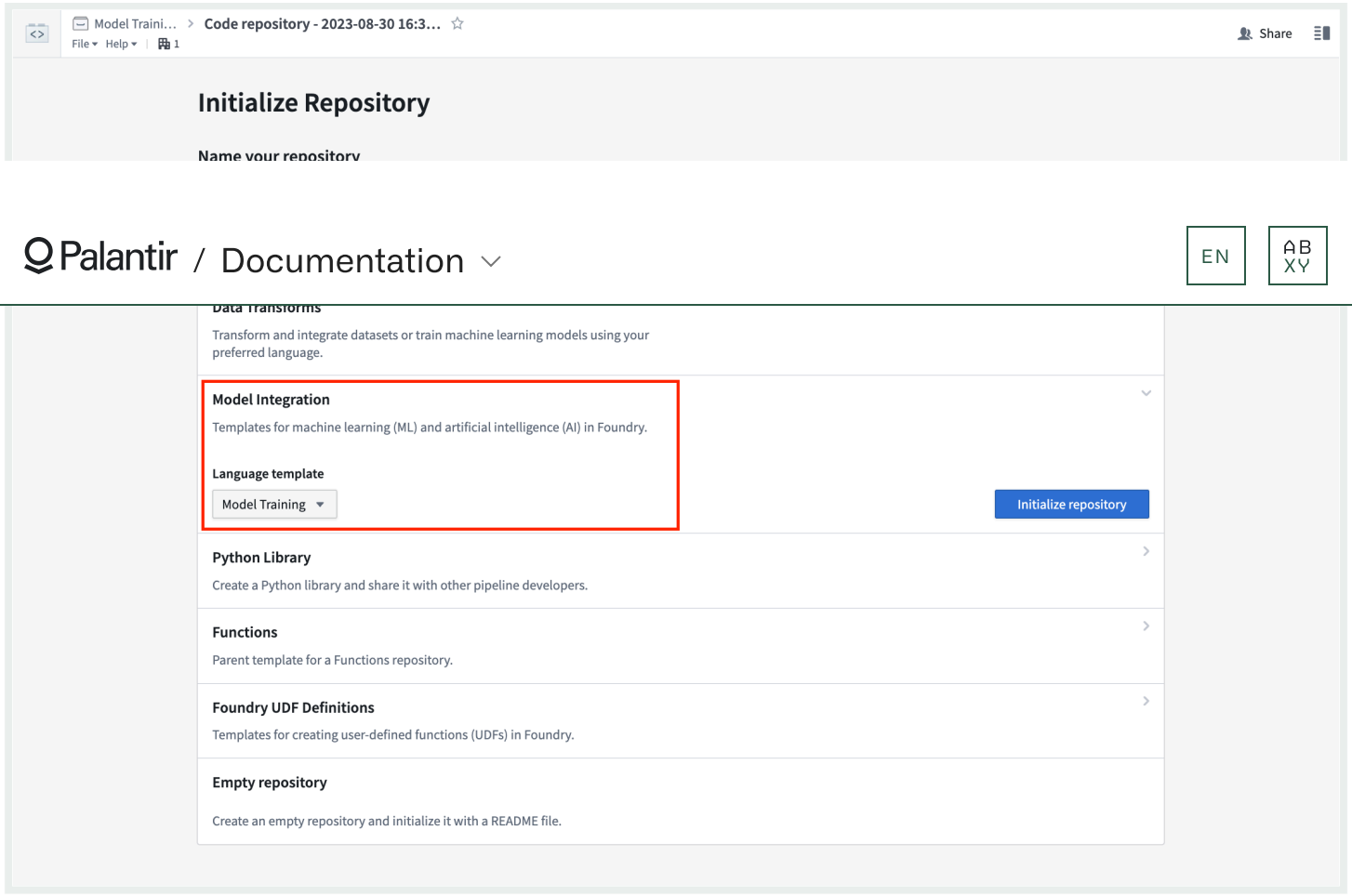
In this step of the tutorial, we will train a model in Code Repositories. This step will cover:

1. Creating a code repository for model training
2. Splitting feature data for testing and training
3. Authoring model training logic in Code Repositories
4. Testing inference logic in Code Repositories
5. Viewing a model and submit it to a modeling objective

## 2b.1 How to create a code repository for model training

The Code Repositories application in Foundry is a web-based development environment for authoring production-grade data and machine learning pipelines. Foundry provides a templated repository for machine learning called the `Model Training` template.

**Action**: In the `code` folder you created during the previous step of this tutorial, select **+ New > Code repository**. Your code repository should be named in relation to the model that you are training. In this case, name the repository "median_house_price_model_repo". Select **Model Training** template, then **Initialize**.

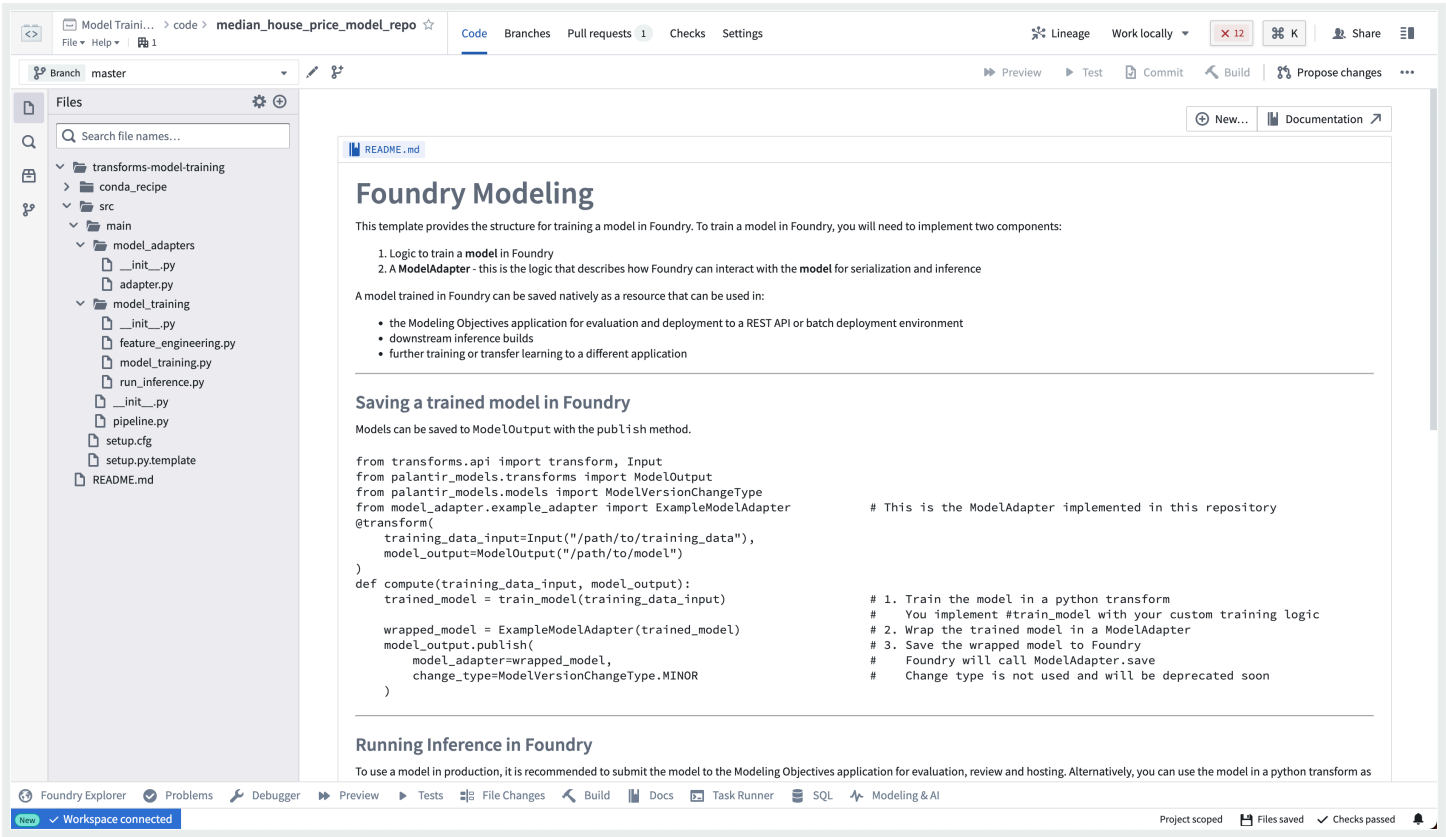








The model training template contains an example structure that we will adapt for this tutorial. You can expand the files on the left side to see an example project.




# 2b.2 How to split feature data for testing and training

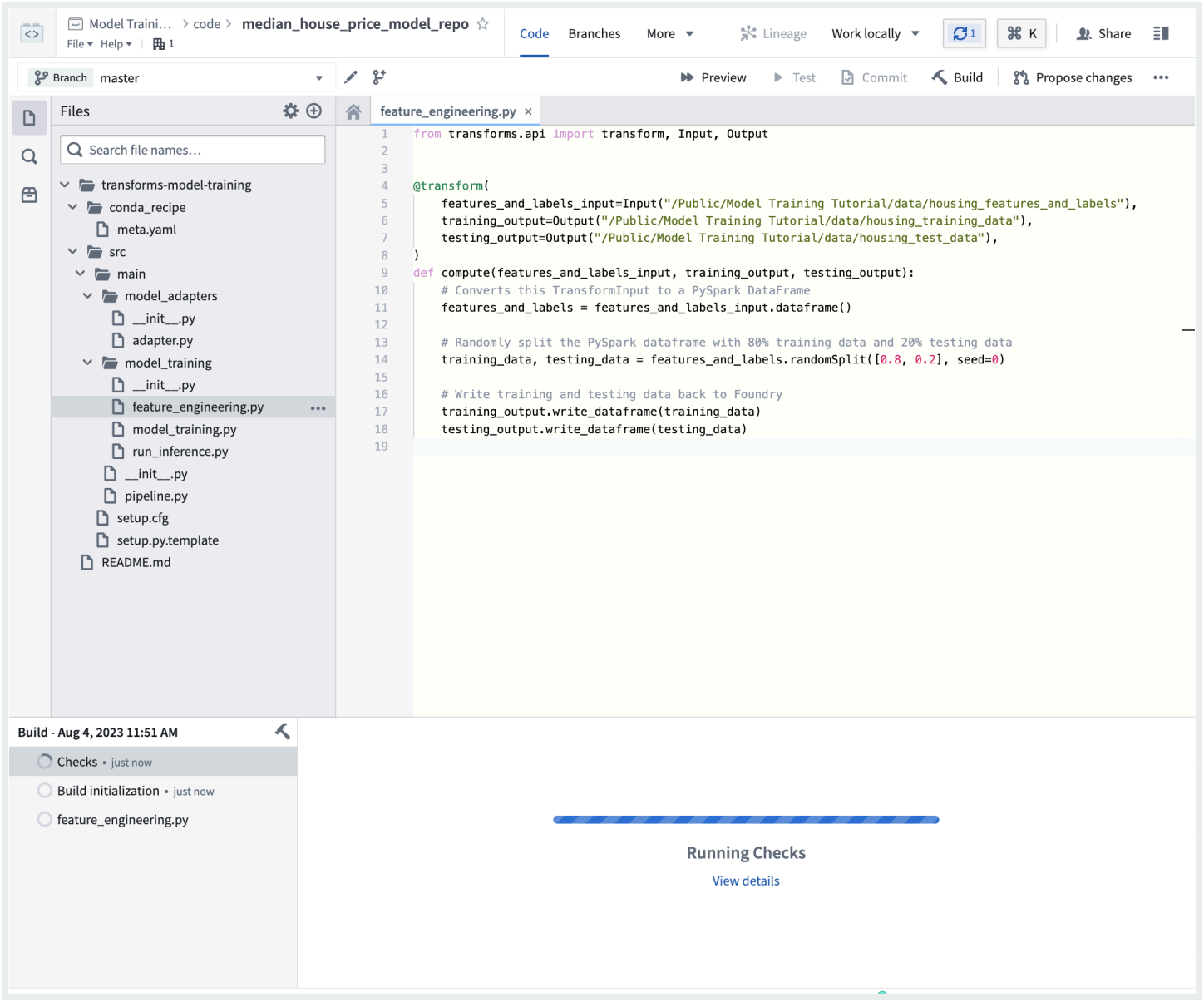
The first step in a supervised machine learning project is to split our labeled feature data into separate datasets for training and testing. Eventually, we will want to create performance metrics (estimates of how well our model performs on new data) so we can decide whether this model is good enough to use in a production setting and so we can communicate how much to trust the results of this model with other stakeholders. We must use separate data for this validation to help ensure that the performance metrics are representative of what we will see in the real world.

As such, we are going to write a Python transform that takes our labeled feature data and splits this into our two training and testing datasets.

```
from transforms.api import transform, Input, Output


@transform(
    features_and_labels_input=Input("<YOUR_PROJECT_PATH>/data/housing_features_and_l
    training_output=Output("<YOUR_PROJECT_PATH>/data/housing_training_data"),
    testing_output=Output("<YOUR_PROJECT_PATH>/data/housing_test_data"),
)
```

```
def compute(features_and_labels_input, training_output, testing_output):
    # Converts this TransformInput to a PySpark DataFrame
    features_and_labels = features_and_labels_input.dataframe()

    # Randomly split the PySpark dataframe with 80% training data and 20% testing dat
    training_data, testing_data = features_and_labels.randomSplit([0.8, 0.2], seed=0)

    # Write training and testing data back to Foundry
    training_output.write_dataframe(training_data)
    testing_output.write_dataframe(testing_data)
```

**Action**: Open the `feature_engineering.py` file in your repository and copy the above code into the repository. Update the paths to correctly point to the datasets you uploaded in the previous step of this tutorial. Select **Build** at the top left to run the code. You can, optionally, select **Preview** to test this the logic on a subset of the data for faster iteration.




You can continue with `2b.3` while this build executes.

# 2b.3 How to author model training logic in Code Repositories

Models in Foundry are comprised of two components, model artifacts (the model files produced in a model training job), and a model adapter (a Python class that describes how Foundry should interact with the model artifacts to perform inference).

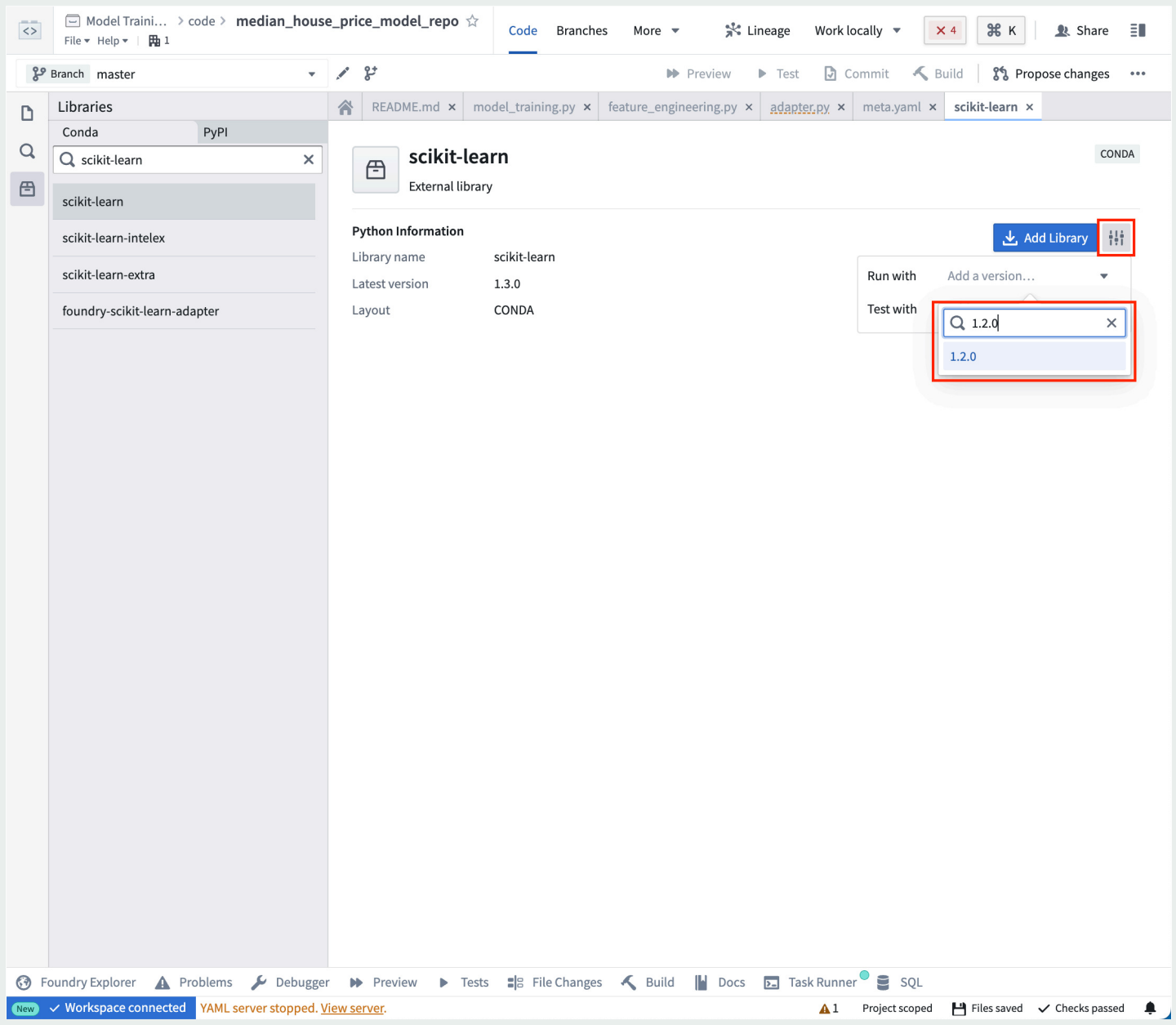
The model training template consists of two modules, `model_training` for the training job and `model_adapters` for the model adapter.

# Model dependencies

Model training will almost always require adding Python dependencies that contain model training, serialization, inference, or evaluation logic. Foundry supports adding dependency specifications through conda. These dependency specifications are used to create a resolved Python environment for executing model training jobs.

In Foundry, these resolved dependencies are automatically packaged with your models to ensure that your model automatically has all of the logic required to perform inference (generate predictions). In this example, we will use `pandas` and `scikit-learn` to produce our model and `dill` to save our model.

**Action:** On the left side bar, select **Libraries** and add dependencies for `scikit-learn = 1.2.0`, `pandas = 1.5.2` and `dill = 0.3.7`. Then select **Commit** to create a resolved Python environment.




# Model adapter logic

Model adapters provide a standard interface for all models in Foundry. The standard interface ensures that all models can be used immediately in production applications as Foundry will handle the infrastructure to load the model, its Python dependencies, expose its API, and interface with your model.

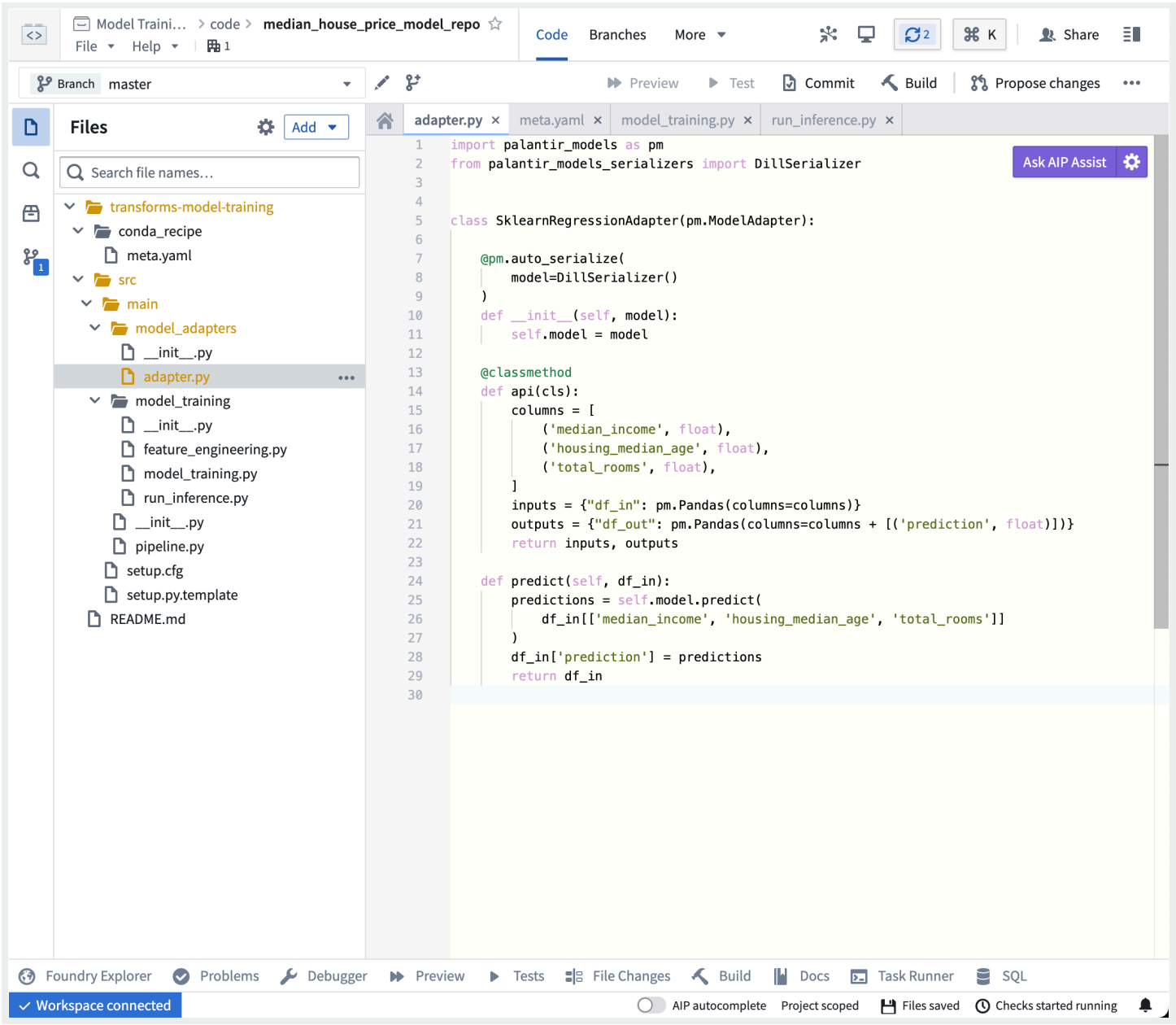
To enable this, you must create an instance of a `ModelAdapter` class to act as this communication layer.

There are 4 functions to implement:

1. **Model save and load:** In order to reuse your model, you need to define how your model should be saved and loaded. Palantir provides many default methods of serialization (saving), and in more complex cases you can implement custom serialization logic.
2. **api:** Defines the API of your model and tells Foundry what type of input data your model requires.
3. **predict:** Called by Foundry to provide data to your model. This is where you can pass input data to the model and generate inferences (predictions).

```python
import palantir_models as pm
from palantir_models_serializers import DillSerializer


class SklearnRegressionAdapter(pm.ModelAdapter):

    @pm.auto_serialize(
        model=DillSerializer()
    )
    def __init__(self, model):
        self.model = model

    @classmethod
    def api(cls):
        columns = [
            ('median_income', float),
            ('housing_median_age', float),
            ('total_rooms', float),
        ]
        return {"df_in": pm.Pandas(columns)}, \
               {"df_out": pm.Pandas(columns + [('prediction', float)])}

    def predict(self, df_in):
        df_in['prediction'] = self.model.predict(
            df_in[['median_income', 'housing_median_age', 'total_rooms']]
        )
        return df_in
```

**Action** Open the `model_adapters/adapter.py` file and paste the above logic into the file.




# Model training Logic

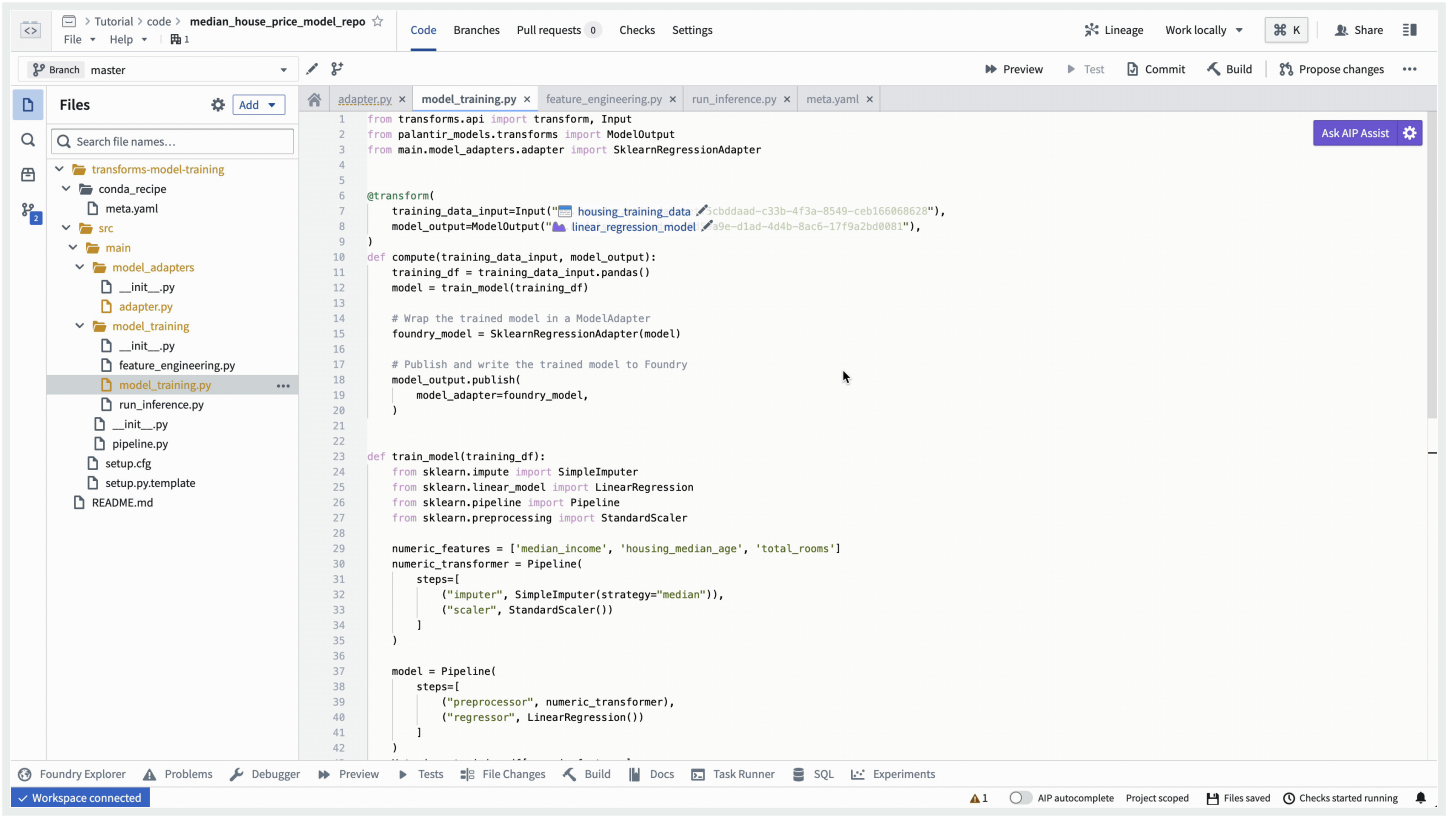
Now that our dependencies are set and we have written a model adapter, we can train a model in Foundry.

```
from transforms.api import transform, Input
from palantir_models.transforms import ModelOutput
from main.model_adapters.adapter import SklearnRegressionAdapter


@transform(
    training_data_input=Input("<YOUR_PROJECT_PATH>/data/housing_training_data"),
```

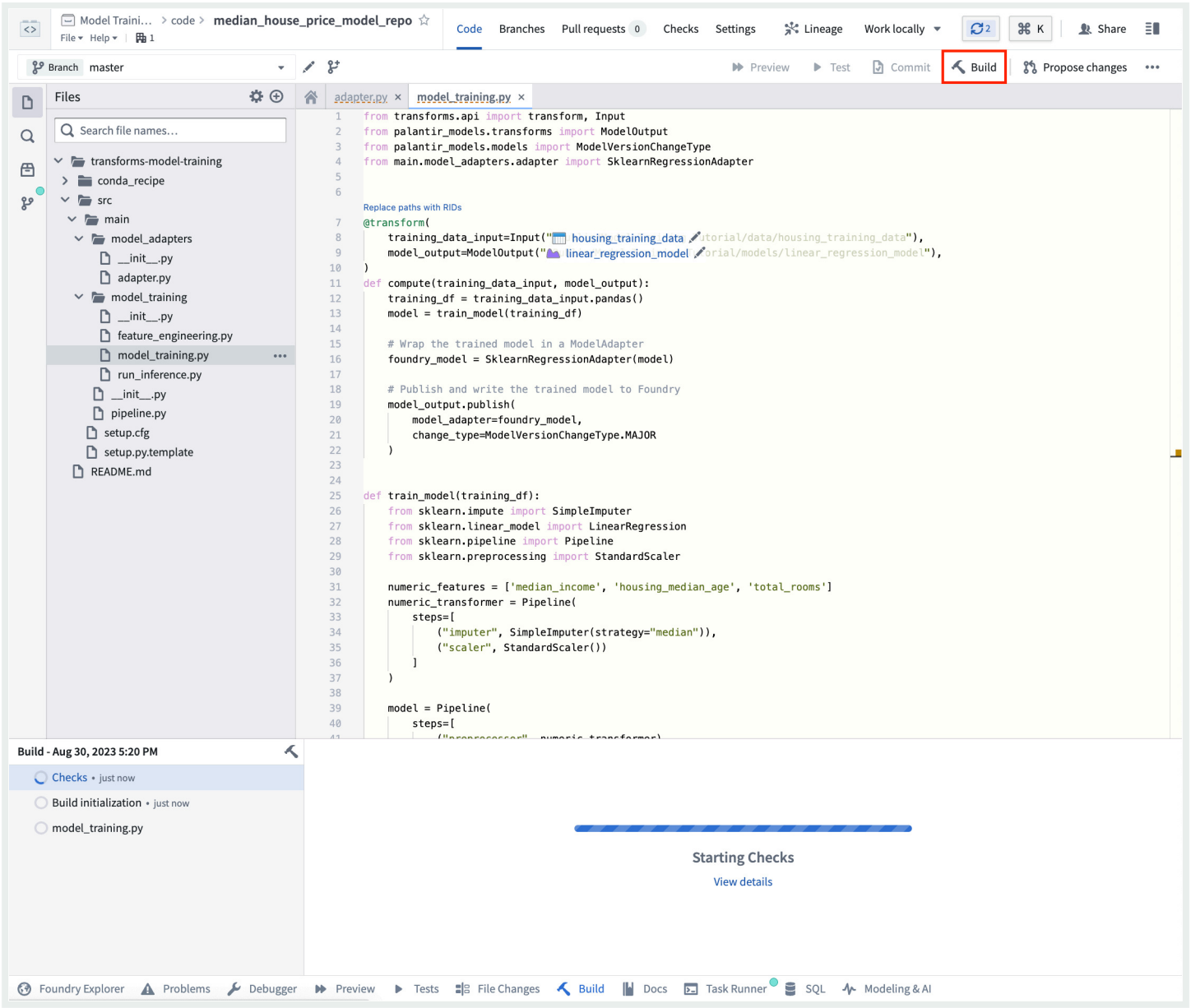
```
    model_output=ModelOutput("<YOUR_PROJECT_PATH>/models/linear_regression_model"),
)
def compute(training_data_input, model_output):
    training_df = training_data_input.pandas()
    model = train_model(training_df)

    # Wrap the trained model in a ModelAdapter
    foundry_model = SklearnRegressionAdapter(model)

    # Publish and write the trained model to Foundry
    model_output.publish(
        model_adapter=foundry_model
    )


def train_model(training_df):
    from sklearn.impute import SimpleImputer
    from sklearn.linear_model import LinearRegression
    from sklearn.pipeline import Pipeline
    from sklearn.preprocessing import StandardScaler

    numeric_features = ['median_income', 'housing_median_age', 'total_rooms']
    numeric_transformer = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="median")),
            ("scaler", StandardScaler())
        ]
    )

    model = Pipeline(
        steps=[
            ("preprocessor", numeric_transformer),
            ("classifier", LinearRegression())
        ]
    )
    X_train = training_df[numeric_features]
    y_train = training_df['median_house_value']
    model.fit(X_train, y_train)

    return model
```

**Optional:** When you are iterating on model training and model adapter logic, it can be useful to test your changes on a subset of your training data before running a build. Select **Preview** at the

top left to test your code.




**Action:** Open the `model_training/model_training.py` file in your repository and copy the above code into the repository. Update the paths to correctly point to the training dataset and model folder you created in the step 1.1. Select **Build** at the top left to run the code.




## (Optional) Log metrics and hyperparameters to a model experiment

Model experiments is a lightweight framework for logging metrics and hyperparameters produced during a model training run, which can then be published alongside a model and persisted in the model page.

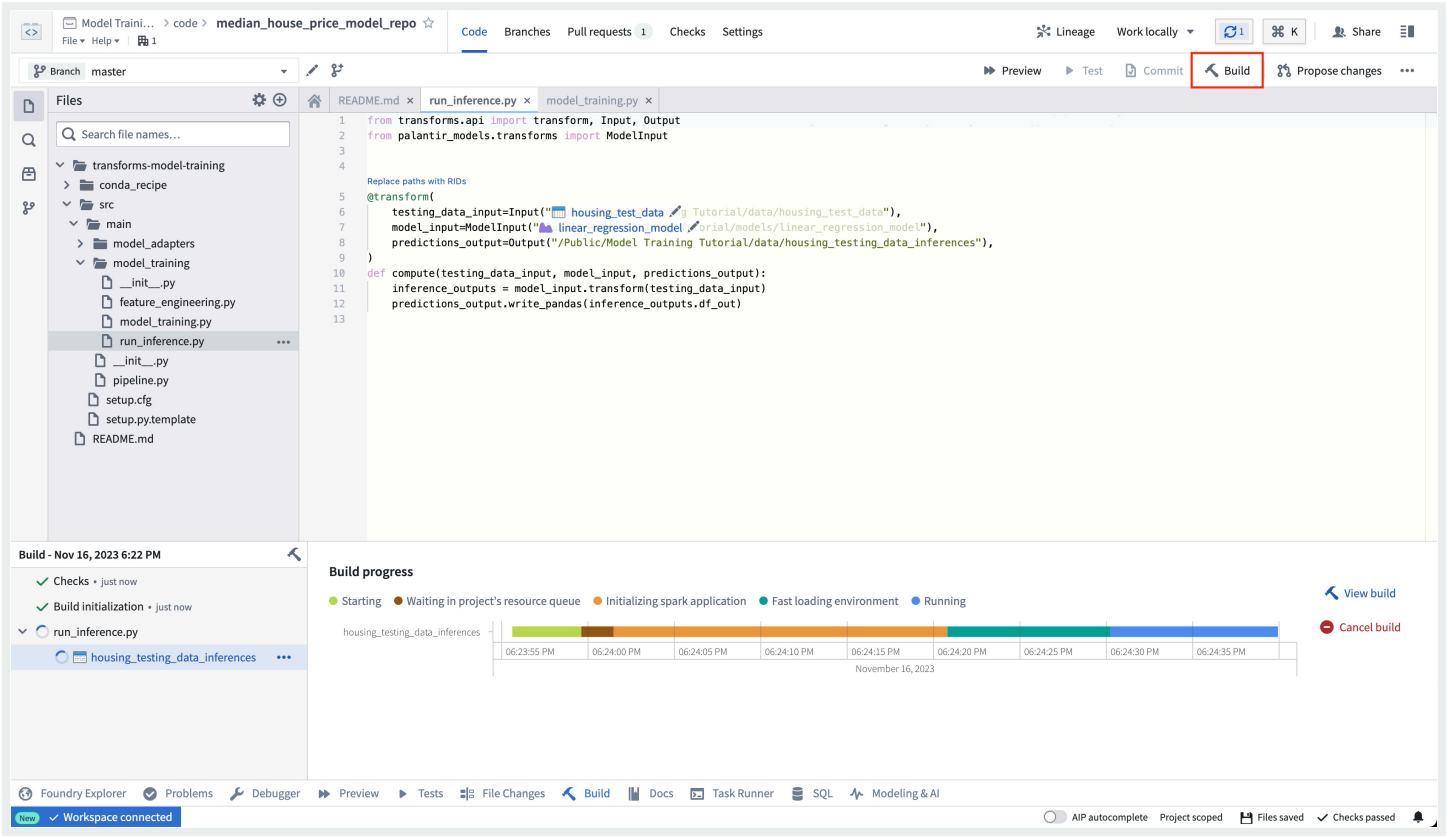
Learn more about creating and writing to experiments.

# 2b.4 How to test inference logic in Code Repositories

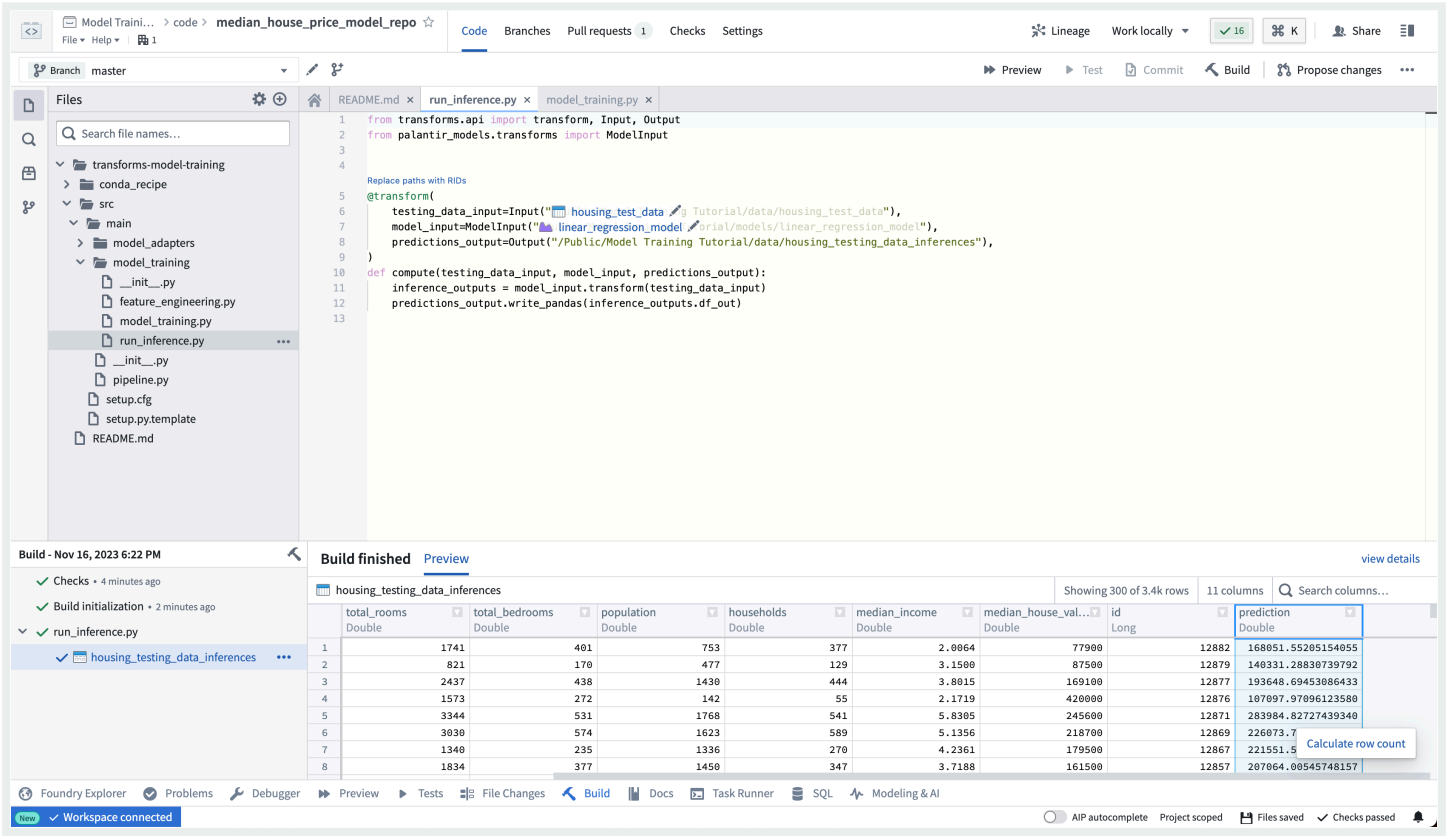
Once your model training logic has finished you can generate predictions (also known as inferences) directly in your code repository.

```python
from transforms.api import transform, Input, Output
from palantir_models.transforms import ModelInput


@transform(
    testing_data_input=Input("<YOUR_PROJECT_PATH>/data/housing_test_data"),
    model_input=ModelInput("<YOUR_PROJECT_PATH>/models/linear_regression_model"),
    predictions_output=Output("<YOUR_PROJECT_PATH>/data/housing_testing_data_inferen
)
def compute(testing_data_input, model_input, predictions_output):
    inference_outputs = model_input.transform(testing_data_input)
    predictions_output.write_pandas(inference_outputs.df_out)
```

**Action:** Open the `model_training/run_inference.py` file in your repository and copy the above code into the repository. Update the paths to correctly point to the model asset and test dataset you created earlier. Select **Build** at the top left to run the code.



Once your build is complete, you can review the generated predictions in the build output panel.

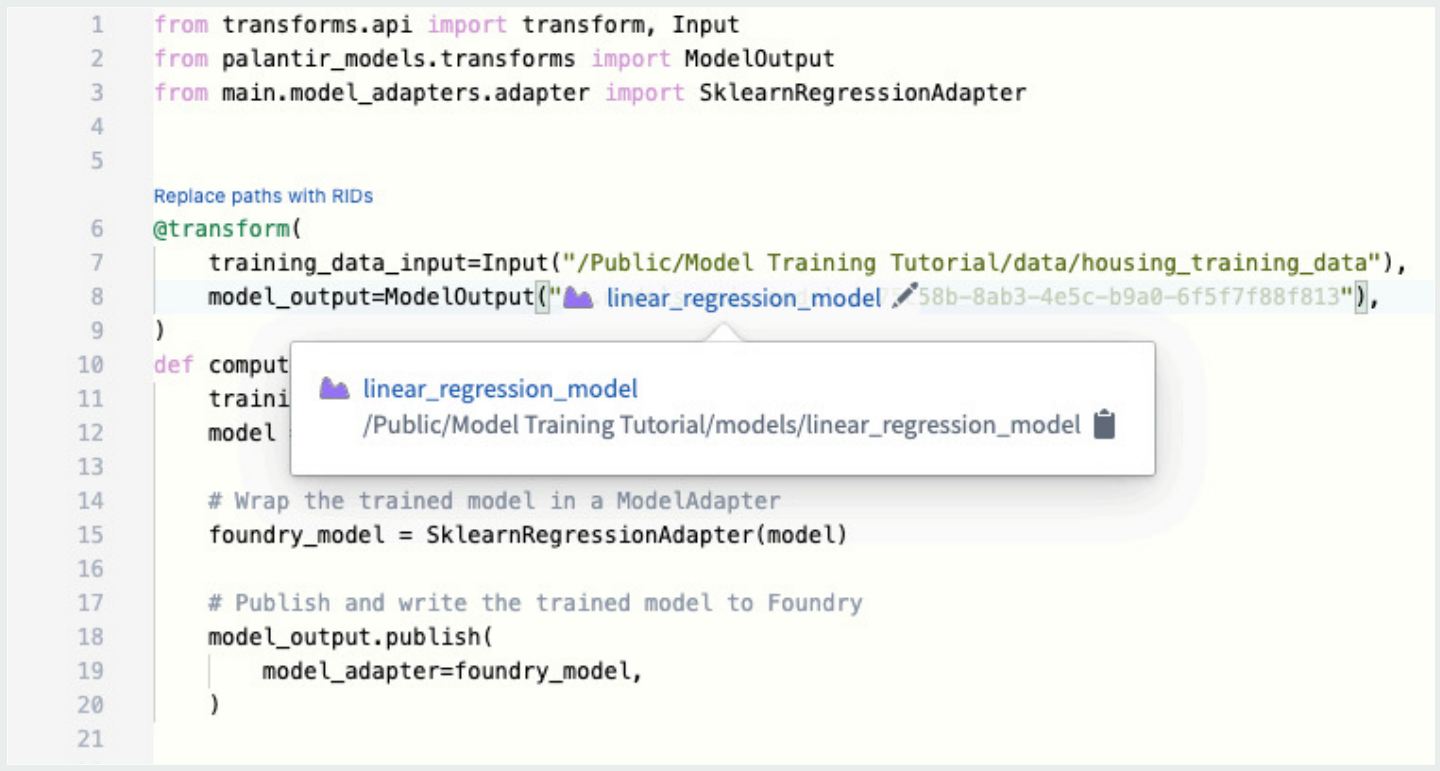



# 2a.5 Optional: Configure live inference

Optionally, this model can be consumed as a REST API via a direct deployment. Learn how to configure a direct deployment.

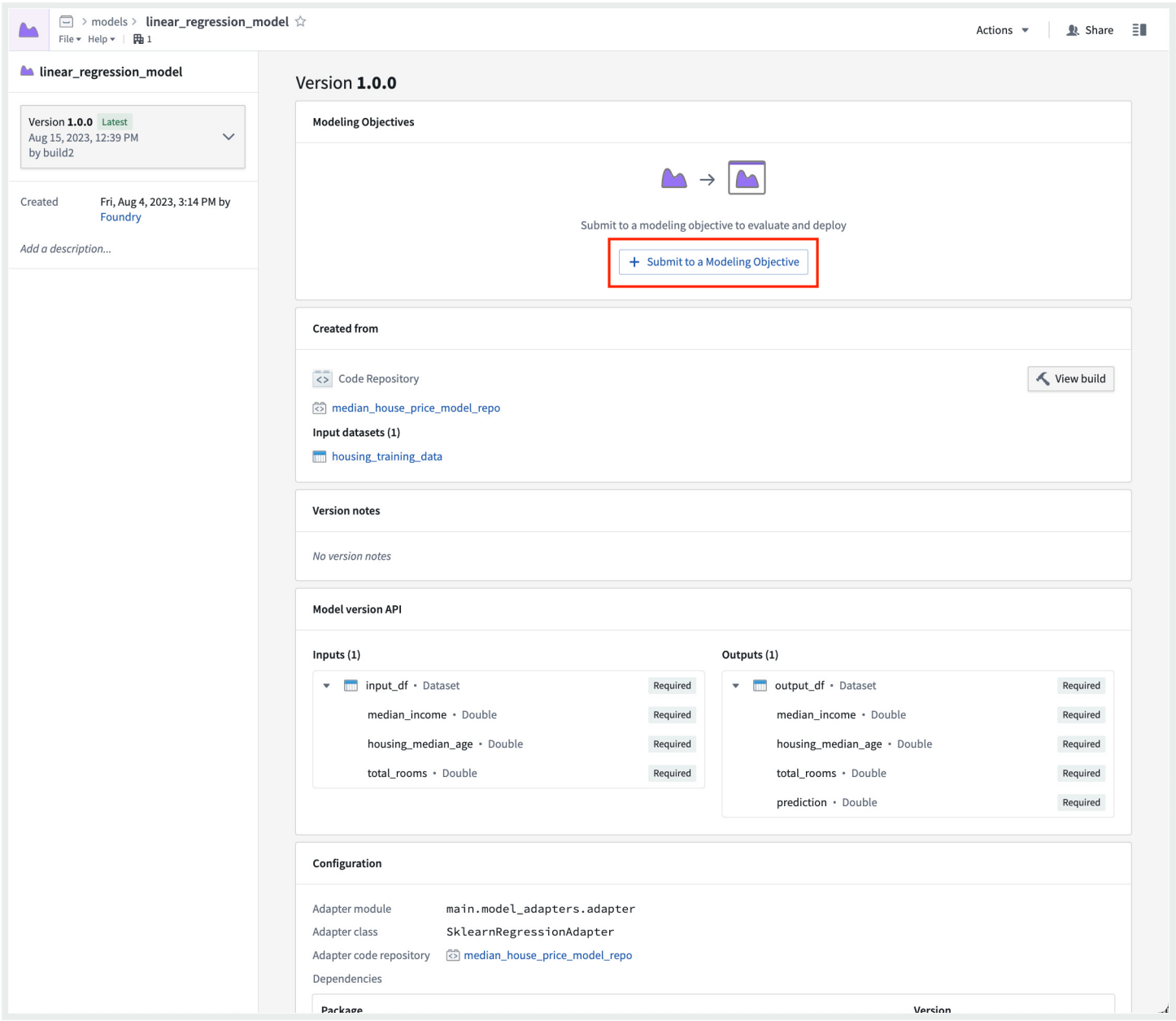
# 2b.6 How to view a model and submit it to a modeling objective

After your model is built you can open the model either by selecting `linear_regression_model` in the `model_training/model_training.py` file or by navigating to the model in the folder structure we created earlier.



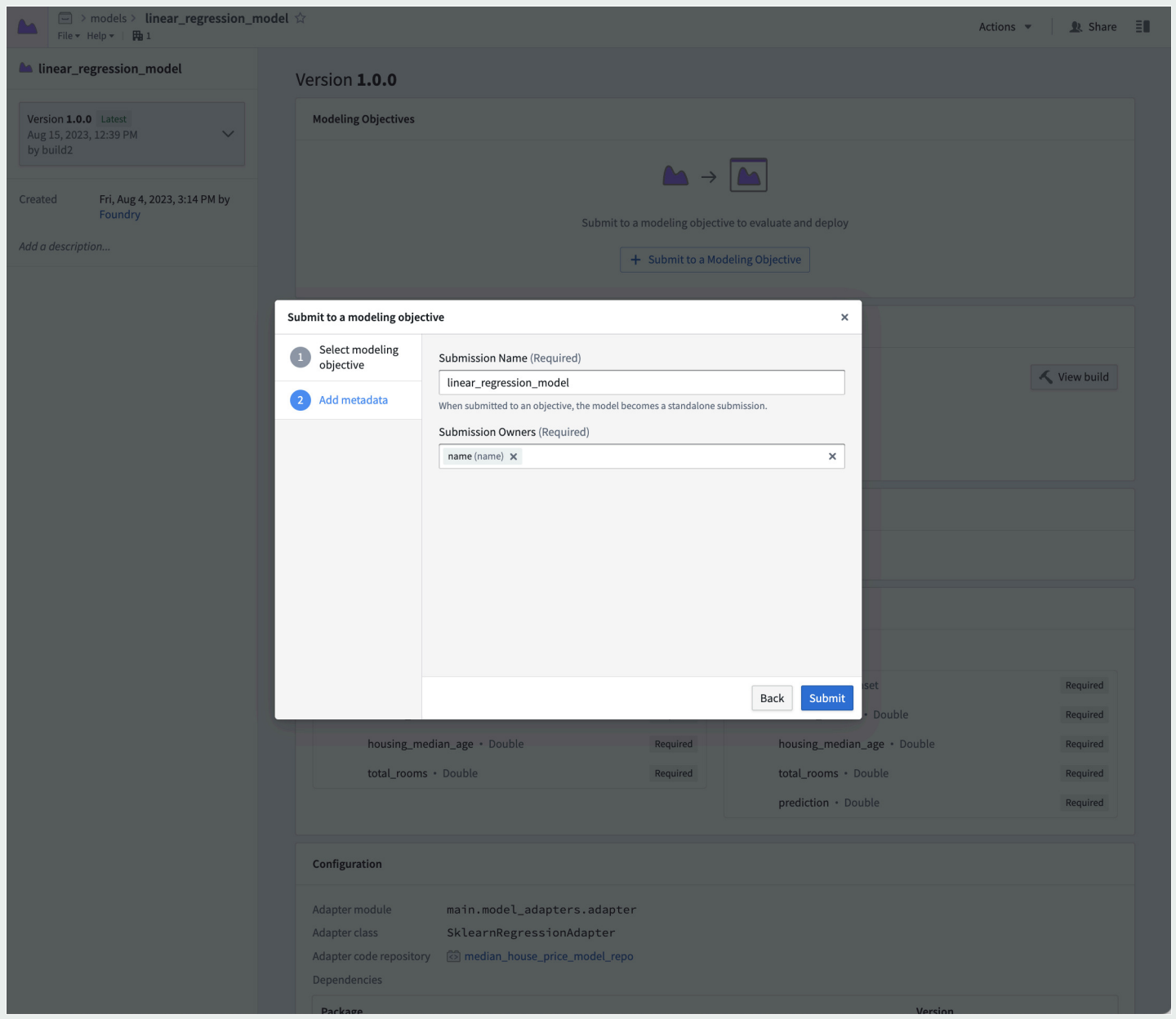

The model view has the source of where the model was trained, the training datasets used to produce this model, the model API, and the model adapter this model was published as. Importantly, you can publish many different versions to the same model; these model versions are available in the dropdown menu on the left sidebar.

As the model version is connected to the specific model adapter used during training, you need to republish and build your model training process to apply any changes to the model adapter logic.




Now that we have a model, we can submit that model to our modeling objective for management, evaluation, and release to operational applications.

**Action:** Select **linear_regression_model** in the code to navigate to the model asset you have created, select **Submit to a Modeling Objective** and submit that model to the modeling objective you created in step 1 of this tutorial. You will be asked to provide a submission name and submission owner. This is metadata that is used to track the model uniquely inside the modeling objective. Name the model `linear_regression_model` and mark yourself as the submission owner.




# Next step

Now that you have trained a model in Foundry, you can move onto model management, testing, and model evaluation. Here are some examples of additional steps you can take in Modeling Objectives:

- Automatic model evaluation
- Configuring checks for model submissions
- Live and batch inference can also be configured from the modeling objective.

Optionally, you can also train a model in a Jupyter® notebook with the Code Workspaces application for fast and iterative model development.

---